Experiments with Clustering as a Software Remodularization Method[*]

Nicolas Anquetil 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 get old, reverse engineering becomes more and more important to the companies that have to maintain the code. Clustering is a key activity in reverse engineering to discover a better design of the systems or to extract significant concepts from the code.

Clustering is an old activity, highly sophisticated, offering many methods to answer different needs. Although these methods have been well documented in the past, these discussions may not apply entirely to the reverse engineering domain. In this paper, we study some clustering algorithms and other parameters to establish whether and why they could be used for software remodularization.

We study three aspects of the clustering activity: Abstract descriptions chosen for the entities to cluster, metrics computing coupling between the entities and clustering algorithms. The experiments were conducted on three public domain systems (gcc, Linux and Mosaic) and a real world legacy system (2 million LOC).

Among other things, we confirm the importance of a proper description scheme of the entities being clustered, we list a few good coupling metrics to use and characterize the quality of different clustering algorithms. We also propose novel description schemes not directly based on the source code and we advocate better formal evaluation methods for the clustering results.

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

1 Introduction

As valuable software gets older, the maintainance becomes more and more an issue. Software engineers face systems, were a minor change in some part can have unexpected results in an apparently unrelated part. The reverse engineering community aims at providing solutions to help software engineers understand, restructure or migrate old software towards more modern architecture and/or language. A key method to do so is clustering. It is used to gather software components into modules significant to the software engineers.

But clustering is a research domain in itself and a very sophisticated one. There are many different methods to answer the different needs: Size of the data, type of data, a priori knowledge of the result expected, etc. Reverse engineering is yet a young research domain, with unclear goals [12] and more ad-hoc methods than well establish methodologies. In this context, clustering has often been used with no deep understanding of all the issues involved.

To try to fulfill this need, Wiggerts presented in [37] a summary of the literature on clustering. This work is definitely valuable, it lists all the possible choices and give insights on some advantages and drawbacks associated to these choices. However, the conclusions it draws are those of the literature which may not entirely hold for reverse engineering.

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 the “things” to be clustered are described, the similarity metrics to quantify the coupling between these “things” and a choice of hierarchical clustering algorithms.

The organization of the paper is the following: After discussing the related works, we give a short introduction on clustering and present the three major issues we wish to discuss. Then we analyze each of the three issues in the context of reverse engineering; we continue by discussing some criteria we used to compare the results. Finally we present and discuss experimental results.

2 Related Works

There has been only a little research that gives the reverse engineering community bases by which to compare clustering approaches.

Wiggerts gives an overview of clustering techniques in [37]. Our paper was intended as a continuation of this work. Wiggerts give a summary of many classical clustering methods, we take back many of these methods and test them to compare their relative strengths and weaknesses for reverse engineering.

Lakhotia [20] proposes a comparison framework for entities' descriptions and clustering methods. He intends to establish common ground on which to compare different methods, but does not actually do the comparison.

Armstrong's [6] and Storey [35] have a comparison of various reverse engineering tools, but it is at a higher level of abstraction than us and they consider such things as the quality of the interface. They compare tools, we compare clustering algorithms.

Girard et al. [14] compares five different heuristics to find objects and classes in procedural code. Their evaluation is based on a comparison of the results with experts' propositions. This work differs from ours in that it does not use general statistical clustering algorithms, but specific algorithms tailored for this task.

Recently, there has been high interest in a new approach called "Concept Analysis" [5, 21, 30]. Again this is not a statistical clustering method, and falls outside the scope of this paper. Based on a quality criterion which was informal manual study of the results, van Deursen and Kuipers [5] compared concept analysis to hierarchical clustering and conclude it is better than the latter for clustering variables into "classes". One of their arguments is that clustering algorithms do not allow an entity to appear in more than one clusters. We believe that despite this difficulty, clustering algorithms have their utility, because they extract the important concepts from a data set, whereas concept analysis extract all concepts and may produce many more information than given in input (e.g. see [1]).

Many publications present and evaluate a single software remodularization method. They do not help in comparing different approaches (hence Lakhotia's work) and also are usually based on more subjective evaluations such as the final user appreciation of their results.

Finally, we would like to mention Clayton's plea for a well defined test of what understanding a program is [12]. This work is relevant in that it advocates the specification of a well-defined goal (a test for successful reverse engineering) which would help in comparing work.

3 Issues in Clustering

Reverse engineering tries to help software engineers understand a presumably large piece of software. A key activity in reverse engineering consists of gathering the software entities (modules, routines, etc.) that compose the system into meaningful (highly cohesive) and independent (loosely coupled) groups. This activity is called clustering. There are many different approaches to clustering; we will concentrate in this paper on generic statistical methods. We will not consider more intelligent solutions tailored to solve specific problems (e.g. [9, 36]). Statistical clustering has been thoroughly studied in taxonomy to classify organisms into species. Most of the clustering activity in reverse engineering is based on these studies.

To do clustering we need to consider the three following issues:

Entities' description: Build an abstraction of the real world in which the entities to be clustered are described according to some scheme.

Entities' coupling: Define when we will consider that a pair of entities should be clustered together to make a cohesive unit.

Clustering algorithm Apply a given clustering algorithm to the abstract descriptions of the entities.

The first issue is of the utmost importance. Clearly, if we decide to describe files with the 77th character they contain (files with the same character would be clustered together), the result would not prove very interesting to software engineers. It is our thesis that this issue has been overlooked in the reverse engineering community and should receive more attention. Most of the studies rely on the source code to build the abstract description of the entities (e.g. 10 studies out of the 12 listed by Lakhotia [20]). We advocate more heterogeneous sources of information (for example based on comments). This will be discussed further in section §4.

Once we have found an abstract description for the entities we need to tackle the second issue: when will, two entities, be considered to form a cohesive cluster. There are two approaches which we call direct and sibling link[1]. We can put together entities that depend on one another, this is the *direct link* approach. Or we can put together entities that have the "same behavior", this is the *sibling link* approach. In figure 1, the file F1.c is clustered with Fa.h because it includes it and presumably needs it to achieve its goal (direct link), and files F2.c and F3.c are clustered together because both include the same header files and therefore have the same needs and presumably similar goals (sibling link).

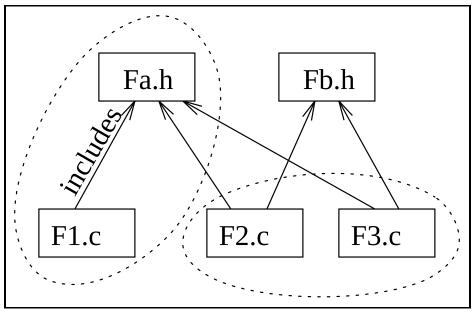

Figure 1: The two reasons for clustering entities (here files) together: one file depends on the other (direct link, left cluster); or two files have the "same behavior" (sibling link, right cluster.)

The difficulty is that typical software systems are fully connected graphs where any entity is connected to all the others by some path. To create "meaningful independent clusters", we must decide to ignore (or break) some links. For example, in the figure, the fact that F2.c includes Fa.h is not considered meaningful enough to cluster them together. For software engineers that can rely on their knowledge of each entity, the task can already be difficult, there are many valid ways of decomposing a software system (functional decomposition, data-oriented, event-oriented, horizontal, vertical, etc.) For a program that relies only on abstract information, with no semantic information about the actual importance of each link, the task is even more complex. For both above approaches, the coupling between two entities is quantified by some metric (we call them "similarity metrics", see §5.1) on which the clustering algorithms base their decision whether or not to break the links.

