# Experiments with Hierarchical Clustering Algorithms as Software Remodularization Methods*

Nicolas Anquetil, Cédric Fourrier and Timothy C. Lethbridge
School of Information Technology and Engineering
150 Louis Pasteur, University of Ottawa
Ottawa, Canada, K1N 6N5
(1) (613) 562-5800 x6688
{anquetil,tcl}@site.uottawa.ca

## Abstract

*As valuable software systems become older, reverse engineering becomes increasingly important to companies that have to maintain the code. Clustering is a key activity in reverse engineering that is used to discover improved designs of systems or to extract significant concepts from code.*

*Clustering is an old, highly sophisticated, activity which offers many methods to meet different needs. The various methods have been well documented in the past, however conclusions from the general clustering literature may not apply entirely in the reverse engineering domain. In this paper, we study three things: some clustering algorithms, some metrics that quantify the coupling between entities to be clustered, and how these entities are represented abstractly. Our objective is to establish whether and why each could be used for software remodularization.*

*The results are compared using three quality criteria: design quality (cohesion and coupling), comparison with an expert decomposition, and size of the clusters obtained. The experiments were conducted on three public domain systems (gcc, Linux and Mosaic) and a legacy telecommunications system.*

*Among our findings, we confirm the importance of carefully choosing the scheme that is used to describe the entities being clustered.*

## 1. Introduction

As software becomes older, its maintenance becomes more and more of an issue. Software engineers face systems where a minor change in some part can have an unexpected impact on an apparently unrelated part. Consequently, a goal of the reverse engineering community is to help software engineers understand and restructure old software as well as to migrate it towards more modern architecture and languages. A key technique for doing this is clustering, which is used to gather software components into modules that are significant to the software engineers.

Clustering, however, is a sophisticated research domain in itself. There are many alternative methods, the choice of which depends on such factors as size of the data, the type of data, a priori knowledge of the expected result, etc. Reverse engineering is a younger research domain, with unclear goals [7] and methods that are somewhat ad-hoc. In this context, clustering has often been used with no deep understanding of all the issues involved.

To try to address this, Wiggerts presents in [24] a summary of the literature on clustering. This summary is valuable since it lists all the possible decisions one needs to make and give insights about advantages and drawbacks associated with various alternatives. For example, he restates the fundamental rule, often overlooked, that clustering algorithms do not *discover* some hidden structure, but rather *impose* one. It is therefore all the more important to choose the right clustering method to impose the structure most suitable to our needs. However, Wiggerts paper offers conclusions drawn from the general clustering literature, some of which may not apply to the domain of software remodularization. We present here a comparative study of different hierarchical clustering algorithms and analyze their properties with regard to software remodularization. We studied three issues: how entities to be clustered are described, the similarity metrics to quantify the coupling between these entities and a choice of hierarchical clustering algorithms.

The organization of the paper is the following. First we analyze each of the above three issues, then we discuss the three quality metrics we used to compare the results: design quality, comparison with an expert decomposition and size

*This work is supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER).

of the clusters. Finally we present and discuss experimental results.

## 2 Issues in Clustering

When one decides to use a clustering algorithm to perform architecture recovery, there is a wide variety of decisions to be made. We will consider three:

- The entities to be clustered and how they are represented;
- the similarity metrics between the entities;
- the clustering algorithms.

For each point, we will briefly summarize the taxonomic point of view and the expected behavior for remodularization.

### 2.1 Entities to Cluster and their Descriptions

Clustering is widely used in the field of taxonomy, for example to group species in biology. Taxonomists describe entities using *features* like "number of leaves", "laying eggs" or not, etc. The result of the clustering clearly depends on this choice of feature. In fact, the issue deserves its own study and we will only draw some preliminary conclusions in this paper.

For software remodularization, entities may be files, routines, classes, processes, etc. The features used to describe them are usually "references" from an entity to some other set of software artifacts (see [11]). For example to cluster routines, one might use the references to the global variables they have in common. One may also use several different features to create a more complex description. In this paper, we choose to use the following among the possible descriptive features[1]:

- References to user defined types. A type is referenced when it's used to define a variable for example. This feature has been used in [10, 17] to describe files or in [6] to describe routines.
- Calls to routines. Used in [16, 23] to describe files or in [3] to describe classes.
- Inclusion of files. Only used to describe files as in [13, 23].
- Use of macros. This feature is only available in some languages (e.g. C).

For example, the five files of figure 1 could be described according to the other files they include (the third of the features proposed above):

[1]For a more exhaustive review see [11].

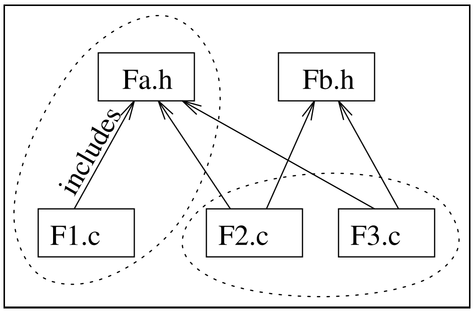


**Figure 1. Some files and possible clusters based on their inclusion.**

Fa.h: ()
Fb.h: ()
F1.c: (Fa.h)
F2.c: (Fa.h, Fb.h)
F3.c: (Fa.h, Fb.h)

It is common to use a geometrical analogy and consider the description of each entity as a *vector* in the space of all types (or routines, or files, ...). Each type (or routine, or file) is a *dimension* in this space and the coordinate for this dimension is the number of references found in the entity described. Where there is no reference to a given descriptive element (i.e. there is a missing dimension), the weight will be zero. The above descriptions might then become:

Fa.h: (Fa.h/0, Fb.h/0)
Fb.h: (Fa.h/0, Fb.h/0)
F1.c: (Fa.h/1, Fb.h/0)
F2.c: (Fa.h/1, Fb.h/1)
F3.c: (Fa.h/1, Fb.h/1)

In taxonomy, the features may come from many different sources, based on anatomy, geographical distribution, temporal distribution, behavior, chemical properties, etc. For software remodularization the choice seems more limited: All features are extracted from the source code; and in many languages there are strong interdependencies between the different features[2]. For example, to reference an externally defined type, one needs to include some file that describes it; if a routine refers to a type as one of its parameters' types, then a calling routine should also refer to this type as one of its variables' types.

