Christos Tjortjis

Data Mining Code Clustering (DMCC): An Approach Supporting Software Maintainers with Program Comprehension

School of Science & Technology, International Hellenic University

14th km Thessaloniki – Moudania, 57001 Thermi, Greece

Tel: +30 2310807576

Fax: +30-2310474590

Email: c.tjortjis@ihu.edu.gr

ORCID 0000-0001-8263-9024

Abstract

Software maintainers face challenges when making decisions to modify programs with little understanding of the overall source code organisation and the full impact of changes. Most software systems are structured as a number of subsystems, consisting of code that collaborates to provide the composed functionality to the program. An important aspect of program understanding is to perceive this subsystem structure. Cluster Analysis can be of use in deriving a meaningful subsystem structure of a program from its source code. The idea is to represent a program as a number of entities, which are grouped in clusters representing subsystems, based on their similarity measured by means of related functionality or data use.

Central issues for any clustering-based approach are the specification of program entities and their attributes, similarity metrics, and clustering strategy. We propose here Data Mining Code Clustering (DMCC), an approach for supporting software maintenance and program comprehension that uses input data extracted from a C/C++ program and produces its abstraction as a number of subsystems. New possibilities in the form of weighting rationales and a framework for similarity metrics based on these, as well as an analysis of these from a novel point of view is a main contribution. The approach was evaluated by implementing a tool that was used for experimentation with programs of various sizes and languages, in collaboration with experts. Results showed that the approach is useful for deriving accurate subsystem abstractions and identifying interrelationships amongst modules. Factors influencing the feasibility of this approach are identified and directions for improvements are discussed.

Keywords*: Software Management; Program Comprehension and Cognition; Code Mining; Clustering.*

Abbreviations: Data Mining Code Clustering (DMCC), Identification of Subsystems based on Associations (ISA), Module Dependency Graph (MDG), Modularization Quality (MQ), User-defined type Global Variables (UGV), Predefined type Global Variables (PGV), Local Variables (LV), Performance Index (PI)

1. INTRODUCTION

Software systems evolve during their lifecycle, in order to conform to increasing user needs and the ever-changing legal, technological, and business environment. Attaining and retaining high levels of software quality requires continuous maintenance, comprising updates, enhancements and upgrades. Program comprehension and cognition is crucial during maintenance, especially in cases where documentation is poor or outdated and program structure is complex. Data mining techniques can assist program comprehension and decision making related to maintenance by producing structural views of legacy systems when source code is the only reliable information source.

This paper presents and evaluates Data Mining Code Clustering (DMCC), an approach for clustering source code to support program comprehension and facilitate software maintenance. It comprises an input data model, a set of similarity metrics and a clustering algorithm. It builds upon work on deriving high-level subsystem abstractions of a program at a coarse level, where subsystems are formed by collections of files. Work presented here achieves that by deriving medium/low level subsystem abstractions of a program at a finer level, where subsystems are formed by collections of functions. DMCC resulted in high precision and recall, successfully separating subsystems when mixed up and producing overviews which capture rather precisely the original developers' mental model of the program. However, it is limited with respect to the amount of time needed to produce the data inputs.

The remainder of this section discusses software maintainers' problems and program comprehension issues. Section 2 presents background on data mining and clustering methods used for program comprehension. Sections 3 and 4 describe the objectives and requirements of the proposed approach, including the data model, similarity metrics and clustering strategy. Section 5 presents and evaluates experimental results. Section 7 concludes the paper with directions for further work.

1.1 Software Maintainers problems

The maintenance stage of software lifecycle typically consumes 50-70% of the total effort allocated to a software system [1], [2]. During maintenance, modifications are made to the source code in order to adapt a software system's functionality and comply with evolving user needs. The increased time, effort and resource levels allocated to software maintenance are attributed to the following [2], [3]:

1. The program being maintained may be poorly structured with little regard to understandability.
2. The original documentation explaining the structure of the software may not be available or it may be outdated.
3. Due to the high staff turnover rate in the software industry, the original developers of the software may not be available to perform the modification. This often results in modifications being performed by staff with little knowledge of the structure and organisation of the source code.
4. Modifications made to the source code may produce new faults that may trigger more maintenance requests from the customer/ end user.

Problems arise because maintenance is usually performed by staff with little understanding of the overall organisation of the program, employing the *fix-it-quick* approach [2]. Hence, modifications are made inconsistently with the original structure of the program. This occurs because the program is too complex to comprehend, and the effect of modifications is not fully understood by maintainers.

1.2 The importance of program comprehension

Program comprehension is of considerable importance for software maintenance, especially when there is incomplete documentation and the source code is the only information resource available to maintainers. 50%-90% of the maintainers' time was reported to be spent on program comprehension, particularly when legacy programming languages are used [4]. Several detailed theories about the

subject have been created. For example, Brooks [5], Soloway & Ehrlich [6], and Letovsky [7] examine in detail the mental processes behind program comprehension.

A theory which is relatively elementary and widely applicable to software maintenance is that of Littman *et al.* [8], who suggest that, in order to successfully modify a program, a maintainer must accomplish two types of knowledge: *static knowledge* and *causal knowledge*. The former is concerned with the subsystem structure formed by program components. The latter is concerned with interactions and data flows between program components. This view of grouping program components into subsystems that provide a common service to the overall program is widely accepted as a key stage in program understanding.

Lakhotia [9] argued that such an abstraction of a software system is of immense significance in software maintenance activities, because it helps maintainers to infer interactions between subsystems. Lakhotia believes this knowledge helps the maintainer to understand the full impact of modifications to the source code.

Kunz and Black [10] argued that grouping program components into subsystems reduces the perceived complexity. They believe such a view helps maintainers predict the full impact of making modifications to the source code. Tzerpos and Holt [11] conjecture that deriving a decomposition of software systems into a set of meaningful subsystems alleviates much of the effort required to understand a software system. Anquetil and Lethbridge [12] believe that a technique to help maintainers understand a software system would be to gather program components into modules that have a common significance.

2. BACKGROUND

Software systems typically consist of a collection of programs. In the vast majority of these programs, the source code is organised into some degree of modularity and is composed of a hierarchy of subsystems, each of which provides some service to the overall program or performs a particular subtask [13]. Subsystems may collaborate to provide a higher-level functionality to the program. A subsystem is a collection of program modules at various levels. Although hidden from the user, all software systems have a hierarchical structure containing subsystems of different levels of granularity, ranging from subsystems that perform primitive subtasks to subsystems that provide a higher level functionality, typically consisting of smaller subsystems. The number of such subsystems and the complexity of the subsystem hierarchy will generally increase with the size of the software system.

Many previous approaches to deriving a subsystem view of a program have been at a *coarser* level, where the program consisted of numerous files, and the subsystems are formed by collections of files. Here, programs were not analysed beyond the level of complete files, therefore representing a very high-level subsystem abstraction of a program. Examples include [11], [12], [14], [15], [16], and [17].