This brings us to the last issue: Clustering algorithms. It is important to understand that these algorithms do not *discover* some hidden or unknown structure in a system, but rather *impose* a structure on the set of entities they are given. They (arbitrarily) decide to ignore some links and favor others. This decision is based on the similarity metrics used and on the algorithm itself (e.g. genetic algorithm or hierarchical algorithm). The structures imposed by the different algorithms have different qualities. Some are definitely useless because they do not correspond to a reasonable software engineer's view of a system. Other structures may be interesting in different ways. Some clustering algorithms may always result in useful structures, or they may give interesting structures for some systems, and useless ones for other systems, etc. We will discuss these points in subsection §5.2.

Clustering is a very broad domain, and it is not possible to cover it all in a single article. In this study, we experimented with a large set of possible entities' descriptions, ranging from "traditional" ones (e.g. based on data binding) to more innovative ones (e.g. based on comments). An important conclusion will be that informal sources of information (e.g. similarities of comments) are interesting and can give as good results as the more formal ones. For coupling, we conducted a few experiments to compare the direct and the sibling link approaches. Our conclusion will be that both give very similar results and should be compared on other issues (e.g. ease of use). We studied more extensively the sibling link approach and the similarity metrics pertaining to it. Finally, on the third issue, we did a few experiments with a hill-climbing algorithm[2] and hierarchical clustering algorithms.

An issue that we will hardly discuss in this paper but which we feel is important too, is the problem of formal evaluation of the results of clustering. This is important because it is needed to compare the results of the research. We felt that there was a need in this domain (see also the section on related works).

4 Entities to Cluster and their Descriptions

In taxonomy, entities are organisms (animals or plants) that should be classified into species. They are describe using *features* like "number of leaves",

[1]Patel in [28] uses a cocktail-party analogy to describe these two approaches.

[2]Thanks to S. Mancoridis at Drexel University for providing the Bunch tool, [7, 23]

"laying eggs" or not, etc. The features may come from many different sources, based on anatomy, geographical distribution, temporal distribution, behavior, chemical properties, etc. For each entity, one first lists all the features it possesses. Clustering is used to group together those entities that have the same features; an approach that we describe as the sibling link (§3). We saw that for software remodularization there is another possible approach (direct link) where direct dependencies between entities (files including other files, processes calling other processes) is used to do the clustering. This will be studied in §4.3.

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 program components. For example, the five files of figure 1 could be described by the other files they include:

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

Other feature like what external routine is called from each files may give a weight to each descriptive element (we will call them dimensions). In that case, missing dimension will receive a weight of zero. The above descriptions 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)

Such descriptive features are based on the code, we will call them formal features. Formal descriptive features are predominant in software remodularization, we will discuss the availability and interest of non-formal descriptive features (§4.2).

4.1 Formal Descriptive Features

We will say that a descriptive feature is *formal* if it consists of information that has direct impact on, or is a direct consequence of, the software system's behavior. For example, describing an entity with the routines it refers to is a formal descriptive feature because it is an information source that has direct impact on the system's behavior. If we change a routine call in the code, we should expect a change in the behavior.

Formal features, based on the code, are an obvious choice when one wishes to do software remodularization because they represent the system's actual state. They are the most commonly used in research; in [20], Lakhotia lists 12 works on clustering among which only two use non-formal features (one exclusively [22][3] and the other a mix of formal and non-formal [26]). We experimented with the following formal descriptive features:

Type: User defined types referred to by the entity described. For example a type is referred to when it is used to define a variable. This feature has been used in [19, 28] to describe files or in [10] to describe routines.

Var.: Global variables referred to by the entity. A variable is referred to when it is assigned a value or when its value is read.

Rout.: Routines called by the entity. Used in [26, 36] to describe files or in [2] to describe classes.

File: Files included by the entity. Only used to describe files as in [24, 36].

Macro: Macros used by the entity. This feature is only available in some languages (like C).

All: Union of the five preceding.

There could be other formal features, like inter processes calls [18] or past bug fixes that impacted the entities.

The choice of descriptive features seems more limited than for taxonomy; most of the literature reports experiments with formal features, extracted from the same source: code. This lack of choice is aggravated by the redundancy between the different features[4]. For example to reference an externally defined global variable ("var." feature, above), one needs to include some file that describes it ("file" feature); if a routine refers to a type as one of its parameters' types ("type" feature), then a calling routine ("rout." feature) should also refer to this type as one of its variables' types. The "var." feature has also redundancy with the "type" feature, since reference to a global variable will usually imply that there is also a reference to the type of this global variable (for example to define a temporary variable of the same type). These redundancies are often imposed by the programming languages to provide for error checking.

[3]is work is not really reverse engineering, but more akin the Information Retrieval and document classification.

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

Another problem with formal features is that they are at a very low abstraction level (call of utility routine, uses of loop counter variables, etc.) and contain much noise. This makes it difficult to extract significant abstract concepts.

4.2 Non-Formal Descriptive Features

To overcome the problems associated with the formal descriptive features, we propose to use non-formal ones. We propose to call descriptive features, *non-formal*, if they use information having no direct influence on the system's behavior. A typical example would be naming of routines, since changing the name of a routine has no impact on the system's behavior.

In a previous study [3], we showed that for the legacy software system we studied, file naming convention was the best descriptive feature to recover the decomposition of small subsystem examples given to us by software engineers. Other non-formal descriptive features could include: Comments describing the entities; names of software engineers who worked on an entity; dates when an entity was created or modified, etc. We experimented with the two following ones:

Ident.: References to words in identifiers declared or used in the entity. Identifiers found in the entity are automatically decomposed into words according to simple word markers (see appendix for a list of the heuristics we used). This is a naming convention descriptive feature similar to what is used in [4].

Cmt: References to words in comments describing the entity. For better results, the words are filtered using a standard stop list[5] and stemmed[6].

Traditionally, for legacy software, documentation in all its forms is considered either missing or outdated. Some researchers argue that non-formal descriptive features are not reliable because nothing guarantees that the information extracted will correspond to the actual behavior of the system. Sneed, for example, states that "in many legacy systems, procedures and data are named arbitrarily. Programmers often choose to name procedures after their girlfriends or favorite sportsmen" [33]. Clearly in such extreme cases, non-formal features should prove useless, however many legacy systems use significant identifiers, intended to help the software engineer grasp the semantics of the software components they denote. Other researchers have used heuristics based on naming conventions for various reverse engineering activities (e.g. [8, 11, 27]).

Assuming they provide relevant information, non-formal descriptive features can offer some or all of the following advantages over formal ones:

- Less redundancy.
- More abstract (describe high-level intent of the code).
- Closer to human understanding.
- Easier to extract.
- More versatile.
- More information (less entities with empty descriptions).

One of the faults of the formal features is that there is redundancy among them. This is inherent to modern programming languages. The hope is that non-formal features, because they are not so tightly restricted by programming languages, should not exhibit this redundancy. This would provide for a different point-of-view on the system. The more independent features we will find, the more views we will have on the system. However, we will see that this independence is not granted, for example "all" (formal feature) is based on all references to identifiers (routine, variables, types or macros) and files (inclusions) in an entity. It would be very close to a non-formal feature based on all identifier (uses and definitions) found in the entity. And this one in turn would probably be close to "ident." which is based on all the words composing the identifiers found in the entities. We designed an elementary experiment to test this issue. It will be discussed in §6.

Non-formal features such as the one we experimented with, are intended for human readers. As such they provide information at a higher level of abstraction than the code. This should make it easier to extract abstract concepts. Again, we will discuss in §6 an elementary test to evaluate the level of abstraction of extracted concepts.

Also because these features were intended for human readers, the result should be readily understandable to the software engineers in terms that are common to them (application domain concepts). We did not test this point. It seems only verifiable by using human experts.