The choice of features is also limited by the amount of information they can provide for a given system. For example, a system may have too few global variables to allow proper clustering.

[2]There are also interdependencies in taxonomy, but to a lesser extent.

It is our thesis (see [1]) that a larger variety of features should be made available so that clustering can be done more effectively. Consequently, we also experimented with a naming convention feature:

- References to words. All identifiers contained in the entities are decomposed into words following simple word markers (underscore character, capital letter at the beginning of words, etc.).

## 2.2 Similarity Metrics

Similarity metrics are used to compare the entities and decide which ones should be clustered together (similar entities) and which ones should not. One can also see similarity metrics as measures of coupling (see also 3.2) between the entities, the higher the similarity, the higher the coupling.

There is a large number of similarity metrics, and they may be grouped in the following four broad categories (from [20]): Distance coefficients, association coefficients, correlation coefficients and probabilistic coefficients. We experimented with the first three. Many of the metrics use a geometrical analogy and consider the description of each entity as a *vector* in the space of all types, routines, words, etc. (depending on the descriptive feature chosen). Each element of the vector (each type, routine, etc.) is a *dimension* in this space and the coordinate for this dimension is the number of references found in the entity described:

**Association coefficients:** These compare the references two entities have in common only considering whether dimensions are zero (absent) or not. Similarity between two entities $X$ and $Y$ is expressed using four quantities: $a = \|X \cap Y\|$, $b = \|X \setminus Y\|$, $c = \|Y \setminus X\|$ and $d = \|\mathcal{F} \setminus (X \cup Y)\|$ where $\mathcal{F}$ is the set of all possible dimensions. The quantity $d$ is usually much larger than $a$, $b$ or $c$ which may cause some difficulties (see the end of this subsection).

**Distance coefficients:** Entities are considered as geometrical points and the coefficients compute the distance (e.g. Euclidean distance) between these points. These distances consider zero-dimensions.

**Correlation coefficients:** Compute the linear correlation between values for all the dimensions[3]. The $[-1, 1]$ linear correlation interval is then mapped into a $[0, 1]$ similarity interval. There are different mappings, some consider a strong negative correlation as a sign of similarity, others don't. We opted for the second possibility (negative correlation is a sign of dissimilarity). Although it does consider zero-dimensions, this coefficient will give good results. Given its complexity, we have no explanation for this behavior.

**Probabilistic coefficients:** Compute the probability that two entities are similar given their respective vectors. We did not experiment with this one for practical reasons. According to Sneath [20], they are close to the correlation coefficients.

We experimented[4] with two distance coefficients, three association coefficients and one correlation coefficient, all proposed in [20]:

**Correlation:** (correlation[3])
$sim(X, Y) = \sqrt{(1 - r)/2}$

**Jaccard:** (association[5])
$sim(X, Y) = 1 - (a/(a + b + c))$.

**Simple Matching:** (association[5])
$sim(X, Y) = 1 - ((a + d)/(a + b + c + d))$.

**Sørensen-Dice:** (association[5])
$sim(X, Y) = 1 - ((a + d)/2a/(2a + b + c))$.

**Taxonomic:** (distance)
$sim(X, Y) = \sqrt{\sum_{i=1}^{\|\mathcal{F}\|} (x_i - y_i)^2}$ where $x_i$ and $y_i$ are coordinates of respectively $X$ and $Y$.

**Camberra:** (distance)
$sim(X, Y) = \sum_{i=1}^{\|\mathcal{F}\|} (|x_i - y_i|/(x_i + y_i))$.

For example the similarity between F1.c and F2.c of figure 1 (preceding subsection) according to Jaccard with a descriptive feature based on file inclusion would be: $a = \|\{\text{Fa.h}\}\| = 1$, $b = \|\emptyset\| = 0$, $c = \|\{\text{Fb.h}\}\| = 1$, $sim(\text{F1.c}, \text{F2.c}) = 1 - (1/(1 + 0 + 1)) = 0.5$

These metrics, useful in taxonomy, may not be well adapted to software remodularization. In taxonomy, the number of dimensions for one feature is much less than it is for software remodularization. A feature like references to routines may have tens of thousands of dimensions in a real legacy system, yet most entities will be linked to only a few of these routines. As a rule, for each entity, the description vector is therefore very sparse, with many zero-dimensions. We distinguish two particular conditions that may raise problems:

**Empty description:** Some entities have no references at all for a given feature. The vectors describing such entities all lie at the origin.

[3] $r = \dfrac{\sum XY - \left(\sum X \sum Y\right)/n}{\sqrt{\left(\sum X^2 - \left(\sum X\right)^2/n\right)\left(\sum Y^2 - \left(\sum Y\right)^2/n\right)}}$

[4] Note that the formulas are actually for dissimilarity metrics (i.e. distance metrics). The higher the similarity, the lower the dissimilarity. This will greatly simplify the discussion on height of cluster in a later section.

[5] see the definition for $a$, $b$, $c$ and $d$ above.

**Quasi-empty description:** Other entities have very few references for a given feature. They will be described by a vector with only one or two non-zero dimensions.

Clearly, the entities with an empty description raise a problem: How can we cluster them if they have no connection to any other entity? We chose to treat the empty descriptions separately in all the similarity metrics and consider such entities infinitely dissimilar to any other entity. This yields a number of singleton clusters (the entities with empty descriptions) left after the clustering process.

Entities with quasi-empty descriptions also raise a problem: Some metrics, which consider zero-dimensions as a sign of similarity, will tend to cluster such entities together: Since they have very similar descriptions, they are considered to be strongly coupled. These metrics make sense in taxonomy, however with the descriptive features used in software engineering, they will be misled by the high proportion of zero-dimensions. This will be the case for the Simple Matching association coefficient and the two distance coefficients. The quasi-empty descriptions can form a core cluster which will attract to it all other entities with only a few more non-zero-dimensions. The clustering method would ultimately gather all the entities into one single cluster, rendering it useless.