De Oca and Carver [15] argued that data mining can be used to identify subsystems within a software system. They developed the ISA (Identification of Subsystems based on Associations) methodology the objective of which is to decompose a software system into *data-cohesive subsystems*. A data-cohesive subsystem consists of programs that use the same persistent data files. Therefore, they used

complete programs as entities comprising the subsystems. They describe a general 3-step methodology for applying Data Mining to such design recovery: Create a database view of the system; perform Data Mining; interpret the results. The ISA methodology was applied to several COBOL systems. It was found that there are programs that cannot be assigned to a subsystem; however, the number of such programs was small compared to the total number of programs. It was also found that some files are used across subsystems. Such files were interpreted as forming interfaces between subsystems.

Mancoridis et al. [16] studied the Module Dependency Graph (MDG) approach for dividing a software system into meaningful subsystems, using files or programs as entities to produce a graphical representation of a system as a set of nodes connected by edges. They used the concepts of *coupling* and *cohesion* to specify three parameters: Intraconnectivity, Interconnectivity and Modularization Quality (MQ). The objective of this work was to use clustering algorithms to create a subsystem view of the software by partitioning the MDG with the aim to achieve the highest MQ. The methodology was successfully applied to several systems of varying sizes. The methodology was expanded in a later study to introduce a facility to detect and isolate omnipresent nodes, i.e. files that do not tend to belong to any particular subsystem, and to allow expert knowledge to influence the clustering [18].

The MDG approach is different from the clustering approach because the similarity between entities is not measured using explicit attributes. In addition, similarity between entities is not calculated by comparison of attributes. Rather, the concepts of coupling and cohesion are used to determine entities to cluster together. The MDG is divided into clusters in such a way that entities in the same cluster have many dependencies (high cohesion) and those in distinct clusters have minimal dependencies (low coupling). The best clustering is one that gives highest cohesion and lowest coupling.

The methodology aids a coarse level of software understanding, at the level of complete component programs. It would have to be radically modified to abstract programs into subsystems. It has also limited accuracy, because in the MDG, an edge represents some dependency between two entities without considering the number or the significance of dependencies.

Anquetil and Lethbridge [12] proposed hierarchical clustering as a software remodularization method for gathering program components into modules that have a common significance. They identified files, routines or processes as possible entities. However, the large scale of their subject system led them to use complete programs and files as entities. To describe these entities, they distinguished between two types of features to be derived from the source code: formal features and descriptive features. They identified a problem of redundancy when describing files using these attributes. Later on, they proposed two approaches to similarity metrics for clustering software systems using programs and files as entities: the direct link approach and the sibling link approach [12]. These approaches were based on the MDG of the software system. They evaluated the quality of clustering using the *coupling* and *cohesion* criterion and the *precision* and *recall* criterion.

Tzerpos and Holt [11] also used files as entities to partition large software systems into a series of subsystems in an effort to get better insight into such systems. Another approach evaluating dynamic clustering was presented by Xiao and Tzerpos [19]. Its scope was to evaluate the usefulness of

dynamic dependencies as input to clustering algorithms. The method consists of three phases. The first is the analysis of dynamic dependencies by adding instrumentations when compiling the source code. The second is the analysis of static dependencies. The last step is filtering in order to help weigh the dynamic dependencies graphs. The work concluded that there is merit in clustering dynamic dependencies of a software system.

Sartipi et al. [20] used data mining for architectural design recovery. They proposed a model for the evaluation of the architectural design of a system based on associations among system components and used system modularity measurement as an indication of design quality and its decomposition into subsystems. Three association views of a system were generated: a) control passing which represents system components correlation based on function invocation, b) data exchange which represents system components correlation based on aggregate data types and c) data sharing which represents system components correlation based on functions sharing global variables. This approach models software systems as attributed relational graphs with system entities as nodes and data-control-dependencies as edges. Application of association rules mining decomposes such graphs into domains of entities based on the association property.

Clustering has also been used to support software maintenance and systems knowledge discovery. A method for grouping Java code elements together according to their similarity was proposed in [21]. It focuses on achieving a high-level system understanding. The method derives system structure and interrelationships, as well as similarities among systems components, by applying cluster analysis on data extracted from source code. Hierarchical agglomerative clustering was employed to reveal similarities between classes and other code elements, thus facilitating software maintenance and Java program comprehension. A similar approach, using K-means clustering was successfully applied to code extracted from C++ programs [22], while association rules were used to capture COBOL program structure thus achieving better system understanding [23]. K-means clustering combined with an improved version of MMS Apriori association rules mining was proposed for producing system overviews and deductions, and identifying hidden relationships between classes, methods and member data in C# code [24]. Also, clustering classes, followed by classification of extracted clusters were recently proposed, in order to assess internal software quality, using Java classes as entities and static metrics as attributes [25].

Other more recent approaches use a variety of data mining methods. For instance association rule mining was applied to the problem of understanding a software system given only the source code [26]. Classification was used as a means for static code analysis of C for the timely identification of software bugs as well as for locating software defects [27], [28]. Software clustering approaches cluster large software systems based on the static or even dynamic dependencies between software artifacts [14].

It follows that the understanding of programs by producing a finer subsystem abstraction is a maturing research area. This study focuses on the problem of understanding a program at this finer granularity.

2.1 Clustering

There are various clustering techniques and algorithms. This subsection briefly describes the most prominent ones, including partitional, density based and hierarchical algorithms.

Partitional/Optimisation algorithms start with an initial partitioning of the data and then modify this partitioning in an attempt to reach an optimum clustering based on some criterion [29], [30]. The final clustering obtained is largely dependent on the initial partitioning. The drawback of these types of algorithms is that the many possible initial partitions combined with subsequent partitioning can lead to very many possibilities. In addition, the end result is a single cluster distribution. To modify the distribution, it is required to re-run the procedure with a different initial partition.

Density search clustering algorithms assign entities to clusters in a single step [29], [30]. These differ from optimisation algorithms in that once an entity is placed in a cluster, it is not relocated. The user may specify the number of clusters to be formed or this may be left to depend on the clustering process. The advantage of these methods is that they can be easily visualised and validated. A drawback occurs when more than three attributes are of importance, in which case more advanced tools must be used, such as OLAP [31]. These are particularly suitable when the objects to be clustered have numerical, continuous attributes but are not be meaningful for those with qualitative or binary attributes. Representation of such entities in such a graphical manner will have poor information content.

Some clustering algorithms require to measure the distance among clusters rather than items, and merge the clusters with the least such distance. This can be done using the single linkage, the complete linkage and the average linkage. *Single* linkage (or nearest neighbour) calculates the minimum distance between any two items which belong to different clusters. *Complete* linkage (or furthest neighbour) uses the maximum distance between any two items which belong to different clusters. *Average* linkage uses the average distance between an item from one cluster and all items from another[1]. For example, if A, B and C are clusters, after B and C are joined, one wishes to determine the similarity between A and B∪C, that is *Sim(A,B∪C).* Assuming the similarity between A and both of B and C is known, the *single linkage rule* would set the new similarity as the minimum of *Sim(A,B)* and *Sim(A,C)*. The *complete linkage* rule would set the new similarity as the maximum of these. The *weighted linkage rule* would calculate the new similarity as a weighted average of these, depending on the number of objects in clusters B and C, or some other criteria such as the size or importance of objects in each cluster [29]. The *unweighted (*or *average*) *linkage rule* would calculate the new similarity as the average of these similarities. Single linkage is known to favour non compact but more isolated clusters whereas, complete linkage usually results in more compact but less isolated clusters; average linkage stands "in between".

