Comparative study of clustering algorithms and abstract representations for software remodularisation

N. Anquetil and T.C. Lethbridge

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 general clustering literature may not apply entirely to the reverse engineering domain. In the paper, the authors study three decisions that need to be made when clustering: the choice of (i) abstract descriptions of the entities to be clustered, (ii) metrics to compute coupling between the entities, and (iii) clustering algorithms. For each decision, our objective is to understand which choices are best when performing software remodularisation. 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, the authors confirm the importance of a proper description scheme for the entities being clustered, list a few effective coupling metrics and characterise the quality of different clustering algorithms. They also propose description schemes not directly based on the source code, and advocate better formal evaluation methods for the clustering results.

1 Introduction

As a software system becomes older, its maintenance becomes more difficult. Software engineers face systems where a minor change in some part can have an unexpected impact on an apparently unrelated part. Consequently, important goals of the reverse engineering community are to help software engineers understand and restructure legacy software as well as to migrate it towards more modern architectures and languages. A key technique for doing this is clustering, which is used to group software components into modules that are meaningful 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 and type of the data, a priori knowledge of the expected result, etc. Reverse engineering is a younger research domain, with still unclear goals [1] and methods that are somewhat ad hoc. In this context, clustering has often been used without a deep understanding of many of the issues involved.

To try to address this, Wiggerts presents in [2] a summary of literature on clustering. This summary is valuable since it lists the possible decisions one needs to make and gives insights about advantages and drawbacks associated with various alternatives. However, as we said, the clustering field is a very sophisticated one, and even summarising it in a single paper is a challenge. As a consequence, Wiggerts' paper 'only' offers conclusions drawn from the general clustering literature, some of which may not apply to the domain of software remodularisation. We present here a study of clustering that is more geared towards the specific needs of reverse engineering. We study three issues: (i) how to describe the entities to be clustered; (ii) the similarity metrics used to quantify the coupling between these entities; and (iii) the choice of clustering algorithms. Our study includes theoretical discussion of important questions one should keep in mind when doing clustering, as well as practical experiments to verify some of our hypotheses. Note that we can only present a few results from the large number of experiments we performed. The interested reader will find the complete results, as well as the clustering programs, on the web: http://www.site.uottawa.ca/~anquetil/Clusters/.

2 Issues in clustering

Software engineers perform reverse engineering to help 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 that have wide applicability. We will not consider more intelligent solutions tailored to solving specific problems
(e.g. [3–5]). 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 perform clustering, we need to consider the following three issues:

- **Entities' description**: how to build an abstraction of the real world in which the entities to be clustered are described according to some scheme.
- **Entities' coupling**: how to define when a pair of entities should be clustered together to make a cohesive unit.
- **Clustering algorithm**: the choice of clustering algorithm to apply to the abstract descriptions of the entities.

The first issue is of utmost importance. Clearly, if we decide to describe files by the 77th character they contain, the result would not be any way useful to software engineers.

Once an abstract description scheme for the entities is chosen, we need to tackle the second issue: When will two entities be considered as forming a cohesive cluster. There are two approaches, which we call direct and sibling link [Note 1]. We can put together entities where one depends on the other, the direct link approach. Alternatively we can put together entities that have the 'same behaviour', the sibling link approach. Fig. 1 gives examples of both approaches.

An important difficulty is that typical software systems are fully connected graphs. To create ‘meaningful independent clusters’, we must decide to ignore (i.e. break) some links. For example, in Fig. 1, the fact that F2.c includes Fa.h is not considered meaningful enough to cluster them together. For software engineers, who may have considerable knowledge about each entity, the task of allocating files to subsystems is often difficult. For a computer program, the task is even more complex, because it must rely only on abstract information, with no semantic information about the actual importance of each link.

For both sibling and direct link approaches, the coupling between two entities is quantified by what we call 'similarity metrics' (see Section 5.1). Clustering algorithms use these similarity metrics to make decisions about whether or not to break links.

At this point, we are left with the third and final issue to be discussed in this paper: the choice of clustering algorithm. There are many alternatives to choose from. It is important to understand that these algorithms do not discover some hidden or unknown structure in a system; instead they impose a structure on the set of entities they are given. They (arbitrarily) decide to ignore some links and favour others — a decision based on the similarity metrics used and on the algorithm itself (e.g. a genetic algorithm or a hierarchical algorithm). The structures imposed by the different algorithms have different qualities. We will discuss this point in Section 5.2.

Clustering is a very broad domain, so we had to pick certain experiments on which to focus our investigations. We experimented with a large set of possible entity-description schemes, ranging from ‘traditional’ ones (e.g. based on data binding) to more innovative ones (e.g. based on comments). With regard to coupling, we conducted a few experiments to compare the direct and the sibling link approaches. We chose the sibling link approach, and the similarity metrics pertaining to it, for more extensive study. Finally, on the third issue, we performed a few experiments with a hill-climbing algorithm [Note 2] but concentrated mainly on hierarchical clustering algorithms.

3 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 [2]. Our paper is intended as a continuation of his work. Wiggerts gives a summary of many classical clustering methods; we take some of these methods and test them to compare their relative strengths and weaknesses for reverse engineering.

Lakhotia [9] proposes a comparison framework for 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 and Trudeau [10] and Storey et al. [11] give a comparison of various reverse engineering tools; however, their work is at a higher level of abstraction than ours: whereas they compare tools, discussing issues such as the quality of the interface, we compare clustering algorithms that might be used by the tools.

Girard et al. [5] 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’ [12–14]. Again, this is not a statistical clustering method, and falls outside the scope of this paper. We believe that clustering remains a valuable approach in that it extracts only important concepts from a data set, whereas concept analysis extracts all concepts and may therefore produce information overload for the user (e.g. see [15, 16]).

Many publications present and evaluate a single software remodularisation method. They do not 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.

4 Entities to cluster and their descriptions

In taxonomy, entities are organisms (animals or plants) that should be classified into species. Each entity is described using

---

Note 1: Patel in [6] uses a cocktail-party analogy to describe these two approaches.

Note 2: We are grateful to S. Mancoridis at Drexel University for providing the Bunch tool [7, 8].

IEE Proc.-Softw., Vol. 150, No. 3, June 2003
a vector of features like 'number of leaves', 'laying eggs' or not, etc. The features may come from many different sources, based on anatomy, geographical distribution, temporal distribution, behaviour, chemical properties, etc.

For each entity, one first lists all the features it possesses; clustering is then used to group together those entities that have features with the same values, i.e. using what we describe as the 'sibling link' approach (Section 2). In addition, for software remodularisation the 'direct link' approach can also be used to do the clustering. This will be studied in Section 4.3. For software remodularisation, 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 Fig. 1 could be described according to the files they include as follows:

\[
\begin{align*}
F_a.h: & \ ( ) \\
F_b.h: & \ ( ) \\
F_1.c: & \ (F_a.h) \\
F_2.c: & \ (F_a.h, F_b.h) \\
F_3.c: & \ (F_a.h, F_b.h) \\
\end{align*}
\]

We will refer to a general label like 'included files' as a feature kind, and one file in this group (e.g. \(F_a.h\)) as a feature or dimension.