### 2.3 Clustering Algorithms

The next step consists of choosing a clustering algorithm. We will concentrate on agglomerative hierarchical algorithms. For software remodularization, non hierarchical algorithms have also been used (see [11]). Agglomerative hierarchical algorithms start from the individual entities, gather them into small clusters which are in turn gathered into larger clusters up to one final cluster that contains every entity. The result is a binary tree of clusters. A notable advantage of such algorithms is that they are unsupervised, that is they don' t need any extra information like the number of clusters expected and the possible region of the search space in which to look for each cluster. On the other hand, we will see in the next subsection that it can be difficult to choose which clusters in the final hierarchy are valuable and which ones only represent partial results.

these algorithms are differentiated by the way they compute the distance from a new cluster to all other ones[6]. The distance between a newly created cluster (e.g. *C4*, in figure 2), to another one (*C3*) is computed from the distances of its two members (*C1* and *C2*) to this other one. There are four main algorithms:

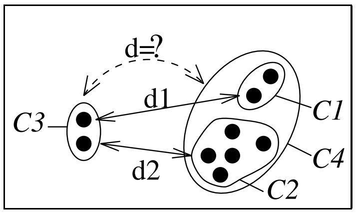


**Figure 2. Distance of a new cluster (*C4*) to other clusters (*C3*) in agglomerative hierarchical clustering algorithms.**

**Single linkage[7]:** (or closest neighbor rule) the new distance will be $d = min(d_1, d_2)$.

**Complete linkage:** (or furthest neighbor rule) $d = max(d_1, d_2)$.

**Weighted average linkage:** $d = (d_1 + d_2)/2$ Since clusters *C1* and *C2* may not have the same number of entities, these entities have different weights depending on which cluster they belong to.

**Unweighted average linkage:** In this one, we take into account the size of the clusters *C1* and *C2*, therefore all entities have the same weight (i.e. they are not weighted). $d = (\|C_1\|d_1 + \|C_2\|d_2)/(\|C_1\| + \|C_2\|)$

The above algorithms influence the result. For example, single linkage is known to favor non-compact but more isolated clusters whereas, complete linkage usually results in more compact but less isolated clusters (see figure 3). Therefore, single linkage should give less coupled clusters and complete linkage should give more cohesive ones (cohesion and coupling are two measures of the design quality of a partition). Unweighted and weighted average linkages are intermediate on the scale: complete, weighted, unweighted and single linkages.

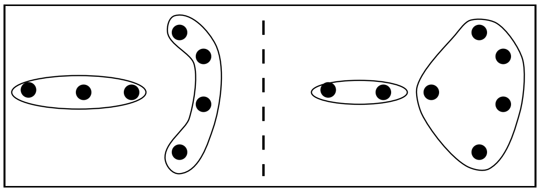

**Figure 3. Influence of the clustering algorithm on the clusters (from [24]). Left: Single linkage. Right: Complete linkage.**

There is no a-priori reason to favor cohesion or coupling, but because the descriptive features (on which cohesion and

[6]Similarity metrics compute similarity between individual entities. The algorithms extend this to clusters.

[7]Do no confound the *Simple* Matching coefficient (§2.2) with the *single* linkage algorithm.

**Table 1. Summary of the three clustering parameters studied. “Zero” identifies the similarity metrics which do consider null-dimensions.**

| Descriptive Feature (reference to ...) | Similarity Metric: category | Similarity Metric | zero | Algorithm |
|---|---|---|---|---|
| types | | Jaccard | | |
| macros | *association* | Simple Matching | √ | Complete linkage |
| routines (call) | | Sørensen-Dice | | Weighted linkage |
| files (include) | *correlation* | Correlation | √ | Unweighted linkage |
| | *distance* | Taxonomic | √ | Single linkage |
| words in ident. | | Camberra | √ | |

coupling are based, see §3.2) used are very sparse, coupling naturally tends to be good (i.e. low). As a consequence, we propose to give more importance to cohesion. Note that this approach departs from the traditional one which attempts to obtain both good cohesion and coupling. It is our experience that one often needs to choose to favor one metric at the expense of another.

Table 1 gives a summary of the three parameters we have studied.

### 2.4 System Partitions

For software remodularization, we often need a partition of the system instead of a hierarchy of clusters. The partition may be obtained by pruning the hierarchy of clusters at the appropriate height and considering only the top-most clusters (see figure 4). If we cut at height 0, the partition will contain only singleton clusters. If we cut at the maximum height, the entire system will be the only cluster.

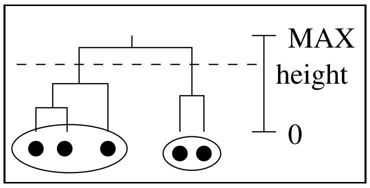


**Figure 4. A hierarchy of clusters and how to cut it to get a partition of the data set.**

One can draw a useful analogy between the height of a cluster and its “diameter”. The farther apart two entities (less similarity), the greater the height of a cluster containing them both. The height of a cluster is always greater than the height of the clusters it contains. This height (or diameter) is directly dependent on the similarity metric chosen. For example, with the association coefficients, the maximum height will be 1, which, for the formulas we gave in §2.2 is the farthest apart[8] two entities can be. With other similarity metrics, notably the distances, the maximum can be much higher. We see heights over 150 in our results.

Finding the appropriate height at which to cut is a difficult problem in itself and there may be more than one choice. We did not attempt to solve this problem; instead, we sliced the hierarchy at many different heights, thus obtaining a succession of “cuts” for each hierarchy. This proved useful to analyze the behavior of each clustering method.

## 3 Quality Criteria

Quality criteria are central to any formal comparison as well as to the precise definition of a scientific research domain. Clustering is often used to remodularize an old software system to help maintainers understand and manage it. As such, we propose that an ideal clustering method should have the following properties:

- Actually represent the system (and not an ideal view of the domain for example).
- Make sense to the software designers (extract modules that implement known concepts).
- Be adaptable to different needs (help novices understand the system and also help experts assert the consequences of a modification in the code).
- Be general (adaptable to different software systems, programming languages, etc.)