Given a large set of multidimensional data points, clustering identifies the sparse and the crowded places, and hence discovers the overall distribution patterns of the data set. Descriptions are then derived for the constructed clusters. There are three main types of clustering algorithms: *hierarchical*, *partitional* and *density search* clustering algorithms.

---

[1] This is the *unweighted* version of average linkage. The weighted version uses each cluster's size as a weigh attribute when calculating the distance.

Hierarchical algorithms partition data into clusters through a series of partitions, as opposed to a single step [29], [30]. The partitions range from a single cluster containing all the objects to *n* clusters each containing a single object. This class of algorithms is divided into two subclasses: *Agglomerative* and *Divisive* algorithms. The former start with *n* clusters, each containing a single object, and join the most similar clusters, until some pre-specified point is reached. The latter start with one large cluster containing all the objects and split this into smaller clusters until some pre-specified point is reached. In cases, joining and splitting of clusters is based on some similarity metric. An advantage of this type of algorithm is that a hierarchy of clusters is formed, and the most suitable level of clustering may be selected from this, one which most closely resembles an 'expert' partition.

3. OBJECTIVES

This work was concerned with applying cluster analysis to derive a meaningful subsystem structure of a program from its source code. A challenge of this work was to adapt clustering techniques to the peculiarities present in the application domain of a source code.

The main objectives of this work are as follows.

1. Specification of the *input-data model* to be provided to the clustering algorithm. This concerns the specification of program entities and their attributes. It is these entities that are to be grouped into subsystems. Program components to be used as entities are constructs within the source code. In addition, program entities must have several attributes that provide a basis for measuring similarity between entities. An aspect of this application domain is that attributes are not numerical and continuous. Rather, attributes are qualitative and binary.
2. Specification of *similarity metrics* with which to determine the similarity between program entities. Existing similarity metrics must be examined for use with binary, qualitative attributes and a suitable one selected, and tailored for the peculiarities of this domain.
3. Specification of the *clustering algorithm* to be used to meaningfully group program entities into subsystems (clusters) based on similarity. The quality of output from the clustering strategy depends largely on the soundness of the input, which is determined from the previous steps.
4. Implementation of a tool, which takes as input data extracted from a program's source code and produces an abstraction of the program as a number of subsystems. Results are then to be *evaluated* by comparison with experts' mental model of the program. The *feasibility* of this approach to obtaining a subsystem abstraction of a program is also to be evaluated. If such a tool is to be used on a wider scale, it should provide a meaningful subsystem abstraction of a program in less time and with less effort than it would take to obtain a similar abstraction by mere inspection.

An important point to note is that this work is concerned with devising an approach and developing a tool for semi-automatic support to program understanding. This means that it is assumed that the user has no expert knowledge of the program to be analysed. Therefore, the input data model provided to the tool must be based on textual features of the program, thus requiring no expert knowledge to acquire.

4. RESEARCH REQUIREMENTS AND APPROACH

In order to design a source code clustering approach certain major issues need to be dealt with, including the selection of programming language and corresponding input data model, selection and customisation of similarity metrics and the formulation of a clustering strategy. These are detailed in the following subsections.

4.1 Selecting a programming language

Clustering source code is programming language dependent task. Any approach that may be applicable to a certain language needs to be modified in order to be applied to another language. Several programming languages have been studied in the past, with regards to clustering. Examples include procedural languages like COBOL [23], as well as object oriented ones such as C++ [22], C# [24] and Java [21], [25].

This work focuses on C, a widespread language, more complex in grammar and structure than COBOL, yet less sophisticated compared to object-oriented languages equipped with many advanced features such as inheritance, polymorphism etc. In fact, this work includes experiments with C and C++ programs in an attempt to explore the applicability of the proposed approach to a more modern and advanced language such as C++.

4.2 Input data model

One of the challenges of this work was the definition of an input data model, which could be derived from C program code and fed to a clustering algorithm. Clustering is a statistical data analysis technique usually applied to conventional databases for data mining. As a result, useful and meaningful entities consisting of the relevant attributes needed to be defined. Preliminary analysis has drawn the following requirements for program entities and attributes that could lead to satisfactory performance of the approach.

1. Entities should be homogeneous, thus allowing for description by a common set of attributes. This is needed for entity comparison based on their attributes: the basis of cluster analysis which conforms to the premise that objects of different ‘species’ should not be compared.
2. Entities must have a enough attributes to provide an informative description and means for comparison. A very small number of attributes could lead to crude entity description, thus providing very little information content and possibly leading to unreliable conclusions.
3. Entities, attributes and attribute values must be clearly defined, for any given C/C++ program. Furthermore, the definition of an entity should be universally applicable to C/C++ programs for the approach to have broad utility.
4. Entity selection should allow for a large proportion of a program’s code to be associated with an entity, when such a program is abstracted as a collection of entities. This ensures that the proposed analysis covers a large proportion of a program.

4.2.1 Deriving entities

Several candidates were examined to be selected for program entities, namely individual statements, functions, classes and sequence, selection or repetition constructs.

The problem with *individual statements* is that they present a minimal set of descriptive features. They are so diverse, that trying to derive a set of attributes which are applicable to any possible statement would result in a great number of null values. Furthermore, many statements occurring in a program do not have a meaning in isolation. For example, the statements representing the "if … else" construct only make sense when viewed collectively. In a similar manner, *sequence*, *selection* and *repetition* constructs provide one with a limited set of descriptive features and impose the difficulty of tracing the beginning and the end of a sequence.

Functions were considered to be the best option. They tend to encapsulate a single functionality, performing a cohesive specialised task. They are clearly defined textually. And they may provide one with a reasonably sized set of attributes. A limitation here is the possibility of neglecting parts of code, which do not belong to any function.

4.2.2 Deriving attributes

The next step was to derive a range of attributes with which to describe program entities. The attributes had to be such that they can be evaluated for each entity through the textual features of the program, thus requiring neither expert knowledge nor sophisticated parsers. Five different types of attributes were specified with which to describe program entities, and a specification given as to how the complete set of attributes can be derived from a program. These are:

a) Use of global and static variables
b) Use of local variables
c) Local variable types
d) Formal parameters types
e) Returned values types

The rationale behind these attribute selections is given in the following:

The first attribute was based on the use of *global* and *static* variables. They were both selected to form a single attribute although they could be broken further down, for example depending on whether the use involves value modification or not. Increasing the level of detail would lead to a more specific description of entities but could lead to a less effective cluster analysis. This is because, as the attributes become more specific, the likelihood of two entities scoring on a common attribute decreases, thus reducing the quality of information fed into the clustering step.

For a given program, the set S_g of all attributes based on the use of global variables is given by (1):

$$S_g = \{\forall g \in G | Uses\ global\ variable\ g\}. \quad (1)$$

G is the set of all global and static variables in the program.