Some features, such as how many references to a particular global variable are found in a file, are quantitative; we can refer to each feature as a dimension, and the quantity is the co-ordinate in that dimension. Where an entity does not refer to a particular thing, the co-ordinate (quantity) for that feature will be zero. In the file inclusion example above, the co-ordinate would be 1 if the file is included and 0 if it is not. The above descriptions might then become:

\[
\begin{align*}
F_a.h: & \ (F_a.h = 0, F_b.h = 0) \\
F_b.h: & \ (F_a.h = 0, F_b.h = 0) \\
F_1.c: & \ (F_a.h = 1, F_b.h = 0) \\
F_2.c: & \ (F_a.h = 1, F_b.h = 1) \\
F_3.c: & \ (F_a.h = 1, F_b.h = 1) \\
\end{align*}
\]

Note that each entity has a vector of the same length, whether or not it includes a file.

Features such as those described above are based purely on a syntactic analysis of the code. We call them formal features. Although formal features are predominant in software remodularisation, in Section 4.2 we discuss the availability and usefulness of nonformal features.

4.1 Formal features
We classify a feature as formal [Note 3] if it consists of information that has direct impact on, or is a direct consequence of, the software system’s behaviour. For example, describing an entity using the routines to which it refers is formal because if we change a routine call in the code, we should expect a change in the system’s behaviour.

Formal features are the most commonly used in reverse engineering research; in [9], Lakhotia lists 12 works on clustering, among which only two use nonformal features (one exclusively [20; Note 4] and the other a mix of formal and nonformal [21]).

We experimented with the following formal feature kinds:

- **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 — used, for example, in [6, 22] to describe files or in [23] 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 [4, 21] to describe files or in [24] to describe classes.
- **File**: Files included by the entity. May only be used to describe files as in [4, 25].
- **Macro**: Macros used by the entity. This feature kind is only available in some languages (like C). It is most relevant to file or macro entities, that is to say: Given the fact that macros are extraneous to the language’s grammar, it would be difficult to define generally what it means for a type or a function to refer to a macro. Several well defined conventions exist for the use of macros; these could allow for greater use of macros as a descriptive feature kind.
- **All**: Union of the above five feature kinds.

One can imagine other formal feature kinds, like interprocess calls [26], or past bug fixes that impacted the entities.

**Observation 1**: The choice of feature kinds for software is more limited than in taxonomy.

Combining feature kinds is easily achieved by describing an entity using the union of its features for all the feature kinds. This is possible because all the features have very similar form [Note 5].

Most of the literature reports experiments with formal features, extracted from the same source: the actual code. Hence, the lack of choice could be aggravated by redundancy.

**Hypothesis 1**: There are redundancies among all the formal feature kinds extracted from the source code.

For example, all files that use a particular variable will include the same header file describing that variable. This would introduce a correlation between the 'var.' and the 'file' feature kinds. Another example would be a routine referring to one of its parameters’ types ('type' feature kind), forcing a calling routine ('rout.' feature kind) to refer to the same type as one of its variables’ types. These redundancies are imposed by programming languages to provide for error checking.

Some researchers propose to reduce the number of features for efficiency reasons (the programs have less data to process) or to remove noisy data. Van Deursen [12] does not consider 'utility' features which have a high fan-in because they may be used in all subsystems; Schwanke and Hanson [27] remove features used in only one entity because they do not contribute to the similarity of two entities (they could, however, contribute to differentiate entities). We did not study this aspect.

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

4.2 Nonformal features
To overcome the problems associated with the formal features, we propose to use nonformal ones. We define a

Note 3: Note that in [17], Gannod and Cheng give another definition of formality and informality based on the use or not of formal methods and formal languages (e.g. [18, 19]).

Note 4: This cited work is not really about reverse engineering, but is more related to information retrieval and document classification.

Note 5: In taxonomy, on the other hand, features may be Boolean, discrete or continuous, bounded or not, etc.
nonformal feature to be one that uses information having no direct influence on the system's behaviour. A typical nonformal feature kind is the naming of routines, since changing the name of routine has no impact on the behaviour of a system.

In a previous study [28], we showed that for a particular legacy software system, the file naming convention was the best feature kind to recover the modular structure of small subsystem examples given to us by software engineers. Other nonformal feature kinds can 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 following two: 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. This is a naming-convention feature kind similar to what is used in [29].

Cmt: References to words in comments describing the entity. For better results, the words are filtered using a standard stop list [Note 6] and stemmed [Note 7].

For legacy software, documentation is often missing or considered outdated. Some researchers argue that nonformal features are not reliable because nothing guarantees that the information extracted will correspond to the actual behaviour of the system. Sneed, for example, states that 'in many legacy systems, procedures and data are named arbitrarily' [32]. Clearly in such cases, the 'ident.' nonformal feature kind would prove useless. However, many legacy systems use significant identifiers, intended to help other software engineers to 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. [33–35]). This leads us to our second hypothesis.

Hypothesis 2: Nonformal feature kinds can be used for software remodularisation.

Assuming they provide relevant information, nonformal features can offer some or all of the following advantages over formal ones, we will discuss each of these in the subsequent paragraphs. Nonformal features:

- have a higher level of abstraction (describe high-level intent of the code)
- are closer to human understanding
- have less redundancy
- are easier to extract
- are more versatile
- contain more information (in particular, result in fewer entities with empty descriptions).

Nonformal feature kinds, such as the ones we experimented with, are intended for human readers. As such it seems reasonable to assume that they provide information at a higher level of abstraction than features based purely on the syntax of the code. Therefore, it would probably be easier to extract abstract concepts from the resulting clusters.

Observation 2: This issue of levels of abstraction is an important one and should be formally tested.

Also, since informal features are intended for human readers, the results should be readily understandable to the software engineers.

Hypothesis 3: Nonformal feature kinds have less redundancy with formal ones than the formal ones among themselves.

As mentioned in the preceding section, one of the difficulties with the formal features is that there is redundancy among them due to the nature of modern programming languages. We hope that, since nonformal features are not so tightly restricted by programming languages, they would not exhibit this redundancy, and instead provide different and independent points of view about the system. We designed two experiments, discussed in Section 6, to test this issue.

Observation 3: The nonformal feature kinds we chose are much easier to extract than the formal ones.

For example, extracting words from comments is a simple task once one knows the comment delimiters. Extracting the formal feature kinds requires either that one has a robust parser for the language, or else that one has tracing capabilities (e.g. for features based on interprocess communication.) Features like C's conditional compilation and macros raise real problems (yet to be fully solved) when it comes to parsing a program.

Observation 4: The nonformal feature kinds we chose are more versatile than the formal ones.

These feature kinds are independent of the programming language and can even be applied to sources of information other than code (e.g. scripts associated with databases, compilation or configuration).

Hypothesis 4: Nonformal features provide more information, on average, than the formal ones.

Since they are more versatile, nonformal features are almost always present in all entities. On the other hand, it is common for an entity to have no reference to a given formal feature kind. For example, a system with well encapsulated data would have very few references to global variables. Our results show that for many formal feature kinds the percentage of entities having an empty description is high (more than 10% and up to 65%). The lack of enough information provided by each feature kind farther restricts an already limited choice. On the other hand, we found our nonformal feature kinds to have below 5% of empty descriptions.

We do not presume, however, that every nonformal feature kind is necessarily useful. It is one of the purposes of this paper to study and compare two of these nonformal feature kinds.

4.3 Direct or sibling links

The direct and sibling links presented in Section 2 are two different ways for computing the coupling between entities. Although this distinction between types of link is mainly of concern in the discussion about similarity metrics, it has some bearing on the entities' descriptions.