Remodularization techniques are often evaluated manually by an expert in the system. Due to the very large number of experiments we performed (thousands), this solution was not acceptable. We found the following automated criteria in the reverse engineering literature:

[8] The “similarity” formulas given actually measure the dissimilarity (or distance) between entities.

**Design:** Clusters should reflect a good design (as evaluated by cohesion and coupling).

**Expert decomposition:** Clusters should match an "expert decomposition".

To these, we added:

**Size:** In order to be useful, clusters should have an appropriate size. For example, one large cluster and many small ones is not good.

### 3.1 Expert Criterion

The source files for two of our systems (Linux and Mosaic) are organized in several directories which form a reasonable decomposition of the systems. We used these as "expert partitions" against which to compare the clusters obtained. These decompositions may not be ideal, but, in both systems, they give a good indication about the designers' intent. The fact that we experimented at a high level of granularity (clustering files into subsystems), helped to make this criterion valid. Files are less sensitive to design drift through maintenance.

Although, the clustering algorithm and the directories both potentially provide tree structures, we compared them as partitions of the system. We already explained how to cut a hierarchy of clusters to obtain a partition of the system. For the directory tree, we considered the lower directories (the one containing only files as opposed to possible subdirectories) as forming subsystems. This should match the abstraction level of the clusters extracted which is usually low.

The difficulty of this criterion is to compare two sets of clusters, when the clusters are only defined by their members. A question arises about how to compare two clusters when one of them has one entity more than the other. It is clear that they are not equal, but we need to acknowledge the fact that there is very little difference between them. We propose considering pairs of entities: Two entities are either in the same cluster (an "intra" pair) or in two different clusters (an "inter" pair). We can then use precision and recall[9] for a given partition as:

**Precision:** Percentage of intra pairs proposed by the clustering method which are also intra in the expert partition.

**Recall:** Percentage of intra pairs in the expert partition which were found by the clustering method.

A partition containing only singleton clusters has good precision (all "zero" of the intra pairs it proposes are in the expert partition) and poor recall. One huge cluster containing all the system has good recall (all the intra pairs in the expert partition are also intra according to the clustering) and poor precision. We will often witness a very good precision and low recall. This is a consequence of the clustering methods extracting smaller clusters than the one in the expert partition. Since the clusters are smaller, they have fewer intra pairs and a higher probability that these are correct (good precision), but there will be many intra pairs from the expert decomposition that will not be discovered (bad recall). In other words, the extracted clusters are at a lower abstraction level than the expert partition.

### 3.2 Design Criterion

Well-designed clusters are more likely to be of interest to the software engineers. This is traditionally evaluated using *cohesion* and *coupling* (e.g. [21]) which we will now briefly describe. Cohesion and coupling are both based on pair-wise coupling between entities. This in turn is computed with a scheme similar to that used for clustering. Entities are described using one of the features already discussed §2.1, and the coupling between entities is evaluated by some kind of similarity metric. We use a different similarity metric than those described in §2.2, one proposed in [10] (see also [17]). Our metrics are as follows:

**Similarity** between entities[10] is:
$sim(e_X, e_Y) = (X^T.Y)/(\|X\|.\|Y\|)$.

**Cohesion** of a partition is the average similarity between any two entities clustered together (i.e. any intra pair, see previous section). Cohesion ranges from 0 (worst) to 1 (best).

**Coupling** of a partition is the average similarity between any two entities in two different clusters (i.e. any inter pair). Coupling ranges from 0 (best) to 1 (worst).

For example the similarity between files F1.c and F2.c of the example in figure 1 are described by the following vectors: $\overrightarrow{\text{F1.c}} = (1\ 0)$, $\overrightarrow{\text{F2.c}} = (1\ 1)$, $sim(\text{F1.c}, \text{F2.c}) = 1/(1 \times \sqrt{2})$. This similarity metric does not consider null-dimensions as a sign of similarity.

It should be noted that there is a lack of objectivity with this criterion because we measure the quality of the results with methods similar to those we used to obtain the results. To try to minimize the bias, one must be careful to use different descriptive features and similarity metrics for performing clustering and for evaluating cohesion and coupling. Doing this is probably not sufficient though; researchers should look for further independent quality criteria.

---

[9] Precision and Recall are standard metrics in Information Retrieval, see for example [18].

[10] $X$ and $Y$ are vectors describing the two entities $e_X$ and $e_Y$; $\|X\|$ and $\|Y\|$ their Euclidean norms; $X^T$ the transposed vector.

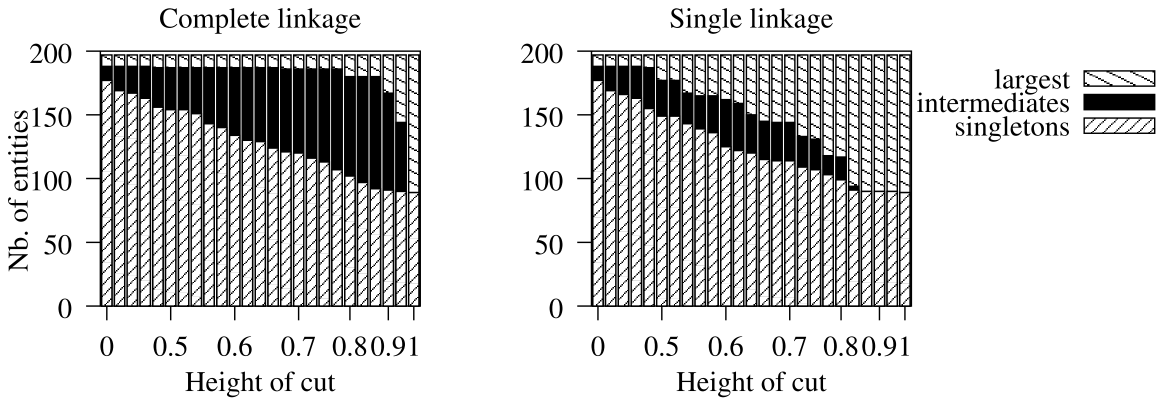


**Figure 5. Size criterion for two algorithms. The algorithm with more intermediate clusters (complete linkage on the left) is better. System: gcc. Feature: File inclusions. Similarity metric: Jaccard coefficient.**

### 3.3 Size Criterion

Modularization of a system implies decomposing it into a set of smaller parts. Although it is conceivable to have clusters notably smaller than others, one wants to avoid extreme scenarios where 90% of the entities are grouped in one large cluster and all the others are in individual clusters. Ideally, all the clusters should be of comparable size.