The utility of using a global variable as an attribute for clustering depends on its nature, which is largely influenced by its *type*. More effective global variables are likely to belong to a user-defined type. This is because most user-defined types are specified to model a particular aspect of the

problem domain; functions that use these are likely to be associated with this aspect of the problem domain, indicating membership of a common subsystem. Thus, the differentiation on whether the type of attributes is *user defined* or *predefined* was considered to be necessary and will be also used for other types of attributes, where applicable.

Another attribute was based on the use of local variables. By definition, a function's local variables cannot be meaningfully used within the body of another function. Attributes based on the usage of static variables then have similar advantages and drawbacks to attributes based on global variables. An automatic local variable can only be used by another function if this other function is called by the former function with the local variable as an actual parameter.

For a given program, the set S_l of all attributes based on the use of local variables is given by (2):

$$S_l = \{\forall l \in L | Uses\ local\ variable\ l\}. \quad (2)$$

L is the set of all local variables declared in a program.

An advantage of this type of attribute is that it introduces a dynamic element into the cluster analysis. This is because it is based on interactions between functions, through their local variables. In addition, when analysing larger programs, such an attribute can be easily detected by automatically locating the occurrences of each function identifier outside of its own header(s). If an occurrence is found, then the function containing the reference scores positively on the attributes based on the usage of each actual variable.

The local variables used by a function will be either its own local variable or that of another function. A function typically will use its own local variables, meaning that its attributes will match those of other functions that use its local variables. Although this may seem unnatural, it is necessary to aid detection of a relationship between two functions through these attributes. This way the parent function of the local variable and the function that uses the local variable will score positively on the attribute based on the usage of the local variable, thus correctly rendering a similarity between them.

An attribute that is directly derived from the description of a function is that which is based on the type of its local variables. It is hypothesised that the overall purpose of a function, hence its parent subsystem, can be predicted by examining the nature of the data items on which it operates. The nature of the data items can be determined by their type. It follows that functions having in common the types of several variables are likely to have a similar purpose, hence may belong to the same subsystem.

For a given program, the set of binary attributes S_t based on the types of local variables is given by (3):

$$S_t = \{\forall t \in T | Has\ local\ variable\ of\ type\ t\}. \quad (3)$$

T is the set of all data types (user-defined and pre-defined) used in a program.

The use of attributes based on the types of formal parameters has the same justification as above. The type of the data operated upon is likely to be an indication of the function's parent subsystem. However, this type of attribute cannot be useful for functions without any formal parameters.

In a similar manner, for a given program, the set of all attributes based on the types of formal parameters *F*, is given by (4):

$$F = \{\forall t \in T | Has\ formal\ parameter\ of\ type\ t\}. \quad (4)$$

The use of function attributes based on the *types of returned values* has also the same justification as above. The type of the data operated upon is likely to be an indication of the parent subsystem of a function. However, this type of attribute will not be useful for void functions.

In a similar manner, for a given program, the set of all attributes based on the types of returned values *R*, is given by (5):

$$R = \{\forall t \in T | Returns\ a\ value\ of\ type\ t\}\ . \quad (5)$$

Additional information required for capturing interrelationships amongst attributes in the input model is covered by three general principles: the Significance Principle, Usage Principle and Relationship Principle. These principles are concerned with using the specified attributes to allow a more detailed description of the relationship between program entities, which will be exploited by the similarity metrics. It should be noted that all five classes of attributes identified are essentially based on the usage and structure of variables (variables and constants, global, local, formal, and returned) present in a program. It follows that these principles can be applied globally to all five classes of attributes.

The *Significance Principle* concerns the relative importance amongst attributes and depends on the type of the variable on which an attribute is based. Presumably a scale of importance governs the significance of attributes. However, the exact difference in significance between different types of attributes is not known; and should logically depend on the individual characteristics of the program (and may even be subjective), thus requiring expert knowledge to deduce. However, a basic way of constructing such a scale for user-defined types would be to place greater importance on more specialised or complex types. The explanation is that user-defined types are specified to model a more specialised aspect of the application domain, thus the use of such a variable is more indicative of a function's purpose. Such a scale could be implemented in an automated approach, as the complexity would be based on a count of the total number of data members belonging to a user-defined type. However, it is not possible to automatically deduce the relative importance between user-defined types having the same number of data members, as this would require expert knowledge.

The *Usage Principle* also concerns the relative importance amongst attributes but is based on the frequency-of-use of the variables on which the attributes are based. Some of the variables on which the attributes are based will be used more times in a program than others.

In the context of the *Relationship Principle*, the relationship between attributes is affected by their *types*. The types of local variables, global variables, returned values and formal parameters, may be either predefined or user-defined. Possible predefined types include *int*, *double*, *char* and *boolean*. The user may also apply the *typedef* operation to an elementary type, thus creating a new type identifier. From these elementary types it is possible to construct the more complex types: pointer types (such as *int**) and reference types (such as *double&*). In turn, from these types the user may create more complex types: *array* and *enumeration*. Finally, the user may create even more complex

types by defining data structures consisting of two or more data members. These data members may belong to any of the above types, or they may themselves be data structures.

An important point to note is that a type may be more closely related to some types than others. For example, the type *int* is more similar to *float* than *char*, or the type *double* is more closely related to the *float* than *int*. Such relationships may also occur between user-defined types. When considering an automatic approach to program clustering, without the need for a user's expert knowledge, the relationships between user-defined types will fall into two main categories: automatically detectable and non-automatically detectable.

Automatically detectable relationships can be detected through an automatic approach which does not require expert knowledge. This concerns the type *enum* and the type *struct*. Variables belonging to an *enum* type may be logically deduced to be related through the textual properties of a program, thus easily incorporated into this approach. Variables belonging to types used to define data members of a *struct* type may also be said to be related because they are members of the same parent structure. Although there exists an intuitive relationship between the predefined types, it is clear that if two functions use the same predefined type, it is does not indicate that these may belong to a common subsystem, as the predefined types exist to represent the most basic objects of any application domain. It follows that the *Relationship Principle* is not applicable to variables of predefined types.

Non-automatically detectable relationships are those that cannot be detected through an automatic approach without expert knowledge. Incorporating such reasoning is beyond the scope of this work.

4.3 Similarity metrics

A significant part of this investigation is the criteria for grouping program entities into clusters. The precise metrics to be used depend on the specification of the input data model, and metrics may have to be tailored to produce meaningful results for the application domain of source code. The main requirements regarding metrics were informed by the discussion above regarding the basic principles and are as follows:

1. The metric must be suitable for comparing binary or qualitative attributes, as this type of attributes is largely predominant in the application domain of source code.
2. The metric must take into account the relative importance of attributes, as the presence of some features may be more significant than the presence of others.
3. The metric must consider the distribution and rarity of features throughout the set of entities.
4. The metric must consider relationships between attributes themselves. When several entities share no common attributes, it is such relationships that could determine which entity is more similar to any of the rest.
5. The metric must normalise similarity measures taking into account the probability of a match between attribute values of two entities. This is because the significance of a match on an attribute depends on the characteristics of the complete set of attributes.

4.3.1 Choosing an appropriate metric

Different types of similarity measures were investigated including distance measures, correlation coefficients, probabilistic coefficients and association coefficients.