Observation 5: The sibling link approach is more versatile than the direct one.

The direct link approach has an appealing simplicity; that is, if a routine calls another routine, the two are coupled to some degree. However, using only the direct link approach limits one to features that record dependencies between the types of entities that one is trying to cluster, e.g. files including other files, or structured types built from other types. Among other things, this rules out nonformal features. The sibling link approach, on the other hand, puts fewer constraints on the form of the descriptions; all it requires is that it be able to compute the similarity between them.

Notes:

Note 6: We used the stop list from the information retrieval system Smart [30].

Note 7: 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 [31].
Table 1: Summary of characteristics of different feature kinds

<table>
<thead>
<tr>
<th>Feature kind</th>
<th>var.</th>
<th>type</th>
<th>rout.</th>
<th>macro</th>
<th>file</th>
<th>all</th>
<th>cmt</th>
<th>ident.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal Link</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Sibling Link</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>sibling</td>
<td>yes</td>
</tr>
</tbody>
</table>

'both' means suited for the direct and sibling link approach

Therefore we experimented mostly with sibling links. All the feature kinds presented in Section 4.1 have been used as sibling links. We also performed a few experiments using all the formal features as direct links. It was not clear in advance which approach would work better since we cannot use the taxonomy domain as a reference. We found only one paper comparing the two approaches: in [36] (see also [26]), Kunz reports that he found direct link to be inferior to sibling link.

Hypothesis 5: The sibling link approach does not perform worse than the direct link approach.

Table 1 gives a summary of the different characteristics that each of the eight feature kinds 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. [37] the choice of a proper similarity metric has more influence on the result than does the clustering algorithm.

These metrics may use the direct link or the sibling link approach. 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 between themselves. For finer tuning, links can be weighted (if an entity calls a routine)

Strong negative correlation as a sign of similarity, while others do not. We opted for the second possibility (negative correlation is a sign of dissimilarity).

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 and Sokal [38], they are close to the correlation coefficients.

We experimented with two distance coefficients, three association coefficients and one correlation coefficient, all proposed in [38; Note 8].

Jaccard: (association [Note 9]) $sim(X, Y) = 1 - (a/(a+b+c))$.

Simple matching: (association [Note 9]) $sim(X, Y) = 1 - ((a+b)/(a+b+c+d))$.

Swensen-Dice: (association [Note 9]) $sim(X, Y) = 1 - (2a/(2a+b+c))$.

Correlation: (correlation [Note 10]) $sim(X, Y) = \sqrt{(1-r)/2}$.

For example, the similarity between F1.c and F2.c of Fig. 1 according to the Jaccard association coefficient where entities are described according to the files they include ('file' feature kind) would be: $a = \|\{F1.h\}\| = 1$, $b = \|\emptyset\| = 0$, $c = \|\{F2.h\}\| = 1$ and $sim(F1.c, F2.c) = 1 - (1/(1 + 0 + 1)) = 0.5$

These metrics, useful in taxonomy, may not all be well adapted to software remodularisation. In taxonomy, the number of features (dimensions) for one feature kind is much less than it is for software remodularisation. A feature kind such as 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.

Observation 6: Traditionally, feature kinds for software remodularisation have a large number of dimensions, of which very few will be non-null for any particular entity.

We distinguish two particular conditions that may raise problems:

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

Quasi-empty description: Other entities have very few references for a given feature kind. They are described by a vector with only one or two nonzero 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 choose 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. Bunch [7, 8], which is used to experiment with the direct link approach, makes the same choice.

**Hypothesis 6:** Similarity metrics that consider zero-dimensions as a sign of similarity tend to cluster all entities together in a huge meaningless cluster.

Entities with quasi-empty descriptions all have very similar descriptions and will often be considered strongly coupled. Metrics that consider zero-dimensions as a sign of similarity (such as the Simple Matching association coefficient or the two distance coefficients), will be misled by the high proportion of zero-dimensions. 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. This problem is linked to observation 6.

**Hypothesis 7:** Feature kinds with fewer dimensions are less subject to the problem identified in the previous hypothesis regarding zero-dimensions.

For example, the 'file' feature kind tends to have fewer dimensions (there are fewer files to include than routines to call or global variables to use) and should give better results than other feature kinds with the similarity metrics that do consider null-dimensions as a sign of similarity.

Note that, whereas it does consider null-dimensions as a sign of similarity, the correlation coefficient will be little affected by them. This coefficient looks for similar variations in values; many identical points have barely more influence on the correlation coefficient than a single one. Therefore, for the correlation coefficient, many null values are very close to a single null one, and it will not be impacted by the large amount of zero-dimension in the descriptions.

### 5.2 Clustering algorithms

The next step consists of choosing a clustering algorithm. In this paper, we concentrate on agglomerative hierarchical algorithms. For software remodularisation, non-hierarchical algorithms have also been used (see [9]). Again, we performed a few experiments with non-hierarchical algorithms using Bunch [7, 8]. It uses a hill-climbing algorithm, which tries to minimise an objective function that subtracts the average 'inter-connectivity' (Bunch's measure of coupling) of the partition from the average 'intra-connectivity' (Bunch's measure of cohesion).

Agglomerative hierarchical algorithms start from the individual entities, gathering them, two by two, 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 do not need any extra information such as the number of clusters expected or the possible region of the search space in which to look for clusters. 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 [Note 11]. The distance between a newly created cluster

---

**Note 11:** Similarity metrics compute similarity between individual entities. The algorithms extend this to clusters.

---

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

(e.g. C4, in Fig. 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:

- **Single linkage** [Note 12]: (or closest neighbour rule) the new distance will be \( d = \min(d_1, d_2) \).
- **Complete linkage:** (or furthest neighbour 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:** \( d = (|C_1||d_1| + |C_2||d_2|)/(|C_1| + |C_2|) \). In this one, we take into account the size of the clusters C1 and C2, such that all entities have the same weight (i.e. they are not weighted).

Which of the above algorithms is chosen can have a considerable influence on the resulting clusters. For example, single linkage is known to favour noncompact but more isolated clusters whereas complete linkage usually results in clusters that are more compact but less isolated (see Fig. 3). Unweighted and weighted average linkages are intermediate on the following scale: complete, weighted, unweighted and single linkages.

**Hypothesis 8:** Single linkage results in clusters that are less coupled, while complete linkage results in clusters that are more cohesive.

There is no a priori reason to favour cohesion over coupling, but because the descriptive vectors (on which cohesion and coupling are based, see Section 6.2) 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 good coupling. It is our experience that one often needs to favour one metric at the expense of the other.

Table 2 presents a summary of the three parameters we studied.

**Note 12:** Do not mistake the Simple Matching coefficient (Section 5.1) for the single linkage algorithm.

---

**Fig. 3** Influence of clustering algorithm on clusters (from [2])

Left: single linkage; right: complete linkage

---
If we cut at height 0, the partition will contain only singleton clusters, and considering only the topmost clusters (see Fig. 4), the height of the clusters it contains. If we cut at the maximum height, the entire system will be the only cluster.

One can draw a useful analogy between the height of a cluster and its ‘diameter’. The farther apart two entities are (less similarity), the greater is the height of a cluster that contains both of them. 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 Section 5.1 is the farthest apart [Note 13] that any 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 to be useful for analysing the behaviour of the clustering methods.