This issue may be subject to discussion: For example there may be an argument that in a real system, one subsystem could grow out of hand and actually represent 90% of this system. Nevertheless if we recall that clustering algorithms *impose* a structure on a set of entities (as opposed to *discovering* it), we may as well choose the clustering method that imposes the better balanced set of clusters. In fact, it seems reasonable to assume that a software engineer, given the above system example, would naturally decide to repartition the very large subsystem that grew excessively large, in order to get a new set of subsystems of more manageable size.

The size criterion usually agrees with the design criterion about which clustering approaches are best. Still, it proved useful in analyzing the behavior of a clustering method.

We will quantify this criterion with three values: the number of singleton clusters, the number of entities in the largest cluster, and the number of other entities (in the intermediate clusters). We want to avoid solutions having many singleton clusters or only one huge cluster (which would hardly qualify as modularization).

Figure 5 shows an example of a “good” and a “bad” partitioning method. On the left, the largest cluster slowly grows, while the number of singleton clusters decreases steadily and the number of entities in intermediate clusters increases. The method evolves a set of intermediate clusters of comparable size. On the right, the largest cluster grows fast and there are few entities in the intermediate clusters. This method does not decompose the system, but simply gathers one after the other all the entities into the same cluster.

Note that for both algorithms illustrated in figure 5, there are already some intermediate clusters at height zero. This is because a few entities have exactly the same (non-empty) description. Since they are identical, they form clusters at height zero (they have a distance zero between themselves).

Note also that for the left graph, the intermediate clusters completely disappear in the very last stage of the clustering process for the maximum height (here, with Jaccard similarity metric: Value 1). This is normal since at the end of the algorithm all entities must be contained in one cluster. If there are still singleton clusters at this stage, it is because we forced entities with empty descriptions to be infinitely far from all others, extending the maximum height of the cluster hierarchy. In our actual results, there is another step at height “infinity” (not shown here) where we do have only one cluster. Looking at these graphs, one could also consider completely ignoring entities with empty descriptions during the clustering process (as proposed by Mancoridis [14]).

We usually consider that only the highest cuts in the hierarchy of clusters present some interest. We consider that up to half the maximum height, cuts result in partition containing too small clusters and too many singletons to be really useful. The very last cut (maximum height), sometimes the two or three last ones, should also be discarded because the largest cluster then has too much importance. A heuristic to find the maximum *useful* height where to cut could look at the size of the largest cluster. One should cut before it shows a sudden large increase. This occurs towards the end

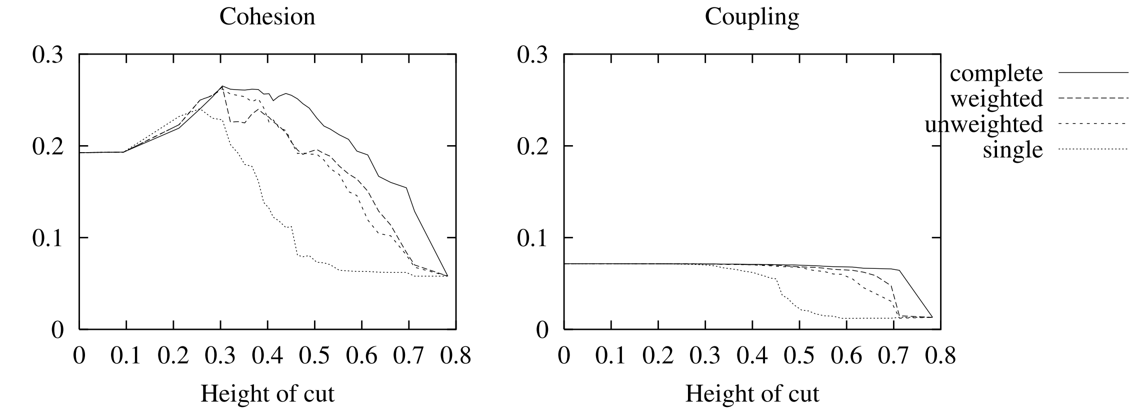


**Figure 6. Comparison of four algorithms using the design criterion. System: Linux. Feature: file inclusions. Similarity metric: correlation.**

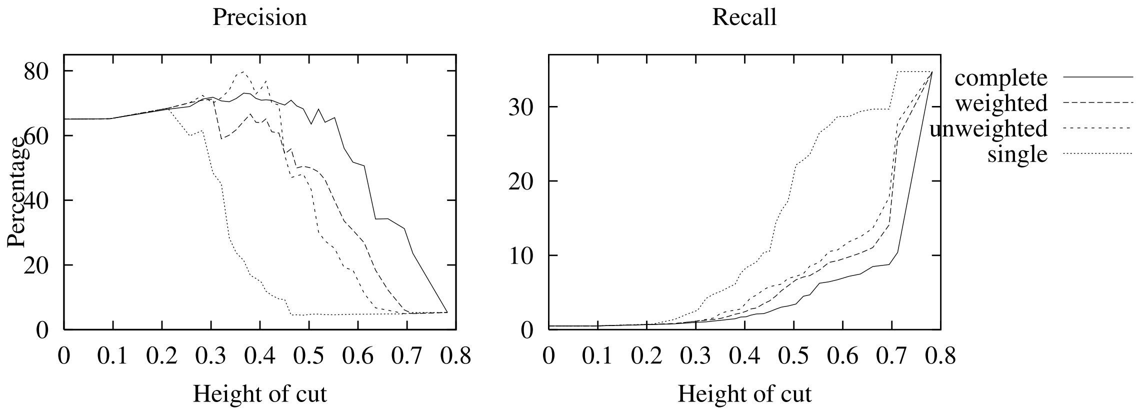


**Figure 7. Comparison of four algorithms using the expert criterion. System: Linux. Feature: file inclusions. Similarity metric: correlation.**

(after height 0.9 in the left graph) of the clustering process.