Distance measures are best suited to entities described by two or three numerical, continuous attributes. This measure has intuitive appeal because of the plots it uses to visually display the distribution of entities in the measurement space. However, this method loses its visual appeal when more than three attributes are considered, and the problem becomes multi-dimensional. The distribution of entities can no longer be observed on a plot. OLAP tools were developed to solve such problems [31].

Using distance measures on the binary attributes derived from source code would mean only two gradations, 1 and 0, on each axis. Furthermore, entities may have null scores for some attributes. The calculated distance between entities would then be a very crude measure of similarity, providing very little information content. Therefore, the advantages of using distance measures would not be exploited in this clustering application domain.

Correlation Coefficients [32] are mainly used to determine correlation, measured across a sample of entities, between pairs of continuous, numerical variables. For clustering source code, it is inappropriate to determine similarity between attributes, as opposed to entities, because clustering relies on the latter rather than the former. Furthermore, the binary, qualitative attributes derived from source code do not readily suit such similarity measures.

Association Coefficients [32] calculate similarity based on the number of features present and absent in entities and are suited to binary attributes. Anderberg [29] uses a grid to summarise this, as depicted in Table 1.

Table 1: Association Coefficients grid [29]

Entity *j* / **Entity *i***	*Present*	*Absent*
Present	*a*	*b*
Absent	*c*	*d*

In Table 1, *a* represents the number of features present in both entities and *d* represents the number of features absent in both entities. *b* is the number of features present in entity *i* but absent in entity *j*. *c* is the number of features present in *j* but absent in *i*. There are a number of coefficients available that measure similarity based on the values of *a*, *b*, *c* and *d*. Examples are the *Matching Coefficient* and the *Jaccard Coefficient.* Many others exist but these are unnamed [33]. These coefficients vary in the relative importance given to each of the four parameters. This means that weight may be added to give preference, say, to presence of features over absence of features.

These coefficients may need to be modified with source code entities because the presence of some features may be more significant than the presence of other features. A similar argument states that 0-0 matches are not an indication of similarity between entities. This is because entities may score *null* on numerous features. If this was interpreted as a sign of similarity, it would lead to such entities forming meaningless clusters.

Finally *Probabilistic Coefficients* [32] are based on the idea that agreement on rare features contributes more to the similarity between two entities than agreement on features that are frequently present. Probabilistic coefficients take into account the distribution of the presence of features over the set of entities. Probabilistic coefficients are developed to include feature distributions into similarity calculations. Given that probability coefficients are complex and must be tailored to suit a particular application, a similar function may be achieved by incorporating attribute-weight into association coefficients, which would be much simpler, yet effective.

4.3.2 Customising the metrics

Four types of similarity measures were investigated. Association Coefficients was identified to be the type of measure most suited to determine the similarity between program entities. These, however, need to be modified to account for the three principles mentioned above and tailored for the particularity of source code.

Accounting for similarity based on use of global variables, we suggest that one should differentiate between variables of two types: predefined or user-defined. The similarity metric for the use of global variable is specified separately for each type because the *Significance Principle* and the *Relationship Principle* are applicable to user-defined, but not predefined types. The two 'sub-metrics' are then combined to give an overall similarity metric.

Consider the basic data grid that leads to a simple association coefficient for two entities, *i* and *j*. This is now presented in a modified form to account for two types of global variables.

Table 2: Association Coefficients grid for two variables

Entity *j* / **Entity *i***	*Present*	*Absent*
Present	a_u+a_p	b_u+b_p
Absent	c_u+c_p	d_u+d_p

The subscript *u* indicates that the count is for attributes based on a global variable of user-defined type, whereas the subscript *p* indicates that the count is for attributes based on a global variable of predefined type.

Firstly, consider *attributes based on User-defined type Global Variables (UGV).*

A *basic* similarity coefficient for any such *UGV* attribute is given by (6):

$$S_{UGV}^{Basic} = \frac{a_U}{a_U + b_U + c_U + d_U}. \quad (6)$$

This metric gives an elementary indication of the similarity between *i* and *j*. It is simply the ratio of the attributes common to i and j to the total number of attributes (based on the use of global variables).

A similarity metric based on the *Usage Principle* is given by (7):

$$S_{UGV}^{Usage} = \sum_{1}^{a_U} \frac{2}{Total\ number\ of\ entities\ that\ use\ the\ UGV}. \quad (7)$$

Thus, if a UGV is used predominantly by *i* and *j*, this contributes a large value to the coefficient. If the UGV is also used by numerous other functions, then the contribution will be small. If a UGV is used *exclusively* by *i* and *j* then this will produce a maximum contribution of 1 to the above metric. In the special case where *all* of the UGVs covered by a_U are used exclusively by *i* and *j*, the metric will have a maximum value of a_U.

A similarity metric based on the *Significance Principle* is given by (8):

$$S_{UGV}^{Significance} = \sum_{1}^{a_U} \frac{Number\ of\ data\ items\ in\ UGV}{Maximum\ number\ of\ data\ items\ in\ a\ UGV}. \quad (8)$$

The contribution of an attribute to this metric will be higher if the attribute is based on a more sp*ecialised U*GV, that is, one with more data members, whereas an elementary UGV will produce a smaller contribution to the coefficient.

A coefficient based on the *Relationship Principle* is given by (9):

$$S_{UGV}^{Relationship} = \frac{Number\ of\ related\ parameters\ in\ 'b_U + c_U'}{b_U + c_U}. (9)$$

This metric analyses the attributes that are present in *i* but not in *j*, and vice versa. Such attributes are covered by the b_U and c_U counts. With a *basic* association coefficient, these 'mismatched' attributes would not count toward similarity whatsoever. This coefficient will exploit the usage of *related* global variables, as opposed to the usage of the *same* global variable, and thus contribute to the similarity measure. The more variables in b_U and c_U that are related, the higher the contribution will be.

Therefore, the overall similarity metric for attributes based on UGVs is calculated by combining the above four metrics. S_{UGV}^{Basic} and $S_{UGV}^{Relationship}$ do not need normalisation as their possible values lie between *0* and *1*. However, S_{UGV}^{Usage} and $S_{UGV}^{Significance}$ need normalisation, as they have a maximum value of a_U. Therefore, these two metrics will be divided by the total number of UGVs (sum of a_U, b_U, c_U and d_U) to constrain their values between *0* and *1*.

The overall similarity metric for attributes based on UGVs is given by (10):

$$S_{UGV} = \frac{S_{UGV}^{Basic} + \frac{S_{UGV}^{Usage}}{a_U + b_U + c_U + d_U} + \frac{S_{UGV}^{Significance}}{a_U + b_U + c_U + d_U} + S_{UGV}^{Relationship}}{4}. (10)$$

Note that the denominator of *4* applied in the above metric in order to constrain its value between *0* and *1*.