[5]We used the stop list from an Information Retrieval system: Smart [31].

[6]Word stemming consists of extracting the "root" of a word, for example by removing the "s" at the end of plural nouns. We used the stemming functions provided by WordNet [38].

The non-formal descriptive feature we chose are also much easier to extract. For example, extracting words from comments is a simple task once you know the comment delimiters. The formal features suppose that one has a complete grammar for the language, or tracing capabilities (description based on process calls).

Non-formal features are more versatile, they should be programming language independent and could, therefore, be applied to other sources than code.

Finally non-formal features should provide more information than formal ones. We refer here to the quantity of information provided, the quality issue as been discussed above. Because they are more versatile, non-formal features are almost always present in all entities. On the contrary it is not rare that an entity has no reference for a given formal feature. For example, a system having well encapsulated data would have very few references to global variables. We noticed that, for many formal features, the percentage of entities having an empty description is high (more than 10% and up to 65%). The lack of enough information for a feature restricts further an already limited choice of features. On the other hand, we always found our non-formal features to have below 5% of empty descriptions.

We do not pretend however that any non-formal features is useful. It is the one of the purposes of this paper to establish whether they are and which ones are.

4.3 Direct or Sibling Links

The direct and sibling links, presented in §3 are two different ways for computing the coupling between the entities. Although this issue applies mainly to the discussion on similarity metrics, it has some bearing on the entities' descriptions.

The direct link approach has an appealing simplicity, if a routine calls another routine, the two are coupled to some degree. But it is limited to descriptive features recording links between the entities to cluster themselves (files including other files, or structured types built from other types). This appears to rule out things like non-formal features. The sibling link approach, on the contrary, put less constraint on the form of the descriptions, all it requires is to be able to compute a similarity between them.

We experimented mostly with sibling link. All the descriptive features presented in §4.1 have been used as sibling links. In addition to this, we also did a few experiments using all the formal descriptive features for direct link. It was hard to know what to expect from the result of both approaches since we can not use the taxonomic domain as a reference. We found only one work comparing the two approaches: In [17], Kunz reports that he found direct links to be inferior to sibling link. This matches our experience, we often found sibling links to be slightly better.

It should be noted that, the redundancy we already pointed out between formal descriptive features, may sometimes establish interdependencies between the direct and sibling links. Consider again the example of a calling routine referring to a type for one of its variables because the called routine refers to the same type for one of its parameters. The routine call is a direct link, the fact that both routines refer to the same type is a sibling link. Other examples could be given: a file including another file (direct link) because one defines a global variable (or type or routine) that the other uses (sibling link).

4.4 Data and Functional Bindings

Some authors distinguish between descriptive features using data or functional bindings [24]. We propose to classify "var." and "type" as data bindings features, and "rout." as functional bindings. The "macro" feature is unclear since these could be constants (macros without "parameter") and hence more akin to data bindings, or else, full fledged macros (with "parameters") and here more akin to functional bindings. Similarly "file" inclusions may be needed to get access to data structures or externally defined functions.

The importance of each one depends on the specific type of software one is working on. For example, scientific applications, with an emphasis on computation and algorithms, would seem better candidates to be clustered using functional binding features. Older software (poor data encapsulation) and multi-processes applications using shared memory for communication would appear better fitted for data binding features. However, in the general case, the choice would depend more on the modeling principles followed by the designers of the system than on peculiarities of the domain. As a simple heuristic to help decide what kind of binding would be better suited for a system, we simply propose to look how many entities have some references with the feature chosen. If more than 90% of the entities have some references, the feature seems an acceptable choice.

Again, the redundancy between formal features has some repercussions here. The already presented example of routine calls having interdependencies with type references, links functional binding (routine call)

Table 1: Summary of the characteristics of different descriptive features. "Both" means suited for the direct and sibling link approach.

	Formal	Binding	Link
var.	yes	data	both
type	yes	data	both
rout.	yes	functional	both
macro	yes	-	both
file	yes	-	both
all	yes	-	both
ident.	no	-	sibling
cmt	no	-	sibling

to data binding (type reference).

Table 1 gives a summary of the different characteristics each of the eight descriptive features we tested may have.

5 Other Issues in Clustering

5.1 Similarity Metrics

Similarity metrics compute a coupling value between two entities. This is an important issue, according to Jackson et al. [16] the choice of a proper similarity metric has more influence on the result than the clustering algorithm.

These metrics may use the direct link or the sibling link approach (described above). In the direct link approach, the coupling between two entities will be stronger if they have more links between themselves. For finer tuning, links can be weighted (if an entity calls a routine 17 times, this is a more significant link than calling it only once). We did a few experiments with the direct link approach using an independent tool (Bunch [7, 23]).

We concentrated our efforts on metrics for the sibling link approach. They consider two entities' descriptions and try to compute the similarity between the entities. The more similar two entities, the higher the coupling between them. There is a large number of similarity metrics, but they may be grouped in the following four broad categories (from [32]): 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 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[7]. 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 tend to 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 [32], they are close to the correlation coefficients.

We experimented with two distance coefficients, three association coefficients and one correlation coefficient, all proposed in [32]:

Correlation: (correlation[7])
$sim(X,Y) = 1 - \sqrt{(1-r)/2}$

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

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

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

Jaccard: (association[8])
$sim(X, Y) = a/(a + b + c)$.

Simple Matching: (association[8])
$sim(X, Y) = (a + d)/(a + b + c + d)$.

Sørensen-Dice: (association[8])
$sim(X, Y) = 2a/(2a + b + c)$.

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 for software remodularization. A feature like references to routines may leads tens of thousands of dimensions for 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 very sparse, there are 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.

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, all entities with an empty description raise a problem. How can we cluster them if they have no connection to any other entity? We arbitrarily chose to treat the empty descriptions separately in all the similarity metrics and consider that they are infinitely dissimilar to any other. This yields a number of singleton clusters (the entities with empty descriptions) left after the clustering process. Bunch, used to experiment with the direct link approach, makes the same choice.

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: Because they have very similar descriptions, they are considered to be strongly coupled. These metrics make sense in taxonomy, but in software engineering, with the descriptive features used, they will be misled by the general high proportion of zero-dimensions. This will be the case for the Simple Matching association coefficient and the two distance coefficients. These quasi-empty descriptions can form a core cluster which will attract to it all other entities with only a few more non-zero-dimensions. This can ultimately lead the clustering method to gather all the entities into one single cluster, rendering it useless.

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

5.2 Clustering Algorithms

The next choice is that of a clustering algorithm. We will concentrate on agglomerative hierarchical algorithms. For software remodularization, non hierarchical algorithms have also been used (see [20]). Again we did a few experiment with non hierarchical algorithms using Bunch (see also previous subsection). It uses a hill climbing algorithm, trying to minimize an objective function which subtracts the average "inter-connectivity" (Bunch's measure of coupling) of the partition to the average "intra-connectivity" (Bunch's measure of cohesion). For a more detailed description, see [7, 23].

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 containing everything. This results in a binary tree of clusters. One advantage of these algorithms is that they are unsupervised, they don't need any extra information such as the number of cluster expected and possible region of the search space where 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.

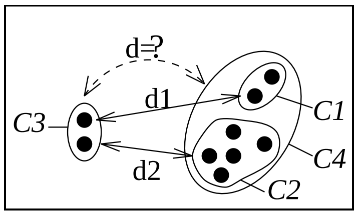

Figure 2: Distance of a new cluster (*C4*) to other clusters (*C3*) in agglomerative hierarchical clustering algorithms.

These algorithms are differentiated by the way they compute the distance of a new cluster to all other ones. 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 different algorithms:

Single linkage[9]: (or closest neighbor rule) the new distance will be $d = min(d_1, d_2)$.