## 4 Results

We will now present the results of some experiments. Entities are files and the clusters may be thought of as subsystems. We experimented with four systems: Linux, Mosaic [15], gcc [8] and a real world legacy telecommunication system (2 MLOC).

For each system, cohesion and coupling (§3.2) are computed using different features. For Mosaic and gcc the description vectors give the number of calls to each routine in the system; for Linux, it is the references to macros (C language) and for the legacy software it is the references to user defined types. Our goal was to use the feature with the least number of empty descriptions to compute the design criterion.

It is impossible to present here all experimental results. We will concentrate on Linux and Mosaic, the two systems where the comparison with the expert criterion applies. Mosaic is also the system for which we conducted the most extensive set of experiments. The results are consistent for the four systems.

All results are available on the web: `http://www.site.uottawa.ca/~anquetil/Clusters/`.

### 4.1 Clustering Algorithms

The four agglomerative hierarchical clustering algorithms behave as expected. There is a gradation from complete linkage, to weighted, unweighted and single linkage for the three quality criteria (figures 6 and 7). However, unweighted and weighted average linkages are often hard to distinguish.

As expected, complete linkage gives more cohesive clusters and single linkage less coupled ones (figure 6).

Single linkage has a tendency to gather all the entities in one cluster. It creates a big cluster acting like a magnet and

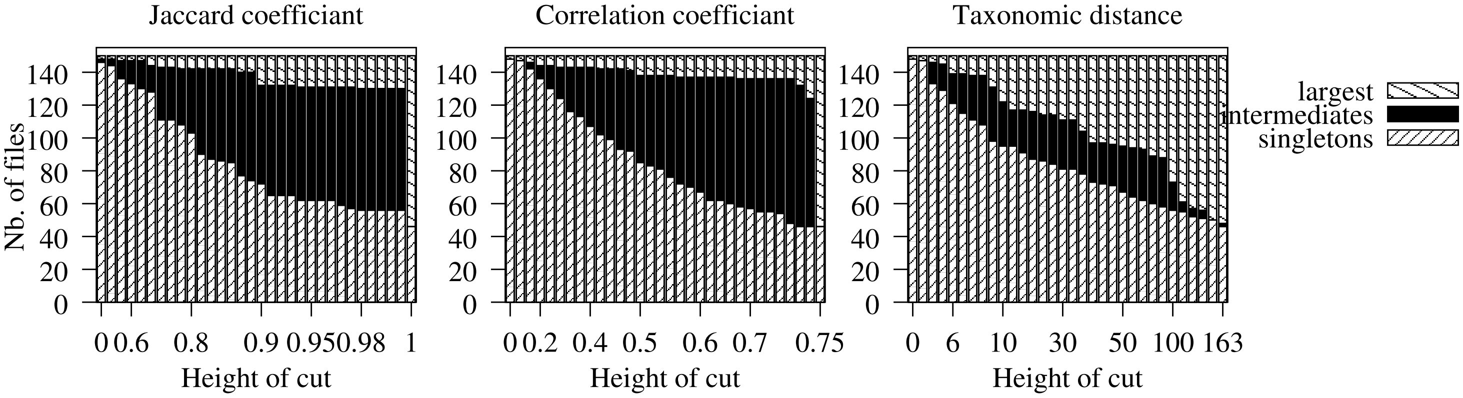


**Figure 8. Comparison of three similarity metrics using the size criterion. System: Mosaic; Feature: type references; Algorithm: Complete linkage.**

pulling to itself all entities around it. The bigger this magnet becomes, the more entities it will engulf (see again figure 5). For this reason we suggest avoiding this algorithm.

In general, the best compromise between cohesion and coupling comes from the unweighted linkage algorithms.

### 4.2 Similarity Metrics

Figure 8 shows the results for three similarity coefficients with the size criterion. The results confirm that it is best not to consider zero-dimensions as a sign of similarity between entities. The Jaccard association coefficient (left graph) and the Sørensen-Dice (not shown) which do not consider zero-dimensions give good results, i.e. many entities in the intermediate clusters. The Simple Matching association coefficient (not shown) and the two distances (Taxonomic distance in the right graph) which consider zero-dimensions tend to cluster all entities into a single rapidly growing cluster and are less satisfactory for this criterion. Finally, the correlation coefficient (middle graph) gives good results although it does consider the zero-dimensions. Because of the complexity of this coefficient (computing correlation then mapping the correlation interval $[-1, 1]$ to a similarity interval $[0, 1]$), it is difficult to explain this good result.

The results from the design and the expert criteria are similar: Jaccard and Sørensen-Dice give the best results, correlation follows closely and the three others (Simple Matching, Taxonomic distance and Camberra distance) give poor results.

### 4.3 Entity Descriptions

The features do not behave in the same way for all systems. As stated earlier, the different features supply different amounts of information. For example, we did not use the type reference feature for gcc because there are few such references. The feature also impacts the number of empty descriptions we have to deal with. For Mosaic, the naming convention feature gives 24 of them and type references gives 46 (see the final number of singleton clusters in figure 8).

The very good result for the routine call feature with the design criterion (figure 9) arises from the fact that design quality is measured using this very feature. This illustrates our assertion that one should always use an independent feature to measure design quality.

The naming convention feature (references to words inside identifiers) produces some of the best results (see for example figures 9 and 10). This supports our theory that one should not focus all the attention on the source code but also consider less formal sources of information. The naming convention feature has two significant advantages: It usually provides more information (few files should contain no identifiers; the 24 we have in mosaic contain only macro definitions which we do not extract) and it is much easier to extract (identifiers are basically all words which are not comments or keywords).

More experiments are in progress to further our knowledge in this area.

## 5 Related Work

There has been a rather limited amount of research that gives the reverse engineering community grounds with which to compare clustering approaches.