Now consider the *attributes based on Predefined type Global Variables* (PGV); that is counts with subscript *p*. These will only produce a coefficient based on the *Usage Principle*. There will be no coefficient based on the *Significance Principle* and the *Relationship Principle*. In a similar manner to before, the overall similarity metric for PGV attributes is given by (11):

$$S_{PGV} = \frac{S_{PGV}^{Basic} + \frac{S_{PGV}^{Usage}}{a_P + b_P + c_P + d_P}}{2}. \quad (11)$$

where:

$$S_{PGV}^{Basic} = \frac{a_P}{a_P + b_P + c_P + d_P}. \quad (12)$$

and:

$$S_{PGV}^{Usage} = \sum_{1}^{a_P} \frac{2}{Total\ number\ of\ entities\ that\ use\ the\ PGV}. \quad (13)$$

The coefficients S_{UGV} and S_{PGV} are now combined to give the *overall* similarity metric based on this class of attributes. Thus, the overall similarity metric for use of global variables, between entities i and j is given by (14):

$$S_{GlobalUsage} = \frac{W_U \times \left(\frac{N_U}{N_T} \times S_{UGV}\right) + W_P \times \left(\frac{N_P}{N_T} \times S_{PGV}\right)}{W_U + W_P}. \quad (14)$$

N_U, N_P and N_T represent the number of User-defined, Predefined and Total global variables in the program, respectively. W_U and W_P represent optional weights that can be applied in order to vary the relative importance of User-defined global variables of user-defined and Predefined type.

This similarity metric accounts for the ratio of user-defined over predefined type global variables in a program. It does this by placing more weight on the majority class of global variable. The division by N_T is performed to normalise the value of each term to between *0* and *1*. The optional weights have been included in order to allow the relative importance of attributes based on attributes based on Global Variables of user-defined and predefined type. Although the former attributes will always be more significant than the latter, it was decided not to fix this relative importance, but rather allow it to vary. This way, the performance of the approach can be evaluated using several different weights. In addition, in the tool, it may be left to the user to specify the value of these weights.

In a similar manner *overall* similarity metrics between entities *i* and *j*, based on the rest of the attributes, are given. The overall similarity metric for use of Local Variables (LV) between entities i and j is given by (15):

$$S_{LocalUsage} = \frac{S_{LV}^{Basic} + \frac{S_{LV}^{Usage}}{a + b + c + d}}{2}. \quad (15)$$

where:

$$S_{LV}^{Basic} = \frac{a}{a + b + c + d}. \quad (16)$$

and:

$$S_{LV}^{Usage} = \sum_{1}^{a} \frac{2}{Total\ number\ of\ entities\ that\ use\ the\ LV}. \quad (17)$$

It should be noted that user defined and predefined types are treated collectively, since the significance and relationship principals do not need to be considered, as explained in section 4.2.2.

The overall similarity metric for local variable types is given by (18):

$$S_{LocalTypes} = \frac{W_U \times \left(\frac{N_U}{N_T} \times S_{ULV}\right) + W_P \times \left(\frac{N_U}{N_T} \times S_{PLV}\right)}{W_U + W_P}. \quad (18)$$

where:

$$S_{ULV} = \frac{S_{ULV}^{Basic} + \frac{S_{ULV}^{Significance}}{a_U + b_U + c_U + d_U} + S_{ULV}^{Re\,lationship}}{3}. \quad (19)$$

$$S_{ULV}^{Basic} = \frac{a_U}{a_U + b_U + c_U + d_U}. (20)$$

$$S_{ULV}^{Significance} = \sum_{0}^{a_U} \frac{Number\ of\ data\ items\ in\ LP}{Maximum\ number\ of\ data\ items\ in\ a\ LP}. (21)$$

$$S_{ULV}^{Re\,lationship} = \frac{Number\ of\ related\ parameters\ in\ 'b_U + c_U'}{b_U + c_U}. (22)$$

and:

$$S_{PLV} = S_{PLV}^{Basic} = \frac{a_P}{a_P + b_P + c_P + d_P}. \quad (23)$$

The overall similarity metric for formal parameter types is given by (24):

$$S_{FormalTypes} = \frac{W_U \times \left(\frac{N_U}{N_T} \times S_{UFP}\right) + W_P \times \left(\frac{N_U}{N_T} \times S_{PFP}\right)}{W_U + W_P}. \quad (24)$$

where:

$$S_{UFP} = \frac{S_{UFP}^{Basic} + \frac{S_{UFP}^{Significance}}{a_U + b_U + c_U + d_U} + S_{UFP}^{Re\,lationship}}{3}. (25)$$

$$S_{UFP}^{Basic} = \frac{a_U}{a_U + b_U + c_U + d_U}. (26)$$

$$S_{UFP}^{Significance} = \sum_{0}^{a_U} \frac{Number\ of\ data\ items\ in\ FP}{Maximum\ number\ of\ data\ items\ in\ a\ FP}. \quad (27)$$

$$S_{UFP}^{Re\,lationship} = \frac{Number\ of\ related\ parameters\ in\ 'b_U + c_U'}{b_U + c_U}. (28)$$

and:

$$S_{PFP} = S_{PFP}^{Basic} = \frac{a_P}{a_P + b_P + c_P + d_P}. (29)$$

Finally, for returned values types $\boldsymbol{S_{RetVal}}$ equals 0 if either of the functions under comparison does not return a value.

If they both return values of the same type, then:

$$S_{RetVal} = \frac{S_{RV}{}^{Basic} + S_{U}^{Significance}}{2}. \quad (30)$$

where:

$$S_{RV}{}^{Basic} = 1 \quad (31)$$

and also, for user defined types:

$$S_{URV}^{Significance} = \frac{Number\ of\ data\ items\ in\ returned\ value\ type}{Maximum\ number\ of\ data\ items\ in\ a\ type}. \quad (32)$$

while for predefined types:

$$S_{URV}^{Significance} = 0 \quad (33)$$

otherwise if they return values of different types:

$$S_{RV}{}^{Basic} = 0. \quad (34)$$

thus:

$$S_{RetVal} = S_{URV}^{Relationship} = \frac{Number\ of\ pairs\ of\ related\ types\ in\ 'b+c'}{b+c}. \quad (35)$$

Note that S_{RetVal} can only have a value of either *0* or ½. Note also that since a function can only return a single value or none at all, this implies that the significance of this similarity metric will be relatively small compared to the metrics specified earlier. This will be taken into account when specifying the *total* similarity metric

To summarise the above it can be said that the similarity metric based on each class of attributes is specified separately, as each class has different characteristics to be considered. Furthermore, it was defined such that for all but one class of attribute, attributes based on variables of user-defined and predefined type are considered separately. The following four types of similarity metric were specified, with which to determine the similarity between program entities:

1. The *basic* similarity metric
2. The similarity metric based on the *Usage Principle*
3. The similarity metric based on the *Significance Principle*
4. The similarity metric based on the *Relationship Principle*

These metrics were found not to be applicable to all classes of attributes, and the latter two were deemed not to be applicable to attributes based on variables of *predefined* type. The following Table 3 summarises the applicability of the metrics to each class of attributes.

U and *D* represent attributes based on parameters of user-defined and predefined type, respectively. Note that only *local-variable-usage* attributes allow attributes based on variables of user-defined and predefined type to be analysed collectively.

Now that the similarity metrics based on each of the five classes of attributes have been specified, these must be combined to give the *total* similarity between two entities. For every entity, it will be

this *total* similarity value that will be computed with respect to every other entity. This complete set of similarity values forms the input to the clustering step.