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

[9]Do no confound the *Simple* Matching coefficient (§5.1) with the *single* linkage algorithm.

Table 2: Summary of the three clustering parameters we studied.

Descriptive Feature		Similarity Metric		Algorithm
	var.			
	type		Jaccard	
formal	macro	*association*	Simple Matching	complete linkage
	rout.		Sørensen-Dice	unweighted linkage
	file	*correlation*	correlation	weighted linkage
	all	*distance*	Taxonomic	simple linkage
non-formal	ident.		Camberra	
	cmt			
formal features		Bunch (direct link)		Bunch (hill climbing)

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

Unweighted average linkage: $d = (\|C_1\|d_1 + \|C_2\|d_2)/(\|C_1\| + \|C_2\|)$ 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).

These algorithms have an influence on 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 more cohesive ones (cohesion and coupling are two measures of the design quality of a partition). Unweighted and weighted average linkages stand "in between", on the scale: complete, weighted, unweighted and single linkages.

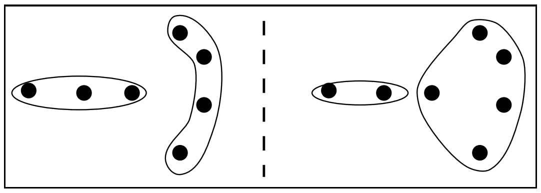

Figure 3: Influence of the clustering algorithm on the clusters (from [37]). Left: Single linkage. Right: Complete linkage.

There is no a-priori reason to favor cohesion or coupling, but because the descriptive features (on which cohesion and coupling are based, see §6.3) 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 which aspect one wishes to favor.

This was the last of the three parameters we studied. We give in table 2 a summary of these three parameters.

5.3 System Partitions

For software remodularization, we will often want a partition of the system instead of a hierarchy of clusters. Bunch does provide us with such a partition, but the hierarchical algorithms described in the previous subsection generate a hierarchy of clusters. In this case, the partition can be obtained by pruning the hierarchy at an 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. This maximum height depends on various aspects of the clustering method (similarity metric, algorithm, etc.)

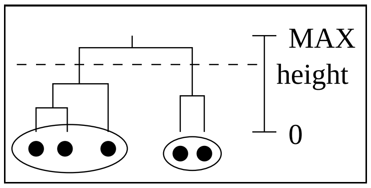

Figure 4: A hierarchy of clusters and how to cut it to get a partition of the data set.

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

6 Quality Criteria

Quality criteria are central to any formal comparison as well as the precise definition of a scientific research domain. Clustering is often use to remodularize an old software system to help the software maintainers understand and manage this system. 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 novice understand the system, help expert assert the consequences of a modification in the code).
- Be general (adaptable to different software, programming languages, etc.)

We found the following criteria in the reverse engineering literature:

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:

Redundancy: A simple test to assess the level of redundancy between two descriptive features.

Size: Whether the clusters are of a similar size or not (e.g. one large cluster and many small ones is not good).

6.1 Redundancy Among Features

We said that formal descriptive features have redundancy among themselves.This would mean that different features tend to give similar results. We propose to compute for each entity the other entity which is most similar to it for any given feature. We then count how many closest pairs two features have in common. Features with a lot of redundancy should have many common closest pairs.

This is not a perfect test:

- There can be several closest entities to any given entity.
- The test results depend on the similarity metric used.

It sometimes happens that one entity does not have one closest other entity, but a group of them all equally close. In this case, our test choose one of them to be the closest. If the test does not choose the same closest for two different features this reduces the level of redundancy measured. A related problem is that sometimes there is one closest entity and a second one, barely further for one feature and the order is reversed for the other feature. Again, this will not be considered a common pair. To overcome these problems we could have considered a group of closest entities instead of only one, and give a threshold within which different distances would be considered equal, but this would have made the test for the number of common pairs more complex. We will just accept the possibility of errors in our results.

The second difficulty with our test is that it is dependent on the similarity metric used and can give different results for different metrics. Problems can occur that are similar to the one described in the last paragraph: For one metric, there might be a closest entity and again a second one a bit further a part; the order might be reversed for a second metric. Since no metric is, a priori, better than the others, we chose to test several of them (presented in §5.1) and make a summary of all results.

6.2 Expert Criterion

This quality criterion is usually assessed by a real expert giving his appreciation of the results. Due to the very large number of experiments we did (thousands), it was not possible to proceed that way. We used an automated comparison with an "expert decomposition". 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 two factors contribute to their significance:

- The directories themselves help to keep the initial design by giving a framework to the maintainer.
- The high granularity level used in our experiments (files) is less sensible to design drift through maintenance.

Although, the clustering algorithm and the directories both potentially provide a tree structure, 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 files and not only subdirectories) as forming subsystems. This should match the abstraction level of the clusters extracted which is usually low.

The difficulty of this quality criterion is to compare two sets of clusters, whereas the clusters are only defined by their members. If one cluster has one entity more than another, how do we compare them? 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 to do it by considering pairs of entities. Two entities are either in the same cluster (we call it an "intra" pair) or in two different clusters (an "inter" pair). We can then compute precision and recall[10] for a given partition:

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 would have 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 would have 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 more chances that these are correct (good precision), but there will be many intra pairs from the expert decomposition that will not be discovered (bad recall). One could say that the extracted clusters are at a lower abstraction level than the expert partition (see also 6.5).

One problem with this automated comparison is that it does not have the flexibility of the human evaluation. An expert judging a partitioning will try to establish if it is reasonable. Our criterion accepts only one good partition. On the other hand, the human evaluation is highly subjective and not easily quantified. These problems could be acceptable if the same set of experts evaluated all the results, but they are definitely a major drawback if one thinks about comparing different works in the community.

[10]Precision and Recall are standard metrics in Information Retrieval, see for example [29].

6.3 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. [34]) which we will now briefly describe. Cohesion and coupling are based on pair-wise coupling between the entity. This in turn is computed with a scheme very similar to what we used to do the clustering. Entities are described using one of the features already discussed (usually formal features: §4.1), and the coupling between entities is evaluated by some kind of similarity metrics. We used a different similarity metric than the ones described in §5.1. This one is proposed in [19] (see also [28]) for cohesion and coupling:

Similarity between entities[11] 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).

This criterion is not entirely objective when applied to automatically generated clusters: Clustering algorithms partition the set of entity, trying to optimize intra cluster similarity and to minimize inter cluster similarity. This is based on a similarity metric between entities' abstract descriptions. The design criterion measures cohesion and coupling between the clusters, also based on a similarity metric between entities' abstract descriptions. In other words we measure the quality of the result with the same method we used to get the result. Even if the similarity metrics used in both case are different, and if we are careful to use two different descriptive features, some doubts remain on the validity of this criterion. Remember for example the interdependence between some formal features we denounced in §4.1.

6.4 Size Criterion

In [15], Hutchens draw an analogy between a partition of a system and star systems. He defines three types of system:

[11]X and Y are description vectors for the entities e_X and e_Y; $\|X\|$ and $\|Y\|$ their Euclidean norms; X^T the transposed vector.