Wiggerts gives an overview of clustering techniques in [24]. Our paper is intended as a continuation of his work. Wiggerts give a summary of many classical clustering methods; we take some of these methods and test them to com-

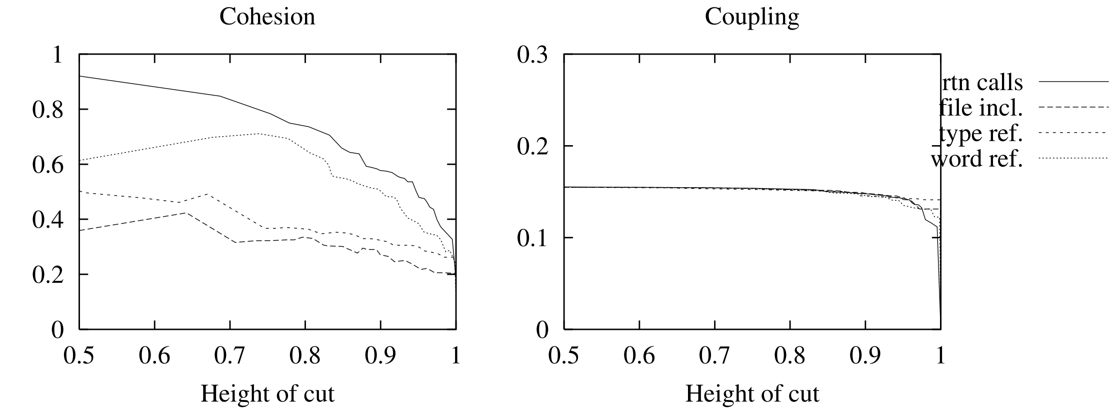


**Figure 9. Comparison of four different features using the design criterion. Earliest cuts (height $< 0.5$) were ignored to improve readability. System: Mosaic. Algorithm: Complete linkage. Similarity metric: Jaccard coefficient.**

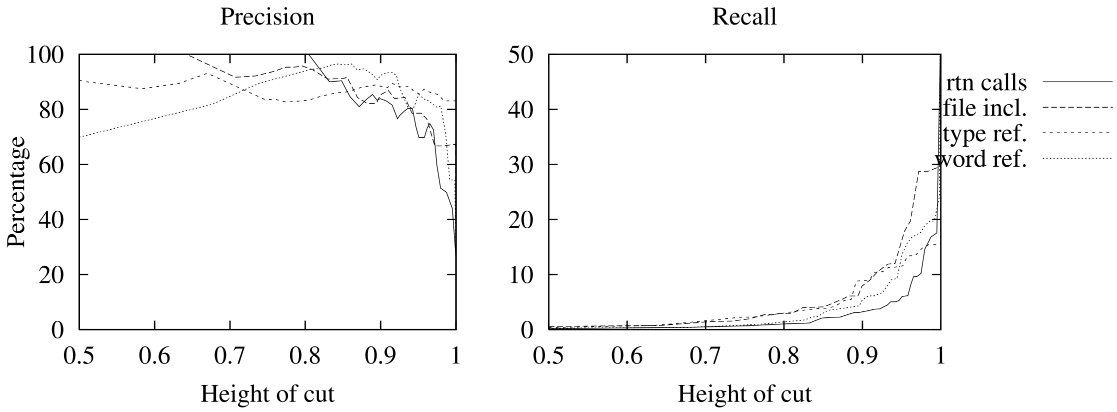


**Figure 10. Comparison of four different features using the expert criterion. Earliest cuts (height $< 0.5$) were ignored to improve readability. System: Mosaic. Algorithm: Complete linkage. Similarity metric: Jaccard coefficient.**

pare their relative strengths and weaknesses for reverse engineering.

Lakhotia [11] proposes a comparison framework for both the description of entities and for clustering methods. He establishes some common ground on which to compare different methods, but does not actually do the comparison.

Armstrong [5] and Storey [22] give a comparison of various reverse engineering tools, however their work is at a higher level of abstraction than ours: They compare tools, discussing such things as the quality of the interface, while we compare clustering algorithms.

Girard et al. [9] compare five different heuristics to find objects and classes in procedural code. Their evaluation is based on a comparison of their results with results proposed by experts. Their work differs from ours in that it does not use general statistical clustering algorithms, but specific algorithms tailored for the task.

Recently, there has been high interest in a new approach called "Concept Analysis" [4, 12, 19]. Again, this is not a statistical clustering method, and falls outside the scope of this paper. Using the informal manual study of their results as a quality criterion, van Deursen and Kuipers [4] conclude that concept analysis is better than hierarchical clustering for grouping variables into "classes". One of their arguments is that clustering algorithms do not allow an entity to appear in more than one cluster. We believe that despite this difficulty, clustering algorithms remain useful, since they extract only *important* concepts from a data set, whereas concept analysis extract *all* concepts and may therefore produce information overload for the user (e.g. see [2]).

Many publications present and evaluate a single software remodularization method. They do provide a comparison of different approaches (hence Lakhotia's work) and also are usually based on more subjective evaluation criteria such as users' appreciation of their results.

Finally, we would like to mention Clayton's plea for a well defined test of what understanding a program is [7]. His work is relevant in that it advocates the specification of a well-defined goal (a test for successful reverse engineering) which would help in comparison of the methods.

## 6 Conclusion

In [24], Wiggerts notes that to make a wise choice in all the available clustering techniques, one should be aware of all the possible choices. However, the descriptions of the different parameters he gives come from the taxonomy literature. Some of their conclusions do not apply to the software remodularization domain.

In this paper, we presented an analysis of three parameters that may influence the clustering results when doing software remodularization with agglomerative hierarchical clustering algorithms: how the entities are *described*, how *distance* between the entities is computed and what *algorithm* is used. The effect of each parameter is evaluated according to three quality criteria: *Design* quality, how well they compare to an *expert partitioning* and *size* of the clusters.

From our experiments, we draw the following conclusions:

1. The descriptive features commonly used are very sparse, they yield many zero-dimensions for each entity (i.e. the description vectors contain many zeros which hinder the comparison). In this context, one should favor similarity metrics which do not consider zero-dimensions as a sign of similarity between entities.