### 6 Evaluation criteria

Evaluation criteria are central to any formal comparison as well as to the precise definition of a scientific research domain. We will use two types of criterion: general criteria to evaluate the quality of the clustering, and specific criteria to test some particular hypotheses. This part is made difficult by the following observation.

**Observation 7:** There are few well established formal criteria available to compare the results of various remodularisation approaches.

The general criteria should aim at measuring the quality of a clustering approach in its normal use, which is to remodularise 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:

1. Actually represent the system (and not an ideal view of the domain, for example).
2. Make sense to the software designers, i.e. extract modules implementing known concepts.
3. Greatly reduce the amount of information that software engineers have to cope with (abstract the system).
4. Be adaptable to different needs, e.g. help novices understand the system or help experts assert the consequences of a modification in the code.
5. Be general, i.e. adaptable to different software systems, programming languages, etc.
6. Be stable, i.e. give similar results when minor changes are introduced to the entities being clustered (see [39]).

Unfortunately, not all of these properties are easily measured formally. For example, remodularisation techniques are often evaluated manually by an expert in the system (which addresses properties 1 and 2 above). Owing to the fact that we perform thousands of experiments to analyse various combinations of variables, manual evaluation is not feasible in our work. The reverse engineering literature suggests two alternative evaluation criteria that can provide a suitable substitute, while still verifying that the approach being evaluated represents the system and makes sense to users:

- **Design criterion:** Clusters should reflect a good design (as evaluated by cohesion and coupling).
- **Reference decomposition criterion:** Clusters should match a reference decomposition (established by ‘experts’).

We added the following criterion, to verify that the approach being evaluated makes sense to users:

- **Size criterion:** To be useful, clusters should have an appropriate size. For example, one very large cluster and many small ones is undesirable.

In addition, some hypotheses will be evaluated using specific criteria based on information theory (entropy criterion) and a method known as k-nearest neighbours (nearest neighbours criterion). We will discuss each of the evaluation criteria in the following five sections.

### 6.1 Reference decomposition criterion

An important way of evaluating clustering approaches is to have a real expert give his or her opinion of the results. Owing to the very large number of experiments we...
performed, it was not possible to proceed that way, and we
used an automated comparison with a reference decom-
position. We call this the reference decomposition cri-
tion. This criterion will test Hypotheses 2 and 5.

The source files for two of our systems (Linux and Mosaic) are organised in several directories which form a reason-able decomposition of the systems. We used them as reference 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 sensitive to design drift during maintenance.

Although the clustering algorithms and the directories both potentially provide tree structures, we compared them as partitions of the system. In the last section, we 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 ones containing only files) as forming subsystems. This should match the abstraction level of the clusters extracted, which is usually low.

The difficulty with 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 to consider 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 [Note 14] for a given partition as follows:

Precision: the percentage of intra-pairs proposed by the clustering method which are also intra in the reference partition.

Recall: the percentage of intra-pairs in the reference partition which were found by the clustering method.

We will often witness good precision and low recall. This is a consequence of the clustering methods extracting smaller clusters than the ones in the reference partition. Since the clusters are smaller, they have fewer intra-pairs and a higher probability that they are correct (good precision); at the same time there will be many missing intra-pairs (bad recall). In other words, the extracted clusters are at a lower abstraction level than the reference partition because each cluster covers fewer entities.

One problem with this automated comparison is that it does not have the flexibility of human evaluation. An expert judging a partitioning will try to establish if it is reasonable, whereas our criterion will reject a possibly interesting partitioning if it does not happen to match the pre-selected reference. On the other hand, human evaluation is highly subjective and not easily quantified.

### 6.2 Design criterion

Software engineers are more likely to be interested in cluster sets that correspond to a good design of the system. Design quality is traditionally evaluated using cohesion and coupling (e.g. [41]), which we now briefly describe. This design criterion will be used to test hypotheses 2, 3 and 8.

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 (usually formal features: Section 4.1), and the coupling between entities is evaluated by some kind of similarity metric. We use a similarity metric (proposed in [22, 6]) different from those described in Section 5.1:

\[
\text{Similarity between entities } \text{[Note 15]} = \text{sim}(e_i, e_j) = \frac{(X^T \cdot Y)/(||X|| \cdot ||Y||)}{1/(1 \times \sqrt{2})}
\]

Cohesion of a partition is the average similarity between any two entities clustered together (i.e. any intra-pair; see preceding 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)=(1, 0) and (F2,c)=(1, 1) in Fig. 1 is \(\text{sim}(F1,c, F2,c) = 1/(1 \times \sqrt{2})\).

This similarity metric does not consider null-dimensions as a sign of similarity.

This design criterion is not entirely objective when applied to automatically generated clusters. As we have discussed, clustering algorithms partition the set of entities, trying to optimise intra-cluster similarity and to minimise inter-cluster similarity. This computation is based on a similarity metric between abstract descriptions of the entities. The design criterion measures cohesion and coupling between the clusters, also based on a similarity metric between abstract descriptions of the entities. In other words, we measure the quality of the results with methods similar to those we used to obtain the results. Even if the similarity metrics used in the two cases are different, and if we are careful to use two different feature kinds, some doubts remain about the validity of this criterion. Remember, for example, the interdependence between some formal features we hypothesised in Section 4.1.

Similarly, the design criterion is probably biased towards the unweighted average linkage algorithm. Both approaches compute an (unweighted) average of some similarity metric between the members of the clusters.

### 6.3 Size criterion

In [42], Hutchens and Basilii draw an analogy between partitions of a system and star systems. They define three types of system:

- The **planetary system**, where several subsystems (planets) are interconnected to form the system. This is the ideal case.
- The **black hole system**, which has no visible planets, one key subsystem absorbing everything revolving around it.
- The **gas cloud system**, which 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.

This criterion will mainly test the two hypotheses regarding the pertinence of similarity metrics that consider zero-dimensions as a sign of similarity (hypotheses 6 and 7).

Since the hierarchical algorithms we use produce trees of clusters, with leaves being singletons and the root containing all entities, we need to be more precise about what we consider to be a gas cloud or a black hole configuration.

---

Note 14: Precision and Recall are standard metrics in information retrieval; see [40] for example.

Note 15: \(X\) and \(Y\) are vectors describing the two entities \(e_i\) and \(e_j\); \(||X||\) and \(||Y||\) are their Euclidean norms; \(X^T\) is the transposed vector.
We define a **black hole** configuration as a situation where the algorithm tends to create one big cluster that grows regularly during the clustering process and drags, one after the other, all entities to it. We define a gas cloud configuration to be a situation where the algorithm tends to create very small clusters and then 'suddenly' clusters all of them into one big cluster near the top of the hierarchy.

We will quantify this criterion with three values, which are: the number of singleton clusters, the number of entities in the largest cluster, and the number of other entities (in the intermediate clusters). We would like to avoid solutions having many singleton clusters (a gas cloud) or only one huge cluster (a black hole). Fig. 5 shows an example of a good and a bad partitioning method. On the left, we can see the number of singleton clusters decreasing steadily, with 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 clouds were always a consequence of the special treatment we imposed on entities with empty descriptions (Section 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 clouds arise as a sign of a bad feature kind (many empty descriptions) and it is simple to detect this beforehand (see Section 4.2). The black hole configuration, on the other hand, is usually a consequence of an ill-adapted algorithm or similarity metric. The two problems can also combine as in the right graph in Fig. 5.