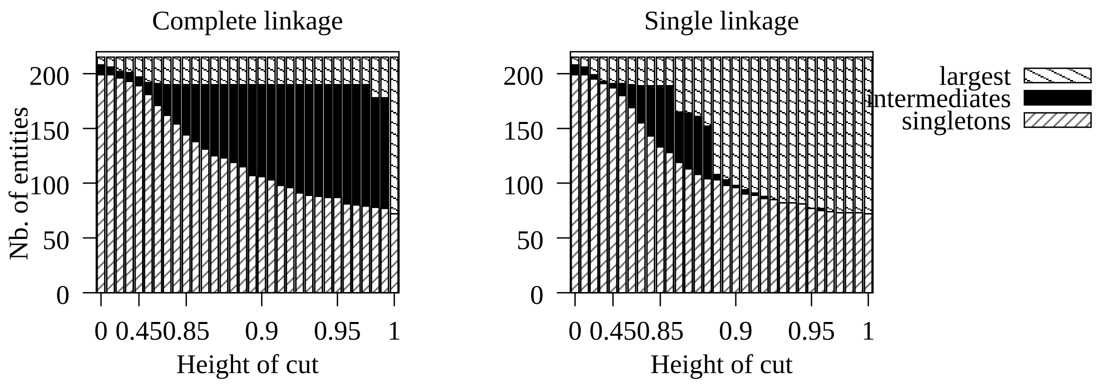

Figure 5: Size criterion for two algorithms. The algorithm with more entities in intermediate clusters (complete linkage on the left) is better. System: gcc. Feature: "rout.". Similarity metric: Jaccard coefficient.

- The *planetary system*, where several subsystems (planets) are interconnected to form the system. There may be a larger subsystem (the sun) acting as the core of the system. This is the ideal case.
- The *black hole system* has no visible planets, one key subsystem absorbs everything revolving around it.
- The *gas cloud system*, has no cluster. All entities tend to stand alone.

Each particular configuration may be an inherent property of the software system —poorly designed systems may naturally lead to black hole or gas cloud configurations— but it may also depend on the specific similarity metric or clustering algorithm chosen.

We will quantify this criterion with three values: Number of singleton clusters, number of entities in the largest cluster and number of other entities (in the intermediate clusters). We want to avoid solutions having many singleton clusters (gas cloud) or only one huge cluster (black hole). Since the hierarchical algorithms we use produce a tree of cluster with leafs being singleton and the root containing all entities, we need to precise what we consider a gas cloud or black hole configurations. We will call a *black hole* configuration a situation where the algorithm tend to create one big cluster that grows regularly along the clustering process and drag one after the other all entities to it. We will call a *gas cloud* configuration a situation where the algorithm tend to create very small clusters and then "suddenly" cluster all of them into one cluster at the higher height for the given similarity metric. Figure 5 gives examples of a good and a bad partitioning method. On the left, we can see the number of singleton clusters decreasing steadily and many entities in intermediate clusters. This corresponds to a planetary system configuration. On the right, the largest cluster grows fast and there are few entities in the intermediate clusters. This is a black hole configuration.

In our experiments, the gas cloud and black hole configurations arise from different factors. The gas cloud were always a consequence of the special treatment we imposed on entities with empty descriptions (§5.1). This is illustrated by the high number of singleton clusters remaining at the end of the clustering process in both graphs. The gas cloud arises as a sign of a bad descriptive feature (many empty descriptions), it is simple to detect it beforehand (see §4). The black hole configuration, on the other hand, was usually a consequence of a ill-adapted algorithm or similarity metric. The two can also combine as in the right graph.

Note that for both algorithms, 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 in one cluster. If there are still singleton clusters at this stage this is only because we forced entities with empty descriptions to be infinitely far from all others. This arbitrarily extend the maximum height of the cluster hierarchy. In the 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 that entities with empty descriptions are completely ignored during the clustering process (as proposed by Mancoridis [23]).

6.5 Clusters' Level of Abstraction

Another quality, the abstraction level of the cluster, could presumably be measured using the size of the clusters. More abstract clusters should cover more cases which means contain more entities. Thus the average abstraction level of a partition could be evaluated as the average size of its clusters.

However, with the hierarchical algorithms we used, this information has no practical meaning since the algorithms build a hierarchy of clusters going from all entities being singleton clusters at the leafs to all entities being in one cluster at the root. Therefore if we cut the hierarchy at heigh 0, the average size of the clusters will be 1 and if we cut at the maximum height, the average size will be the number of entity in the system.

This quality criterion could only be used for algorithms extracting one single partition of the system. For this reason, we did not test it.

7 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 [25], gcc [13] and a real world legacy telecommunication system. You can find some informations on the four system in table 3.

Table 3: Some numbers on the four systems we experimented with.

	gcc	Mosaic	Linux	telecom.
Language	C	C	C	Pascal
# LOC	460K	140K	600K	2M
# files	215	225	875	1817
# routines	4281	3222	7893	15354
# glob. var.	2311	691	2694	25420
# types	531	1251	1551	8742
# macros	3657	7037	18533	-

Cohesion and coupling (§6.3) are based on the "all" descriptive feature (see §4.1), i.e. the union of: references to all variables, types and macros, routines called and files included. The rational for this choice is that we wanted a formal feature without giving any advantage to the basic formal features. Note that the legacy telecommunication system being written in Pascal, the "macro" feature is not available for it.

We also did a few experiments with Bunch, a clustering tool using direct link between the entities. Bunch offers two different algorithms: hill-climbing and genetic algorithm, both having an optimal and sub-optimal version. For efficiency reasons we mainly experimented with the sub-optimal hill-climbing algorithm. This algorithm finds a local optima for its own version of the design criterion. We experimented Bunch on Mosaic with all formal features. We also did one experiment with the optimal hill-climbing algorithm, using the "all" descriptive feature. The results according to our quality criteria were not significantly different from the sub-optimal results for the same feature (and often worst).

It is impossible to present here all experimental results. We will concentrate on Linux and Mosaic, the two systems to which the expert criterion applies. The results are consistent for the four systems.

7.1 Redundancy Among Descriptive Features

The main results of our redundancy test for the descriptive features are:

- The formal features have more redundancy between themselves than with the two non-formal ones.
- The two non-formal features have more redundancy between themselves than with the formal ones.
- "All" and "ident." are an exception, there is usually more redundancy between them than between "all" and the other formal features (but still less than between "ident." and "cmt", the two non-formal features).
- The similarity metric used has some influence on the results.

In table 4 we propose the example of the Linux system; closest pairs of entities were computed with the Jaccard association coefficient.

As hypothesized, the test shows more redundancy between the formal features as between formal and non-formal features. Their is an exception, however, "all" has higher redundancy with "ident." than with the other formal features. This can be explained by the fact that "all" is the union of all references to all identifiers (variables, routines, types and macros) and "ident." is extracted from all identifiers (references and definitions).

Table 4: Redundancy among descriptive features for Linux with the Jaccard association coefficient. Percentage of closest pairs between two features.

	var.	macro	rout.	file	type	all	cmt	ident.	Average
var.		35.6	47.1	29.8	25.8	21.6	10.1	7.8	25.4
macro	35.6		42.1	30.7	27.8	48.6	14.8	26.4	32.3
rout.	47.1	42.1		34.6	29.9	29.9	14.5	16.8	30.7
file	29.8	30.7	34.6		25.1	35.1	16.7	15.4	26.8
type	25.8	27.8	29.9	25.1		37.3	13.3	18.3	25.4
all	21.6	48.6	29.9	35.1	37.3		20.1	37.7	31.5
cmt	10.1	14.8	14.5	16.7	13.3	20.1		25.0	16.4
ident.	7.8	26.4	16.8	15.4	18.3	37.7	25.0		21.1

"Cmt" has very few redundancy with the formal features. It has, however, a relatively high redundancy with the other non-formal feature: "Ident." We have no real explanation for this, other than the fact that they are both intended for human. Although we were expecting low redundancy between non formal features, this could be in fact a good news in the sense that it demonstrates some reliability among these features and validate this classification.

In addition, "var." seems to have little redundancy with "type" whereas we considered both were data binding features. This result is confirmed by the features' results comparison (§7.2). May be this classification needs to be reconsidered.