2. The descriptive features may provide very different amounts of information; particularly, different numbers of empty descriptions (entities referring to none of the features chosen for description; these entities are described by a vector contain only zeros).

3. The naming convention feature gives good results. It should receive more attention in reverse engineering. It has notable advantages over what we call source code features (references to user defined types, routine calls, etc.): For example, the naming convention feature is easier to compute and it usually provides more information.

4. There is the same gradation among the four agglomerative hierarchical clustering algorithms as in taxonomy: From complete, to weighted, unweighted and finally single linkage.

5. The complete linkage algorithm gives more cohesive but also more coupled clusters; single linkage less cohesive but less coupled clusters. There is therefore the need to favor one or the other. Since single linkage tends to gather all entities in one large cluster, we propose avoiding it and concentrating on complete linkage. Unweighted linkage is also a good choice, it appears to give the best compromise.

We experimented only with file clustering; further research should consider clustering different entities (routines, classes, processes, etc.) We expect the results to be similar for the same quality criteria. However, clustering has a wide range of application and expectations may vary. Using other entities than files or routines, would probably mean that remodularization is not the end goal. For example in [4], the authors cluster global variables to find possible classes. One of their quality criteria is that the method does not create a partition of the system, but eventually may put the same entity in different clusters. The algorithms tested here do not allow this.

This work raises two issues which deserve more attention:

**Descriptive features:** Our early results with the naming convention feature are very encouraging. We are working on a new set of experiments to concentrate on this issue.

**Quality criteria:** We found the existing criteria weak (objectivity problem) and scarce. The main criterion used in other research seems to be expert opinion which does not promote cross-validation of results between studies.

## Thanks

The authors whish to express their thanks to other members of the Consortium for Software Engineering Research[11] which provided the basic tools that made this research possible: Bell Canada's C/C++ parser "Datrix" (`mailto:datrix@qc.bell.ca`) and the Software Bookshelf base of facts for the linux system (`http://plg.uwaterloo.ca/~holt/guinea_pig/`). We also wish to thank K. Teresa Khidir for editing the paper and greatly contributing to improve the readability as well as the gramar.

## References

[1] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:1–21, 1999. to appear.

[11] see: `http://www.cser.ca/`

[2] N. Anquetil and J. Vaucher. Extracting Hierarchical graphs of concepts from an object set : Comparison of two methods. In *Knowledge Acquisition Workshop, ICCS'94*, 1994.

[3] N. Anquetil and J. Vaucher. Meta-Knowledge for the Object Model : Simple as NOT. In *Metamodelling in OO Workshop, OOPSLA'95*, 1995.

[4] T. K. Arie van Deursen. Identifying object using cluster and concept analysis. In *$21^st$ International Conference on Software Engineering, ICSE'99*, pages 246–55. ACM, ACM press, may 1999.

[5] M. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Working Conference on Reverse Engineering*, pages 30–39. IEEE, IEEE Comp. Soc. Press, oct. 1998.

[6] G. Canfora, A. Cimitile, M. Tortorella, and M. Munro. A Precise Method for Identifying Reusable Abstract Data Types in Code. In *International Conference on Software Management*, pages 404–13. IEEE, IEEE Comp. Soc. Press, 1994.

[7] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *Working Conference on Reverse Engineering*, pages 69–78. IEEE, IEEE Comp. Soc. Press, oct. 1998.

[8] gcc version 2.8.1. `http://www.gnu.ai.mit.edu/software/gcc/gcc.html`.

[9] J.-F. Girard, R. Koschke, and G. Schied. Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding. In *Working Conference on Reverse Engineering*, pages 66–75. IEEE, IEEE Comp. Soc. Press, Oct. 1997.

[10] T. Kunz and J. P. Black. Using Automatic Process Clustering for Design Recovery and Distributed Debugging. *IEEE Transaction on Software Engineering*, 21(6):515–527, Jun 1995.

[11] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *J. of Systems and Software*, 36:211–231, Mar 1997.

[12] C. Lindig and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *19th International Conference on Software Engineering, ICSE'97*, pages 349–59. ACM SIGSoft, ACM Press, May 1997.

[13] S. Mancoridis and R. C. Holt. Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering. In *International Conference on Software Maintenance, ICSM'97*, pages 23–32. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[14] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the $6^{th}$ International Workshop on Program Comprehension*, pages 45–52. IEEE, IEEE Comp. Soc. Press, june 1998.

[15] NCSA Mosaic Version 2.6. Available through anonymous ftp at `ftp.ncsa.uiuc.edu`, in `/Mosaic/Unix/source`.

[16] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse-engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[17] S. Patel, W. Chu, and R. Baxter. A Measure for Composite Module Cohesion. In *$14^{th}$ International Conference on Software Engineering*. ACM SIGSoft/IEEE Comp. Soc. Press, 1992.

[18] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1983.

[19] M. Siff and T. Reps. Identifying modules via concept analysis. In M. J. Harrold and G. Visaggio, editors, *International Concept on Software Maintenance, ICSM'97*, pages 170–79. IEEE, IEEE Comp. Soc. Press, oct. 1997.

[20] P. H. Sneath and R. R. Sokal. *Numerical Taxonomy*. Series of books in biology. W.H. Freeman and Company, San Francisco, 1973.

[21] I. Sommerville. *Software Engineering*. International Computer Science. Addison-Wesley Publishing Comp., 5th edition, 1995.

[22] M.-A. D. Storey, K. Wong, and H. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Working Conference on Reverse Engineering*, pages 12–21. IEEE, IEEE Comp. Soc. Press, oct 1996.

[23] V. Tzerpos and R. C. Holt. The Orphan Adoption Problem in Architecture Maintenance. In C. V. Ira Baxter, Alex Quilici, editor, *Working Conference on Reverse Engineering*, pages 76–82. IEEE, IEEE Comp. Soc. Press, Oct 1997.

[24] T. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *Working Conference on Reverse Engineering*, pages 33–43. IEEE, IEEE Comp. Soc. Press, Oct. 1997.