Table 3: Applicability of similarity metrics to each class of attribute

Class of Attribute		**Similarity Metric**			
		Basic	*Usage*	*Signif.*	*Relation.*
Attributes based on Global Variable Usage	U	●	●	●	●
	P	●	●		
Attributes based		●	●		
Attributes based on the Type of Local Variables	U	●		●	●
	P	●			
Attributes based on the Type of Formal Parameters	U	●		●	●
	P	●			
Attributes based on the Type of Returned Values	U	●		●	●
	P	●			

The main issue to consider here is that of *inter-class importance* amongst attributes. This means that, for a given program, one class of attributes may be more important than another class. This is explained by the two examples below.

- A program may use a very limited number of user-defined data types, but there may exist extensive interaction between entities via calls from one function to another. This indicates that the information content of attributes based on *use* of local variables will be considerably more than attributes based on the *types* of local variables. Thus, the similarity metric based on the former class of attributes should be given more weight when computing the *total* similarity metric.
- Inevitably, there will be domain-specific features that will determine the relative importance amongst attribute classes; hence the importance of similarity metrics based on the attribute classes. Such features cannot be considered in an automatic approach to program understanding without the use of expert knowledge.

It follows that in this approach to program understanding, the importance placed on each similarity metric specified in the previous paragraphs is based on the relative numbers of attributes present in an attribute class. Thus, the total similarity between two entities, i and j, is as follows:

$$S_{Total} = \frac{(N_{GV} \times S_{GlobalUsage}) + (N_{LV} \times S_{LocalUsage}) + (N_{LV} \times S_{LocalType}) + (N_{FP} \times S_{FormalTypes}) + (N_{RV} \times S_{RetVal})}{N_{GV} + 2 * N_{LV} + N_{FP} + N_{RV}} \quad (36)$$

This metric applies a weight to each of the five metrics specified earlier. This weight depends on the relative numbers of variables on which each individual metric is based. These weights are: $N_{GV}, N_{LV}, N_{FP}, N_{RV}$ representing the number of global variables, local variables, formal parameters and returned values respectively. For example, if a program contains a larger proportion of global variables, then the metric based on global variables have the highest potential for information content. Thus, this metric will have the highest contribution to the *total* similarity metric. In a typical program, the count N_{RV} will have the smallest value of all the counts. Thus, the metric based on the

types of returned values, which is typically the least informative metric as pointed out in the previous section, should justifiably have the smallest contribution to the *total* metric.

4.4 Clustering strategy

This part reflects the algorithmic means by which subsystem structure of a program is derived. Several classes of clustering algorithms have been surveyed. The main requirements for the clustering strategy were as follows:

1. The clustering technique must create several solutions to the problem, as opposed to a single cluster distribution. This allows for the most meaningful clustering to be selected as a representation of the final subsystem abstraction. It should ideally form clusters *incrementally*, as opposed to a single step. The steps involved in this incremental cluster formation must be as small as possible so that if a suitable abstraction is not produced then it is possible to pinpoint the step in which the solution started to deviate from a model solution given by an expert.
2. The clustering technique must not rely on an initial partitioning to be made to the entities by the user, as this is only feasible when the user has some idea of the nature of the required clustering. Furthermore, the initial partition may influence the final result in an undesirable manner.
3. The clustering technique must not require the final number of clusters to be specified.

Different types of clustering algorithms were investigated including Partitional/Optimisation, density search and hierarchical algorithms. Hierarchical agglomerative algorithms were considered to be the most suitable ones for this work. They create a range of entity partitions. The starting point is when each cluster contains a single entity and the ending when all entities are contained in a single cluster. Clusters are formed incrementally, meaning that a single entity at a time is placed in a cluster. The algorithm does not require an initial partition, nor does it require the final number of clusters to be specified. Finally, the only data required by the algorithm is a similarity value for every possible pair of entities in the set. Therefore, all the requirements of the clustering strategy set above are satisfied.

Updating similarity measures is another ability of for hierarchical clustering algorithms. The similarity metrics discussed earlier can determine similarity between *individual* objects, but we also need to determine similarity between clusters containing a number of objects. The main methods used to that end are single linkage, complete linkage and average linkage rules [29].

Agglomerative hierarchical clustering algorithms utilise the total similarity metric described above taking into account the strength of all pair wise relationships between program entities, stored in a similarity matrix. Entities are grouped together depending on these similarities. This process visualised culminates in a '*dendrogram*' as shown in Fig. 1. The branches on the left-most side of the figure represent the starting point where all clusters contain a single entity. Moving to the right, the tree depicts an increasing aggregation of entities into clusters. The ending point is the tree root where all entities belong to the same cluster. Entities and clusters are merged with other entities and clusters, one at a time.

It follows that when the analysis for a set of n entities is complete; there are n sets of clusters from which to choose the most meaningful solution. For the tree of Fig. 1, there are six possible solutions as shown in Table 4.

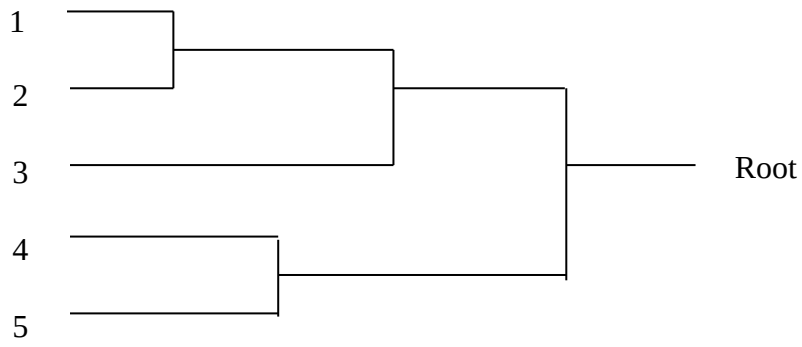

Figure 1: The dendrogram for a problem with single entities

Table 4: Possible solutions to the clustering problem of six entities

# Clusters	Clusters
1	{{{1,2}, {3}}, {4,5}, {6}}
2	{6}, {{{1,2}, {3}}, {4,5}}
3	{{1,2}, {3}}, {4,5}, {6}
4	{1,2}, {3}, {4,5}, {6}
5	{1,2}, {3}, {4}, {5}, {6}
6	{1}, {2}, {3}, {4}, {5}, {6}

In practice, if clustering is allowed to continue until all of the entities are contained in a single cluster, this may result in *forced clustering*; that means an entity may be forced into a cluster even though it is highly dissimilar to the entities in the cluster. This can be avoided by specifying a similarity *threshold*, where the clustering process is stopped if the maximum similarity between two clusters does not exceed the threshold value.

A more formal description of the clustering algorithm, for a problem of n entities, with a similarity threshold of S_T, is depicted in Fig. 2.

1. Begin with n clusters, each containing a single entity. Let the clusters be labelled with labels from 1 to n.
2. Calculate the similarity between every pair of clusters, using the method described previously; Search for the most similar pair of clusters using the associated similarity values. Let these clusters be labelled p and q, and let their associated similarity value be S_{pq}.
3. (a) Merge clusters p and q into a single new cluster r. Reduce the number of clusters by one. If $S_{pq} < S_T$, mark the current cluster distribution with a tag indicating that the similarity threshold is exceeded. Proceed to *step 4*.