Note that, for the left graph, the intermediate clusters completely disappear in the very last stage of the clustering process at 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, which actually corresponds to the clustering process ignoring these entities (as also proposed by Mancoridis et al. [8]).

### 6.4 Entropy criterion

We referred several times to the quantity of information a feature kind can provide to the clustering process (hypothesis 4). We also mentioned a possible redundancy (or correlation) between some feature kinds (hypotheses 1 and 3). These hypotheses can be evaluated using information theory.

In information theory, the quantity of information of an event is equal to its uncertainty. The uncertainty of an event $e$ depends on the probability of this event, $P(e)$, and is measured by the function: $h(e) = -\log P(e)$.

Using this, we can compute the quantity of information conveyed by the fact that an entity refers to a particular feature $f$, given $P(f) = (#\text{ref To } f) / (#\text{entity})$ (i.e. the number of entities referring to $f$ divided by the total number of entities).

The average quantity of information of a set, $X$, of possible outcomes of an event is given by the entropy function: $H(X) = \sum_{i} P(x_{i}) h(x_{i})$ (i.e. sum of the information of each possible outcome weighted by the probability of this outcome). In our case, an 'event' would be a feature (dimension of a feature kind), and there are two possible outcomes: an entity either refers to the feature or not. From this, we can compute the entropy of a single feature: $(P(\text{ref}) \cdot h(\text{ref}) + P(\text{not ref}) \cdot h(\text{not ref}))$.

Finally, we can compute the average entropy of a particular feature kind which should give us an idea of the amount of information this feature kind provides. However, one should remember that the amount of information a particular feature conveys may have little relation with its usefulness to remodelarise a system. The fact that an entity refers or not to a particular global variable could carry little information per se and yet this global variable could be very useful to help pinpoint one particular subsystem.

Another measure of the information that a feature kind can provide is the proportion of entities with null descriptions it contains (discussed in Section 7.1).

Information theory also allows one to evaluate the correlation between two events. Let us define:

$$H(X) = \sum_{i} P(x_{i}) h(x_{i}) = \sum_{i,j} P(x_{i}, y_{j}) h(x_{i}, y_{j})$$

$$-\log P(x_{i}, y_{j}) = \sum_{i,j} P(x_{i}, y_{j}) h(x_{i}, y_{j})$$

The joint entropy of $X$ and $Y$: it represents the average quantity of information of knowing whether an entity refers to feature $X$ and to feature $Y$.

$$H(X|Y) = \sum_{i,j} P(x_{i}, y_{j}) h(x_{i}|y_{j})$$

The conditional entropy of $X$ knowing $Y$: this is the average quantity of information that remains about (an entity referring to) $X$ when we know whether an entity refers to $Y$.

$$H(Y) = H(X) - H(Y|X)$$

The mutual information between $X$ and $Y$: this is the average reduction of uncertainty about (an entity referring to) $X$ when we know whether an entity refers to $Y$.

The relations between all these values are given graphically in Fig. 6. Our intent is to use them to evaluate the redundancy between two feature kinds. The idea is to compute the mutual information between any two features from two different feature kinds. Redunacy between two

---

**Fig. 5** Size criterion for two algorithms

The algorithm with more entities in intermediate clusters (complete linkage on the left) is better. System: gcc. Feature: 'route'. Similarity metric: Jaccard coefficient.
feature kinds should appear as a high average mutual information between their respective features. However, we saw that each feature kind has many features (Observation 6) and we cannot expect one feature (say a reference to a routine) to be correlated with all the features of another feature kind (say reference to all types); rather, we expect one feature to be highly correlated to a single (or only a few) other features. For example, a reference to one routine is likely to be correlated with features of another system, whereas a variable name can appear in the final result whether or not it carries a lot of information.

In our experiments, we used $n = 2$. This algorithm is not symmetric; bestMI(X, Y) usually differs from bestMI(Y, X). Since, in step x.6, we only consider the few $n$ highest mutual informations, it is probable that some features of feature kind Y never appear in the final result because they intrinsically carry little information and, therefore, cannot have a high mutual information with anybody else. On the other hand, all features of feature kind X will enter in the final result whether or not they carry a lot of information.

To help understand and remember this nonsymmetry, we use the notion of coverage. We will say that bestMI(X, Y) gives an idea of how well X is covered by Y (see mathematical definition of cover(X, Y) a little later). The intuition is that, if Y covers X well, all features of X have high correlation with at least $n$ features of Y. In fact, saying that Y covers X well could actually be the consequence of a subset of Y (with at least $n$ elements) covering X well.

The quantity bestMI(X, Y) needs to be normalised (i.e. divided) by the average joint entropy $H(XY)$. This is to say, the quantity of information common to X and Y needs to be normalised by the total quantity of information that X and Y carry. Again have a look at Fig. 6 for the relationship that exists between $H(XY)$ and $H(X)$, $H(Y)$.

To get a notion of what is a good coverage, we also compute cover(X, X'). Again, this value and the idea of 'covering' could prove counter-intuitive, as cover(X, X) is not a maximum value for coverage. For $n > 1$, we can find Y such that Y covers X better than X itself, for example if Y is very little correlated between the features inside X, whereas each feature of X is highly correlated to at least $n$ features of Y.

### 6.5 Nearest neighbour criterion

To compare feature kinds between themselves, we will also use the k-nearest neighbours criterion. For each entity, this criterion will compare the entity's k-nearest neighbours calculated using one feature kind, with the k-nearest neighbours calculated using another feature kind. The more nearest neighbours in common, the more similar the two feature kinds should be. This criterion was used in [27], for example.

A problem with this criterion is that it is likely to be dependent on the similarity metric used. That is to say, two feature kinds could look very similar when using one similarity metric and less so when using another. Another problem is to decide for a proper value of $k$.

The k-nearest neighbours can also be used to estimate the quality of a similarity metric, using a reference partitioning (the same as in Section 6.1). We can pick each entity and look how many of the k-nearest neighbours according to the similarity metric studied are actually in this entity's subsystem in the reference partitioning.

### 7 Evaluating the hypotheses

We will now present the results of some experiments to evaluate the hypotheses formulated in the first part of the article. In the experiments, entities are files and the clusters may be thought of as subsystems. Each file is described according to eight feature kinds, labelled as follows: Type (types referred to in the file, references include definition of the types and their use); Var. (global variables referred to in the file); Rout. (routines called); File (files included); Macro (macros referred to); All (union of the five previous formal features); Ident. (words in identifiers referred to in the file); and Cmt (words in comments). These feature kinds were discussed in detail in Section 4.

We experimented with four systems: Linux kernel [43], Mosaic [44], gcc [45] and a real world legacy telecommunication system. Table 3 gives some information about the four systems. Note that, in the legacy telecommunication system, the high number of global variables referenced is due to the nature of the system which uses shared memory to exchange information between its numerous processes.

The design criterion (Section 6.2) uses the 'all' feature kind (see Section 4.1). The rationale for this choice is that, first, we wanted a formal feature because the use of nonformal features is not yet well established and,

![Table 3: Some information on the four systems with which we experimented, and number of dimensions for each feature kind](data:image/png;base64, Base64 encoded image data)
second, we did not want to give an advantage to the basic formal features.