The test also shows a high redundancy of "macro" with "all" and to a lesser extent "ident." We explain this by the large number of macros the three C systems contain. "Macro" seems statistically more represented in "all" than other formal features. This would also explain the redundancy with "ident." (by a kind of transitivity).

7.2 Entity Descriptions

We already pointed out that the descriptive feature has few influence on the planetary system configuration of the clusters. It only influences the number of singleton clusters which are a consequence of entities with empty descriptions. The size criterion does not provide much information here and we only give the percentage of entities with empty descriptions each feature yield (see table 5).

The results proposed in example in figure 6 are for the Mosaic system. For lack of space we had to put a lot of information in the graphs. The two points on which we whish to insist are: Comparison between the non-formal (the two curves with "x"s) and formal features (simple curves); comparison between the direct link approach (Bunch's results represented by black dots) and the sibling link approach (hierarchical clustering results represented by the curves). For the left graph, the ideal point is the lower right one (high cohesion, low coupling). Curves evolve from the top of the graph (lower cuts, bottom of the hierarchy) towards the bottom (higher cuts, top of the hierarchy). For the right graph, the ideal point is the upper right corner (high cohesion and coupling), curves evolve towards the top. We usually consider that only the last cuts represent interesting partitions (better planetary system for example).

Table 5: Percentage of entities with empty-descriptions for the different features.

	gcc	Mosaic	Linux	telecom.
type	23	24	21	19
var.	44	65	53	28
rout.	33	29	42	38
file	18	12	24	21
macro	22	12	27	-
all	8	6	10	12
ident.	4	0	0	15
cmt	3	1	6	-

In summary, we noticed the following points:

- "Ident." and "all" give good design results.
- "File" has good expert comparison.
- "Cmt" has good expert comparison, but bad design results.
- "Type" (data binding) has results significantly different from "var."
- "Var." give poor results for Mosaic and Linux, average for gcc and good for the telecommunication system.

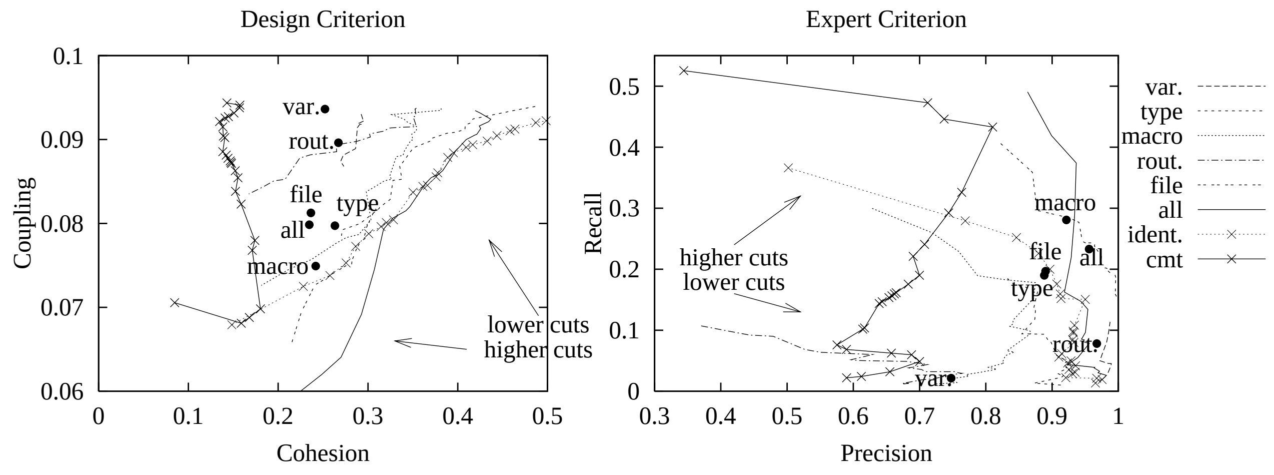

Figure 6: Comparison of descriptive features, design and expert criteria. To improve readability, the five first and the last cuts were ignored. System: Mosaic. Similarity metric: Jaccard coefficient. Algorithm: Complete linkage (curve), Bunch (dots).

- "Rout." generally performed poorly (except for gcc).
- The direct link approach, as used by Bunch, does not perform better than the hierarchical algorithm.
- Combining features gives better results.

As expected, feature "all" gives good design results, this is because the design criterion is based on a similarity metric using this very feature. The good results of "ident." must probably be partly accounted on its redundancy with "all".

Similarly, "file" gave good results with the expert criterion. This seems reasonable since this criterion is based on a comparison with the file system structure of the softwares.

The "cmt" feature is the one having the least amount of redundancy with the formal features. It shows good results for the expert criterion (Mosaic and Linux), but generally poor results for the design criterion. It is hard to tell to what extent the independency of this feature with "all" can be accounted for this bad result. But it must be noted that "cmt" results for the expert criterion are in the same range as the other features (and usually better), whereas, it is significantly lower for the design criterion.

What we defined as data binding features do not behave similarly. We must conclude that either classifying "type" as data binding was a mistake, or this classification (data vs. functional binding) has few influence on the results which depend on other parameters. With only two features in data binding and one in functional binding, it is hard to decide whether this classification is useful or not.

The "var." feature (which is definitely data binding), however, performed as expected on an other aspect: It gives very good results for the telecommunication system which being older, and based on multi-processes having to communicate, relies more heavily on global variables. Tables 3 and 5 also show that "var." is an important feature for the telecommunication system. Note that "var." also gives good design results for gcc, for which we have no explanations.

We were surprised by the general poor results of "rout." for the design and expert criteria. This feature is one of the most natural one to use and there seem to be no reason for its poor performances.

The poorer results of Bunch could be the result of our lack of experience with this tool and the algorithms it uses. Nevertheless, it seems safe to conclude that the direct link approach does not perform better than the sibling approach. We see an advantage in the sibling link approach, however, in the fact that it allows for a wider range of descriptive features (e.g. non-formal features).

We also did some experiments, adding the definitions of software components (types, routines or variables) to their references. For example, for the "var." feature, it means each entity is described with the global variables it defines and the ones it refers to. Overall this gave slightly better results. One may also notice that the "all" feature (the plain line without "x"s) gives good result for the expert criterion (its results for the design criterion are not significant). Our conclusion is that the more information we give

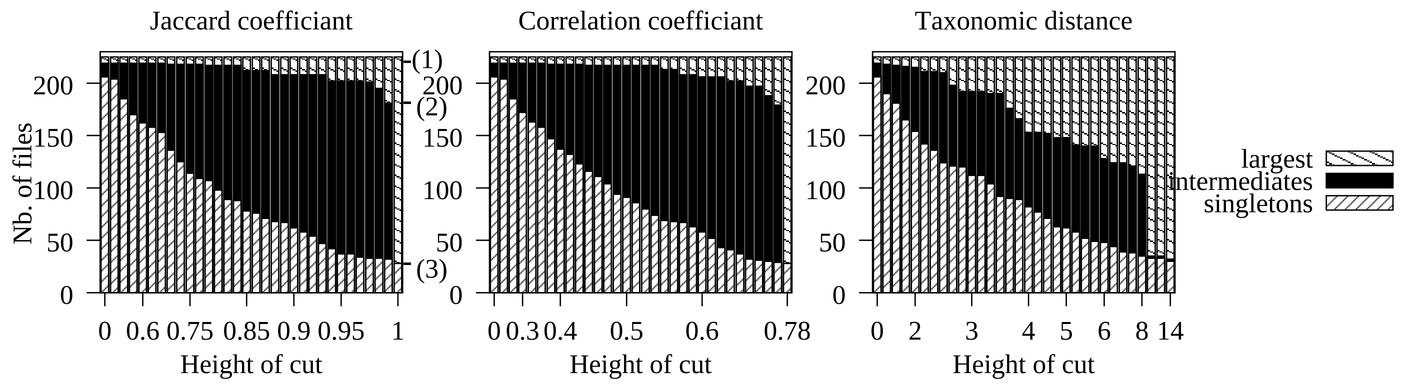

Figure 7: Comparison of three similarity metrics using the size criterion. System: Mosaic; Feature: "macro"; Algorithm: Complete linkage. The marks on the right side of the first graph give the results for the Bunch experiment with the same feature: (3) is the number of singletons, (2) adds to it the number of entities in intermediate clusters and (1) adds to this the size of the biggest cluster.

to the algorithm, the better job they can do, even if the information may not appear very significant at first: Clustering entities on the software components they define is rather strange, since no entities should define the same identifiers.