We also carried out a few experiments with Bunch, a clustering tool using direct links between the entities. We experimented with Bunch on Mosaic using all formal features. Bunch offers two different algorithms: hill-climbing and genetic, both having an optimal and a sub-optimal version. For efficiency reasons we experimented mainly with the sub-optimal hill-climbing algorithm. This algorithm finds a local optimum for its own version of the design criterion. We also performed one experiment with the optimal hill-climbing algorithm, using the ‘all’ feature kind. The results according to our quality criteria were similar to the sub-optimal results for the same feature.

It is impossible to present here all experimental results. We will concentrate on Linux and Mosaic, the two systems to which the reference criterion applies. The results are consistent for the four systems. All the results, programs and input data are available on the web at: http://www.site.uottawa.ca/~anquetiliClusters."

**Table 4** Average quantity of information of all feature kinds for the four systems

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>Mosaic</th>
<th>Linux</th>
<th>Telecom</th>
</tr>
</thead>
<tbody>
<tr>
<td>var.</td>
<td>0.087</td>
<td>0.054</td>
<td>0.019</td>
<td>0.013</td>
</tr>
<tr>
<td>type</td>
<td>0.085</td>
<td>0.086</td>
<td>0.033</td>
<td>0.015</td>
</tr>
<tr>
<td>macro</td>
<td>0.107</td>
<td>0.069</td>
<td>0.021</td>
<td></td>
</tr>
<tr>
<td>rout.</td>
<td>0.088</td>
<td>0.071</td>
<td>0.020</td>
<td>0.011</td>
</tr>
<tr>
<td>file</td>
<td>0.202</td>
<td>0.120</td>
<td>0.084</td>
<td>0.040</td>
</tr>
<tr>
<td>all</td>
<td>0.095</td>
<td>0.072</td>
<td>0.023</td>
<td>0.013</td>
</tr>
<tr>
<td>ident.</td>
<td>0.170</td>
<td>0.115</td>
<td>0.042</td>
<td>0.056</td>
</tr>
<tr>
<td>cmt</td>
<td>0.185</td>
<td>0.132</td>
<td>0.054</td>
<td>0.063</td>
</tr>
</tbody>
</table>

**Hypothesis 1:** There are redundancies among all the formal feature kinds extracted from the source code.

**Hypothesis 3:** Nonformal feature kinds have less redundancy with formal ones than the formal ones among themselves.

We proposed to evaluate them using two criteria: one based on k-nearest neighbours and one based on information theory.

Table 6 gives the results of the information theory experiment for the gcc system. Each row shows how a feature kind covers other feature kinds. Conversely, the columns show how well feature kinds are covered. The last row and column give an idea of how well a feature kind is generally covered by others or generally covers others. Finally, the last number provides an idea of a mean value of coverage for this system.

Results appear to be independent of the quantity of information contained in each feature kind as measured in the previous experiment. For example, the three feature kinds with the highest quantity of information (previous experiment) behave differently in this experiment, as do the three feature kinds with the lowest quantity of information.

From our experiments we established the following scale for the gcc system: coverage is considered high over 0.375, such that all feature kinds cover themselves well. Coverage is considered average between 0.245 and 0.375, such that the mean value (0.310) marks the centre of this interval. Coverage is considered low below 0.245. Scales for other systems would differ in values from this one.

The salient points are:
- The ‘cmt’ feature kind is the least covered by all other feature kinds (e.g. lower value in the last row), and it is one of those who cover least the other feature kinds (e.g. second lower value in the last column).
- The ‘ident.’ feature kind presents no salient characteristic; it covers and is covered as the average formal features.
- The ‘all’ feature kind best covers the other feature kinds.
- The ‘file’ feature kind covers the other feature kinds the least and is normally covered by the others.

The first point seems to agree with our hypotheses, whereas the second point does not. In retrospect, it seems normal that a nonformal feature kind based on words extracted from identifiers has some redundancy with formal feature kinds based on references to identifiers. The results for ‘cmt’, on the other hand, provide very strong evidence that nonformal features may provide a novel perspective to entity description.

The third point confirms the validity of the experiment. Since the ‘all’ feature kind is the union of all other formal features, the fact that it covers them well (and that the converse is not true) confirms expectations.
and 'file' feature kinds seem to have the least redundancy. The 'cmt' redundancies with other feature kinds.

There is strong evidence for hypothesis 3 which with 'all' and also with other formal feature kinds. Hypothesis 1 was partly confirmed since most of the formal features have considerable redundancies among themselves; however, the fact that 'file' has a low redundancy was not expected.

### 7.3 Utility of nonformal feature kinds for clustering (hypothesis 2)

We hypothesised that:

**Hypothesis 2:** Nonformal feature kinds can be used for software remodularisation.

To verify this, we tested the resulting clusters using the design criterion (coupling and cohesion) and reference criterion (intra- and extra-pairs as compared to an 'expert' decomposition). Results are presented in Figs. 7 and 8 for the Mosaic system. In the figures, nonformal feature kinds are represented by dashed curves and the formal ones by plain curves. The dots show Bunch's results.

For the last graph in Fig. 7, the point at which a clustering approach would be 'ideal' is the lower right hand corner (high cohesion, low coupling). The curves evolve from the top of the graph (lower cuts at bottom of the hierarchy) towards the bottom of the graph (higher cuts at top of the hierarchy). For the last graph in Fig. 8, on the other hand, the 'ideal' point is the upper right corner (high recall and precision), and the curves evolve towards the top. We consider that only the last few cuts (except the very last) represent interesting partitions, with the earliest ones giving many very small or singleton clusters. The very last cut can safely be ignored for the opposite reason, since this is the stage where all entities are gathered into one big cluster which offers no interest.

From this experiment, it can be seen that 'Ident.' gives good results according to the design criterion, but average results according to the reference criterion. 'Cmt' has reasonably good reference results if a cut is made at the optimum height—it is clearly more sensitive to cut height than most other feature kinds. On the other hand, it has particularly poor design results. This is also seen when data for Linux are plotted (not shown).

It must be noted that although 'all' appears to have the best design results, this could be a consequence of the design criterion being based on this very feature kind. 'Ident.', which has high redundancy with 'all' (see preceding section) also gives good design results. On the opposite end, 'file' and 'cmt', which were the feature kinds with least redundancy with 'all', have the worst cohesion. These facts could be linked.

We also compared generated clusters to those of the reference decomposition using the 8-nearest neighbours.

### Table 7: Redundancy among feature kinds for gcc with the Jaccard association coefficient. Average number of commonalities among the 1-nearest neighbour