7.3 Similarity Metrics

Figure 7 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. planetary system configuration. The Simple Matching association coefficient (not shown) and the two distances (Taxonomic distance in the right graph) which consider zero-dimensions tend to give a black hole configuration 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.

All these metrics are based on the sibling link approach. The experiment with Bunch, using a direct link approach, gives similar results (marks (1), (2) and (3) on the right side of the first graph). There is no significant difference between the two approaches.

7.4 Clustering Algorithms

The four agglomerative hierarchical clustering algorithms behave mostly as expected. There is a gradation from complete linkage, to weighted, unweighted and single linkage for each independent metric in the three quality criteria (e.g. figures 8 and 9, two left graphs).

As expected, complete linkage gives more cohesive clusters and single linkage less coupled ones (figure 8).

Single linkage has a tendency to gather all the entities in one cluster. It results in a black hole configuration (see again figure 5).

In both figures, the rightmost graph presents the two metrics for a criterion together (design criterion: cohesion/coupling; expert criterion: precision/recall). This allows better comparison of the algorithms' performances. For the design criterion (figure 8), the ideal point is the lower right corner (high cohesion, low coupling): Complete linkage has the best cohesion, but unweighted linkage give the best compromise between cohesion and coupling. For the expert criterion(figure 9), the ideal point is the upper right corner (high precision and recall), again complete and unweighted linkage seem the two best choices.

We cannot present comparative studies with Bunch hill-climbing algorithm here for practical reasons[12]. The preceding subsections present results for the Mosaic system with indications of Bunch's performances

[12]The Linux system proved too large to be handled with the version of Bunch we used.

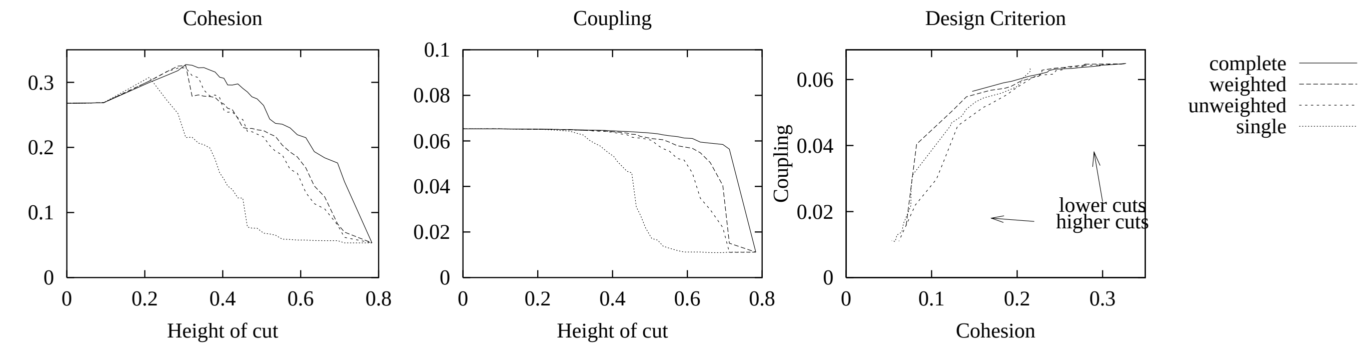

Figure 8: Comparison of four algorithms using the design criterion. System: Linux. Feature: "file". Similarity metric: correlation.

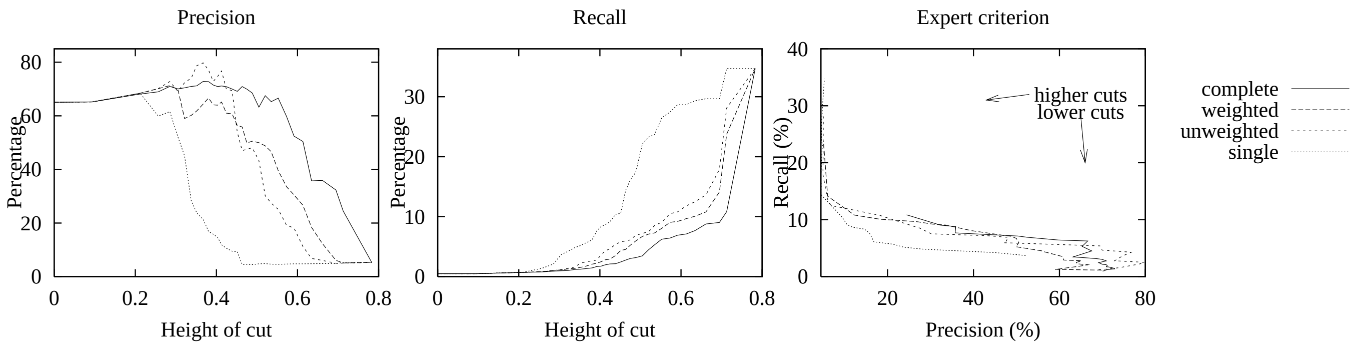

Figure 9: Comparison of four algorithms using the expert criterion. System: Linux. Feature: "file". Similarity metric: correlation.

(see for example figures 7 and 6). They show that the hill-climbing algorithm gives results in the same range as our hierarchical algorithms. Actually, the results are usually a bit worse for Bunch, but, as already stated, this could come from our lack of experience with this tool.

7.5 Practical Tips

In this subsection, we tried to summarize some of the results to help practitioners actually use (hierarchical) clustering for software remodularization.

From our experiments, hierarchical clustering provides as good results as other algorithms. The advantage of the hierarchical is that they can be used to get different partitioning of the system at different level of abstraction. This can prove useful for browsing purposes for example. On the other hand, one must choose the appropriate height at which to cut. We address this issue below.

For algorithm, we suggest complete linkage which gives the best results, especially regarding cohesion. Unweighted linkage is also a good choice if one cares more about coupling, as it provides the best compromise. Single linkage seems to be a popular choice among other researchers see Lakhotia's review [20] but we saw it tends to result in a black hole configuration.

For similarity metrics, we suggest either the Jaccard coefficient or the correlation coefficient. We prefer Jaccard for its conceptual and implementation simplicity.

To decide where to cut is more difficult. Maarek proposes a simple heuristic to do so in [22], but it does not fit the experimental results we got. We propose here our own heuristic.

It is a property of the agglomerative hierarchical clustering that the height of the successive cluster is (non strictly) monotonous, which means, the height of a cluster will always be greater or equal to the height of a preceding cluster (formed earlier by the algorithm) and less or equal to the height of a succeeding cluster (formed later by the algorithm). For our experiments, the evolution of clusters' height typically follows the curves shown in figure 10.

First some identical entities are clustered together, these clusters have a height of zero (i.e. the distance between the entities is zero; they are the same). That is the reason why all curves do not raise from the same point. The curves then have a steep almost vertical slope. It corresponds to a stage where small clusters (two or three entities) are formed from entities which are not identical (actually, they are somehow far apart). Then the curve flattens; there is less difference between the clusters than between the individual entities. The hypothesis is that it corresponds to the creation of subsystems. Although it is not clear for some curves, a short steep slope follows which implies that the clusters are farther apart. The hypothesis is that well designed subsystems are being clustered together. Finally, the curve is flat at height 1 which is the maximum height for the Jaccard coefficient. At this point the rest of the clusters are maximally different one another and are only gathered together to make the final unique cluster containing everything.

We propose to cut in the hierarchy, after the middle flat phase, when the slope becomes steep again. For some descriptive features, this mean cutting just below 1 which is the maximum height for the Jaccard coefficient.

For similarity metrics ill-adapted to software remodularization (e.g. taxonomic distance), the clusters' height curve has a different shape, it looks like a x^2 curve. In this case, entities are very close at the beginning. These are the entities with empty or quasi empty-descriptions.

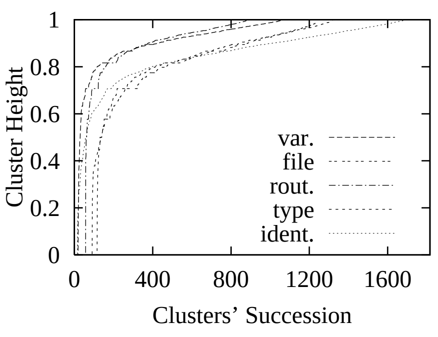

Figure 10: Typical evolution of clusters' height for hierarchical clustering with the Jaccard association coefficient, and the legacy telecommunication system.

8 Conclusion

In this paper, we studied different issues that may influence the clustering results when doing software remodularization with agglomerative hierarchical clustering algorithms: how the entities are *described*, how *coupling* between the entities is computed and what *algorithm* is used. The effect of these parameters is evaluated according to the following quality criteria: *Design* quality, how well they compare to an *expert partitioning*, *size* of the clusters, and evaluation of the amount of *redundancy* among the descriptive features.

From our experiments, we draw the following conclusions:

1. Non-formal descriptive features are a valid choice. They can give as good results as the formal ones, and they offer other qualities: less redundancy, easier to extract, more versatile, higher level of abstraction.

2. 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.

3. The descriptive features may provide very different amounts of information; more 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).

4. There is no fundamental difference between the result with a "direct link" approach or a "sibling link approach". The latter should probably be favored for its generality, for example it allows for use of non-formal features.

5. There does not seem to be any fundamental advantage in choosing hierarchical or non hierarchical algorithm. The hierarchical ones behave has expected and complete and unweighted linkage seem to be the best choices.

6. The quantity of information provided is important: Even features of a priori dubious utility can prove useful when used as a complement with other descriptive feature.

We experimented only with file clustering. Other tests should be done with different entities (routines, classes, processes, etc.) We predict the results would be similar for the same quality criteria. For example, the problem of quasi-empty descriptions would be the same if entities were routines or classes. Note however, that when clustering other entities, the purpose of the clustering may also change. For example in [5] variables are clustered to discover possible classes in procedural code. Although we believe the criteria we used are still valid in this context, other criteria should be considered such as the possibility for a single entity to appear in more than one cluster.

Thanks

The authors whish to express their thanks to other members of the Consortium for Software Engineering Research[13] 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/`).

References

[1] 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.

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

[3] Nicolas Anquetil and Timothy Lethbridge. File Clustering Using Naming Conventions for Legacy Systems. In J. Howard Johnson, editor, *CASCON'97*, pages 184–95. IBM Centre for Advanced Studies, Nov 1997.

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

[5] Tobias Kuipers Arie van Deursen. Identifying object using cluster and concept analysis. In *21^{s}t International Conference on Software Engineering, ICSE'99*, pages 246–55. ACM, ACM press, may 1999.

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

[7] Bunch project (Software Engineering Research Group at Drexel University). `http://www.mcs.drexel.edu/~serg/` (Bunch project).

[8] Elizabeth Burd, Malcom Munro, and Clazien Wezeman. Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity. In *Working Conference on Reverse Engineering*, pages 189–196. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[9] G. Canfora and A. Cimitile. An Improved Algorithm for Identifying Objects in Code. *Software: Practice and Experience*, 26(1):25–48, jan 1996.

[10] 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.

[11] A. Cimitile, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino. Identifying Objects in Legacy Systems. In *5th International Workshop on Program Comprehension, IWPC'97*, pages 138–47. IEEE, IEEE Comp. Soc. Press, 1997.

[12] Richard Clayton, Spencer Rugaber, and Linda 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.

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

[14] Jean-François Girard, Rainer Koschke, and Georg 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.

[15] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data binding. *IEEE Transactions on Software Engineering*, 11(8):749–57, aug. 1985.

[16] Donald A. Jackson, Keith M. Somers, and Harold H. Harvey. Similarity Coefficients: Measures of Co-occurence and Association or Simply Measures of Occurence ? *The American Naturalist*, 133(3):436–453, March 1989.

[17] Thomas Kunz. Developing a measure for process cluster evaluation. Technical Report TI-2/93, Technical University Darmstadt, 1993.

[18] Thomas Kunz. Evaluating Process Clusters to Support Automatic Program Understanding. In *Fourth Workshp on Program Comprehension*,

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

pages 198–207. IEEE, IEEE Comp. Soc. Press, Mar 1996.

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

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

[21] Christian Lindig and Gregor 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.

[22] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[23] S. 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 *Proceedings of the* 6^{th} *International Workshop on Program Comprehension*, pages 45–52. IEEE, IEEE Comp. Soc. Press, june 1998.

[24] Spiros Mancoridis and Richard 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.

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

[26] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A Reverse-engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[27] Philip Newcomb and Gordon Kotik. Reengineering Procedural Into Object-Oriented Systems. In *Working Conference on Reverse Engineering*, pages 237–49. IEEE, IEEE Comp. Soc. Press, Jul 1995.

[28] Sukesh Patel, William Chu, and Rich Baxter. A Measure for Composite Module Cohesion. In *14^{th} International Conference on Software Engineering*. ACM SIGSoft/IEEE Comp. Soc. Press, 1992.

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

[30] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In Mary Jean Harrold and Guiseppe Visaggio, editors, *International Concept on Software Maintenance, ICSM'97*, pages 170–79. IEEE, IEEE Comp. Soc. Press, oct. 1997.

[31] Smart v11.0. Available via anonymous ftp from `ftp.cs.cornell.edu`, in `pub/smart/smart.11.0.tar.Z`. Chris Buckley (maintainor).

[32] Peter H.A. Sneath and Robert R. Sokal. *Numerical Taxonomy.* Series of books in biology. W.H. Freeman and Company, San Francisco, 1973.

[33] Harry M. Sneed. Object-Oriented COBOL Recycling. In *Working Conference on Reverse Engineering*, pages 169–78. IEEE, IEEE Comp. Soc. Press, Nov 1996.

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

[35] M.-A. D. Storey, K. Wong, and H.A. 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.

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

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

[38] Wordnet 1.6 (Cognitive Science Laboratory at Princeton University). `http://www.cogsci.princeton.edu/~wn`.

Appendix

We give here the short list of heuristics we used to automatically decompose identifiers (routines, variables or types' names) into lists of words:

- Cut on word separators (ex: "sub_word"), that is to say the underscore character with most modern languages.
- Cut before an uppercase followed by a lowercase (ex: "subWord", or "SUBWord").
- Cut on a digit following a letter (ex: "Emacs19").
- Also keep the same word to make a version number so that "Emacs19" gives "Emacs" and "Emacs19".