<table>
<thead>
<tr>
<th></th>
<th>var.</th>
<th>macro</th>
<th>rout.</th>
<th>file</th>
<th>type</th>
<th>all</th>
<th>cmt</th>
<th>ident.</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>var.</td>
<td>0.477</td>
<td>0.304</td>
<td>0.253</td>
<td>0.260</td>
<td>0.290</td>
<td>0.365</td>
<td>0.212</td>
<td>0.314</td>
<td>0.319</td>
</tr>
<tr>
<td>macro</td>
<td>0.317</td>
<td>0.454</td>
<td>0.293</td>
<td>0.308</td>
<td>0.308</td>
<td>0.366</td>
<td>0.220</td>
<td>0.328</td>
<td>0.323</td>
</tr>
<tr>
<td>rout.</td>
<td>0.367</td>
<td>0.284</td>
<td>0.309</td>
<td>0.291</td>
<td>0.303</td>
<td>0.375</td>
<td>0.220</td>
<td>0.329</td>
<td>0.339</td>
</tr>
<tr>
<td>file</td>
<td>0.134</td>
<td>0.125</td>
<td>0.137</td>
<td>0.129</td>
<td>0.128</td>
<td>0.135</td>
<td>0.112</td>
<td>0.140</td>
<td>0.161</td>
</tr>
<tr>
<td>type</td>
<td>0.301</td>
<td>0.263</td>
<td>0.262</td>
<td>0.258</td>
<td>0.430</td>
<td>0.277</td>
<td>0.177</td>
<td>0.230</td>
<td>0.275</td>
</tr>
<tr>
<td>all</td>
<td>0.453</td>
<td>0.488</td>
<td>0.547</td>
<td>0.435</td>
<td>0.943</td>
<td>0.252</td>
<td>0.413</td>
<td>0.498</td>
<td></td>
</tr>
<tr>
<td>cmt</td>
<td>0.231</td>
<td>0.228</td>
<td>0.222</td>
<td>0.217</td>
<td>0.227</td>
<td>0.467</td>
<td>0.233</td>
<td>0.295</td>
<td></td>
</tr>
<tr>
<td>ident.</td>
<td>0.297</td>
<td>0.323</td>
<td>0.288</td>
<td>0.291</td>
<td>0.280</td>
<td>0.300</td>
<td>0.206</td>
<td>0.476</td>
<td>0.309</td>
</tr>
</tbody>
</table>

We have no simple explanation for the last point.

The second experiment was done using k-nearest neighbours. Results are given in Table 7 for the Linux system; the closest pairs of entities were computed with the Jaccard association coefficient.

From the example data in Table 7, as well as from experiments with other systems and other similarity metrics, we conclude that:

- 'Cmt.' and 'file' have the least redundancy with other feature kinds (see last column in the table).
- The 'ident.' and 'all' feature kinds have the highest redundancies with other feature kinds.

These results are largely in accord with the ones from the previous experiment—especially the fact that the 'cmt' and 'file' feature kinds seem to have the least redundancy. Also, as before, 'ident.' has particularly high redundancy in this experiment.

In conclusion, the results are partly as expected and partly not. There is strong evidence for hypothesis 3 with regard to 'cmt.', but not with regard to 'ident'. We suggested an explanation why 'ident.' has a high redundancy with 'all' and also with other formal feature kinds. Hypothesis 1 was partly confirmed since most of the formal features have considerable redundancies among themselves; however, the fact that 'file' has a low redundancy was not expected.
To improve readability the first five and the very last cuts were ignored. System: Mosaic. Similarity metric: Jaccard coefficient. Algorithm: complete linkage (curves), Bunch (dots).

Fig. 7 Comparison of feature kinds with the design criterion
To improve readability the first five and the very last cuts were ignored. System: Mosaic. Similarity metric: Jaccard coefficient. Algorithm: complete linkage (curves), Bunch (dots).

7.4 Sibling/direct link approaches (hypothesis 5)
The hypothesis regarding link types is:

Hypothesis 5: The sibling link approach does not perform worse than the direct link approach.

This hypothesis is difficult to test, since each approach is used by different algorithms: hierarchical algorithms for the sibling link approach, and a hill-climbing algorithm (Bunch) for the direct link approach. This is why we expressed the hypothesis in a negative form.

To test the hypothesis, we primarily use the reference and design criteria. The results for the sibling link approach with the Mosaic system can be found in Figs. 7 and 8. The experiment for Bunch's direct link approach are shown as small black circles (rightmost graphs).

When simply comparing the quality of results for the two approaches, there does not seem to be any advantage from either. However, we do see an advantage in the sibling link approach, in that it allows for a wider range of feature kinds (e.g. nonformal features; see Observation 5).

The experiment provides good evidence to support hypothesis 2.

7.5 Similarity metrics and zero-dimensions (hypotheses 6 and 7)
We formulated two hypotheses related to the way similarity metrics deal with zero-dimensions:

Hypothesis 6: Similarity metrics that consider zero-dimensions as a sign of similarity tend to cluster all entities together in a huge meaningless cluster.

Hypothesis 7: Feature kinds with fewer dimensions are less subject to the problem identified in the previous hypothesis regarding zero-dimensions.

Fig. 10 uses the size criterion to compare the results from three similarity coefficients. Note that, due to the use of different similarity metrics, the x-axes differ (see also Section 5.3). One could normalise the cut height by expressing it as a fraction of the maximum height for a given hierarchy. 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 coefficient (not shown), which do not consider zero-dimensions, give good results, i.e. a planetary system configuration. The Simple Matching association coefficient (not shown) and the two distances (such as taxonomic distance, shown 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), which includes zero-dimensions but is not...
really affected by them (see explanation in Section 5.1) also gives good results.

The results for the design and the reference criteria are similar: Jaccard and Sørensen-Dice give the best results, correlation follows closely and the three others (Simple Matching, taxonomic distance and Canberra distance) give poor results.

To further establish the role of null-dimensions in the bad results obtained with some similarity metrics (e.g. taxonomic distance in Fig. 10), we also compared feature kinds with different propensities for null-dimensions. As already mentioned in Section 5.1, the 'File' feature kind, because it has fewer dimensions, tends to suffer less from the null-dimension problem. As a consequence, it gives slightly better results, even with a similarity metric that does consider null-dimensions as a sign of similarity. See Fig. 11 and also the third graph in Fig. 10 for an example of this.

The above discussion shows that there is strong evidence confirming hypotheses 6 and 7.

7.6 Comparison of clustering algorithms (hypothesis 8)

We formulated the hypothesis that:

Hypothesis 8: Single linkage results in clusters that are less coupled, while complete linkage results in clusters that are more cohesive.

The results of design criterion experiments shown in Fig. 12 provide very strong evidence in support of this hypothesis. The leftmost graph shows that complete linkage has the best (highest) cohesion, and the middle graph shows that single linkage has the best (lowest) coupling. The rightmost graph presents cohesion and coupling together, allowing better comparison of each algorithm’s performance. The ideal point is the lower right corner (high cohesion, low coupling).

The graph shows that unweighted linkage offers the best compromise. However, remember that the design criterion could be biased towards this algorithm.

The same conclusions apply with the reference criterion. We can also note that single linkage has a tendency to gather all the entities into one cluster, resulting in a black hole configuration (see Fig. 5). Overall, there is a gradation from complete linkage, to weighted, unweighted and single linkage for the design, reference and size criteria.

8 Practical tips

In this subsection, we try to summarise some of the results to help practitioners actually use hierarchical clustering for software remodularisation.

From our experiments, it can be seen that hierarchical clustering provides results that are as good as other algorithms. The advantage of the hierarchical approach is that it can be used to obtain different partitions of the system at different levels of abstraction. This might, for example, prove useful when developing a browser. On the other hand, one must choose the appropriate height at which to cut. We address this issue below.

As to the choice of algorithm, we suggest using 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 [9]), but we saw that 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.
We recommend taking the results of a hierarchical algorithms and presenting the user with several different cuts, at different levels of abstraction. However, if one single cut is desired, then choosing it can pose a problem. Maurek et al. propose a simple heuristic to do so in [20], but her proposal does not fit the experimental results we obtained. Designing an algorithm to find the ‘optimal’ height to cut the hierarchy of clusters could be a good research topic, but one would first need to define the criteria on which to base the optimisation.

The lower cuts are not normally useful because few clusters are created and they are too small to provide valuable abstractions. We suggest not to cut below half of the maximal height. It is advisable neither to cut at the maximum height nor too close to it, because at these levels the clustering algorithms start to gather all entities together, so the clusters are too big to be of interest. One can usually notice a sudden drop in the number of entities in intermediate clusters. This marks the highest point of interest in the hierarchy.

Finally, we will mention some conclusions that came out as a result of our experiments and that could deserve more attention:

- ‘Var.’ (references to global variables) gives poor results for Mosaic and Linux, average results for gcc and good results for the telecommunication system. The implication is that its results are highly system-dependent. The positive results for the telecommunication system may be related to the fact that it relies heavily on global variables in its design, as shown in Tables 3 and 5.
- ‘Rout.’ (routine calls) generally performed poorly (except for gcc) in both the design and reference criteria. This surprised us since routine calls is one of the most natural feature kinds to use when clustering.
- ‘File’ (file inclusion) gives clusters that match our reference subsystems (i.e. good results for the reference criterion). This could be a result of the experimental conditions since these partitions were based on the directory hierarchy. This feature kind is also language dependent; some languages (e.g. C) use it much more than others (e.g. Pascal).

9 Future work

As mentioned in the introduction, clustering is a sophisticated research domain, with many alternative methods and numerous influencing factors. It was not possible to deal with all the issues in a single paper and many more studies need to be conducted.

The following are some of the research questions raised in this paper, in which more research is needed:

- To what extent can software engineers actually use clustering techniques to perform remodularisation in practice? We have presented experimental evidence showing how to use the techniques effectively, and showing that they can produce sets of modules that appear to be useful. But it is important to give tools to software engineers who actually implement the techniques, and then study how the software engineers make use of them. Currently there are many unanswered questions about what software engineers really need; hence we agree with Clayton et al. [1], who say that the goals of reverse engineering researchers remain somewhat unclear.
- How valuable are the ‘batch clustering’ techniques discussed in this paper, as compared to ‘incremental’ approaches that improve modularisation by moving around a few elements instead of creating a new...
remodularisation from scratch [46]? The incremental approaches have become the focus of much recent research; they include techniques such as 'orphan adoption' [4] and 'maverick analysis' [27]. We believe that for systems that are very disorganised, a complete remodularisation may be warranted; but where is the dividing line at which incremental techniques improve?

- To what extent can software engineers make effective use of clustering techniques for generating different views or interpretations of a system, as opposed to actually remodularising it? We hypothesise that software engineers might be able to make use of such views to more rapidly become acquainted with a software system, but this hypothesis needs testing.

- Using the evaluation criteria we defined, how do nonhierarchical algorithms compare to the hierarchical ones? We performed some experiments with the Bunch tool, but further work is needed.

- How useful are nonformal feature kinds when clustering? We studied two of them, but this is fertile ground for additional research.

- Is there any value in pre-filtering the data? One problem with the formal feature kinds (and to a lesser extent the nonformal ones) is the noise they contain. One could try to use information theoretical analyses to detect and eliminate redundant features, or features providing very little information. Note, however, that this is a complex issue. A previous study [15] showed that in some cases noise can actually be a good thing and lead to the discovery of additional concepts.

- How valid are the evaluation criteria, and should other evaluation criteria also be used? We mentioned in the paper that there are few well established criteria to evaluate the results of clustering (observation 7), and used several criteria in our analysis. Other researchers [Note 16] have suggested additional criteria such as: the variance of the cluster hierarchy depth (to analyse gas cloud, black hole and planetary configurations); using information theory to compare features (as opposed to feature kinds); and using the K-nearest neighbour approach to compare generated clusters to a validated reference clustering ([47, 48]). It would be interesting to compare the usefulness of the different evaluation techniques.

10 Conclusion

In this paper, we presented an analysis of three parameters that may influence the clustering results when doing software remodularisation with agglomerative hierarchical clustering algorithms. The three parameters are: how the entities are described; how coupling between the entities is computed; and what algorithm is used.

During our analysis, we formulated a series of hypotheses. Our experiments then provided strong evidence, largely confirming most of the hypotheses. The hypotheses are as follows:

1. There are redundancies among all the formal feature kinds extracted from the source code. There was some evidence in favour of this; but the 'file' (file inclusion) feature kind was an exception, showing low redundancy.

2. Nonformal feature kinds can be used for software remodularisation. There was good evidence for this hypothesis; both nonformal feature kinds we used (words in comments and in identifiers) gave good results. This agrees with our previous research.

3. Nonformal feature kinds have less redundancy with formal ones than the formal ones among themselves. There was evidence supporting this for the 'cmt' feature kind, which had one of the lowest levels of redundancy. On the other hand, 'ident.' behaved like the formal features. We can conclude that nonformal feature kinds like 'cmt' can provide a novel way of looking at a system, but this independence cannot be taken for granted for all nonformal feature kinds.

4. Nonformal feature kinds provide more information, on average, than the formal ones. There was strong evidence for this.

5. The sibling link approach does not perform worse than the direct link approach. This appears true; in addition, due to the wider range of application of sibling links, we suggest to use this approach.

6. Similarity metrics that consider zero-dimensions as a sign of similarity will tend to cluster all entities together in a huge meaningless cluster. Our data clearly support this hypothesis. This is an important conclusion which should be taken into account when doing clustering.

7. Feature kinds with fewer dimensions are less subject to the problem identified in the previous hypothesis regarding zero-dimensions. The data from our experiments also support this; however, one should still use similarity metrics which do not consider zero dimensions as a sign of similarity.

8. Single linkage results in clusters that are less coupled, while complete linkage results in clusters that are more cohesive. This was also supported by our data. We conclude that the best choice of algorithm should be either complete linkage, which favours good cohesion, or else unweighted average linkage, which gives the best compromise between cohesion and coupling.

We experimented only with file clustering. Other tests should be done on clustering different entities (routines, classes, processes, etc.) We predict that the results would be similar when the same quality criteria are used. For example, the problem of quasi-empty descriptions would be the same whether entities were routines or classes. Note, however, that when clustering other entities, the purpose of the clustering may also change. For example, in [12], variables are clustered to discover possible classes in procedural code. Although we believe that the criteria we used are still valid in this context, other criteria should be considered such as the possibility that a single entity may appear in more than one cluster.

11 Acknowledgments

The authors wish to express their thanks to other members of the Consortium for Software Engineering Research [Note 17] 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 Book-shelf base of facts for the Linux system (http://plg.uwaterloo.ca/~heitz/guinea-pig/). We thank the reviewers for their valuable input. This work was supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER). Finally, we also wish to thank K. Teresa Khidir for editing the paper and improving its readability.


Note 17: see: http://www.cserv.ca
References


7. Much project (Software Engineering Research Group at Drexel University), http://segr.drexel.edu, accessed on 10 June 2003


16. ANQUETIL, N.: 'A comparison of graphs of concept for reverse engineering'. Proc. 8th Int. Workshop on Program comprehension (IWPC), Limerick, Ireland, 2000, pp. 231-240


24. ANQUETIL, N., and VAUCHER, J.: 'Meta-knowledge for the object model: Sample us NOT'. Presented at Metamodeling in OW Workshop (OOPSLA), Austin, TX, 1995


44. NCSA Mosaic Version 2.6. Available via anonymous ftp at ftp.ncsa.uiuc.edu, in /Mosaic/unix/source