   (b) If $S_{pq} < S_T$, terminate the clustering process. The current cluster distribution is the proposed optimum solution.
4. Perform steps 2 and 3 at most *n-1* times, until all entities belong to a single cluster.

Figure 2: The clustering algorithm

The algorithm accounts for a decision whether to continue or stop the clustering process once the similarity threshold exceeded. Therefore, at step 3, paths (a) and (b) represents processing when is or is not used respectively.

4.5 Assumptions and Limitations

DMCC is an approach designed to operate on C and non-object-oriented features of C++ code; it does not address concepts such as inheritance or polymorphism. It processes code within functions and ignores code outside functions. *DMCC* performs static analysis, not dynamic analysis. It is an automated approach and as such it does not require expert knowledge about the system under examination, nor does it use any domain specific information; it only uses the language's grammar and syntax.

Along with these generic assumptions and limitations, there are also certain assumptions and limitations which apply to the input model used by *DMCC*. More specifically, *user defined* types may better discriminate parameters than *predefined* ones. The usefulness of a *global variable* as an attribute depends on its nature, which is largely influenced by its *type*. More effective global variables are likely to belong to a user-defined type. This is because most user-defined types are specified to model a particular aspect of the problem domain, and functions that use these are likely to be associated with this aspect of the problem domain, indicating membership of a common subsystem. Thus, the differentiation on whether the type of attributes is *user defined* or *predefined* was considered to be necessary and will be also used for other types of attributes, where applicable. It is hypothesised that the overall purpose of a function, hence its parent subsystem, can be predicted by examining the nature of the data items on which it operates. The nature of the data items can be determined by their type. It follows that functions having in common the types of several variables are likely to have a similar purpose, hence may belong to the same subsystem.

The use of attributes based on the *types of formal parameters* has the same justification as above. The type of the data operated upon is likely to be an indication of the function's parent subsystem. However, this type of attribute cannot be useful for functions without any formal parameters.

The use of function attributes based on the *types of returned values* has also the same justification as above. The type of the data operated upon is likely to be an indication of the parent subsystem of a function. However, this type of attribute will not be useful for void functions.

Presumably a scale of importance governs the *significance* of attributes. However, the exact difference in significance between different types of attributes is not apparent and should depend on the individual characteristics of the program (and may even be subjective), thus requiring expert knowledge to deduce. However, a basic way of constructing such a scale for user-defined types would be to place greater importance on more specialised or complex types. This is because user-defined types are specified to model a more specialised aspect of the application domain, thus the use of such a variable is more indicative of a function's purpose.

An important point to note is that a type may be more closely related to some types than others. For example, the type *int* is more closely related to *float* than *char*, or the type *double* is more closely related to the *float* than *int*. Such *relationships* may also occur between user-defined types.

5. EXPERIMENTATION AND EVALUATION

The method for deriving the input model from C/C++ source code by extracting entities and the relevant attributes was described earlier in this chapter. The way that these entities could be clustered

in order to retrieve a potentially meaningful modularization of a system was formulated. The approach was evaluated in practice, to assess its effectiveness and to seek possible improvements. A prototype clustering tool was thus developed in order to experiment with real programs.

In order to assess *DMCC* an evaluation procedure was devised. The proposed procedure comprises four main steps as depicted in Fig. 3.

Experts' mental models represent an indication of the subsystem structure of the programs and can be largely subjective. Therefore, it is advisable whenever results are not consistent with the mental model, that the input-model is examined, to give an insight into the accuracy of the mental model and indicate possible reasons for the inconsistency.

1. Parse and search source code to produce the input for the clustering engine.
2. Feed the input to the clustering engine.
3. Derive subsystem abstractions of the program in the form of dendrograms using single (minimum), complete (maximum) and average (weighted) linkage methods. It is shown that the choice of the method affects the nature of the clustering produced.
4. Compare the derived subsystem abstractions to an expert's mental model of the system, if available.

Figure 3: The evaluation procedure for *DMCC*

It should be noted for the purposes of this evaluation, the similarity threshold was not used to stop the clustering process, so that all possible sets of clusters could be derived and convergence to or divergence from the expert's mental model could be observed. The threshold was merely used as a means to indicate different clustering schemes at different levels in the hierarchy of clusters.

Precision and recall were introduced as a quantitative element in judging the accuracy of the results of the approach [12]. In this case, the *precision p* for a subsystem is the percentage of entities in the subsystem that belong to the subsystem according to the expert's mental model. Therefore, precision is high if a cluster (subsystem) contains few only entities which belong to other subsystems. On the contrary, precision is low if a cluster contains many entities which belong to other subsystems. The *recall r* for a subsystem is the percentage of entities belonging to the subsystem's mental model which are actually present in the subsystem. Therefore, recall is high if the subsystem contains most of the entities suggested by the mental model; it is low if the subsystem contains few only of the entities suggested by the mental model. Consequently, a subsystem of 'good quality' should present both high precision and high recall.

However, it is known that normally there is a tradeoff between precision and recall, in other words if one tries to increase recall precision deteriorates. In order to assess overall accuracy taking into account both the precision and the recall at the same time let us propose a simple indicator called the *performance P* of a clustering method, which is the sum of the products (precision) x (recall) for every subsystem. An alternative measure, often used in the literature, is the *balanced F-score*, which is the weighted harmonic mean of precision (*p*) and recall (*r*). The balanced F-score is a value from 0 to 1 inclusive, given by the formula (37):

$$F = \frac{2 * p * r}{p + r} \quad (37)$$

The total balanced F-score F_t for a system is the sum of the F-scores of all constituent subsystems. However, it should be noted that neither P nor F_t takes into account the size of these subsystems. This implies that capturing accurately a small subsystem contributes equally to the value of P or F_t, as compared to capturing accurately a large subsystem. For example, correct identification of a subsystem which only contains one function contributes equally to correct identification of a subsystem which contains twenty functions which by comparison is not such a trivial task.

To account for the size of various subsystems let us introduce another accuracy measure the *Performance Index* (*PI*). *PI* is calculated by summing up the weighted products of precision and recall for each subsystem; weights depend on the number of entities that are present in each subsystem. *PI* for a clustering at a specific level, is given by the formula (38):

$$PI = \sum_{k=1}^{m} \frac{p_k * r_k * e_k}{n} \quad . \quad (38)$$

where m is the number of subsystems, n is the total number of entities, and p_k, r_k and e_k are respectively the precision, recall and number of entities of each subsystem. *PI* is more informative compared to P or F_t as used the same principles, but penalises trivial classification success, in favour of more challenging tasks.

The source code of four programs has been used for experimentation. *T2*[2] was analysed to verify the applicability of the approach and explore its potential. *T2* is a small/medium C program with a known mental model, which contains 46 functions in 18 modules. *Plunder*[3] was analysed to evaluate the suitability of the approach for C++ code. *Plunder* is a small exemplary C++ program with a known mental model, which contains 21 functions in a single module. *Ccg*[4] was analysed to assess the scalability of the approach and its suitability when dealing with unfamiliar software. *Ccg* is a medium/large C program with known functionality but without a known mental model, consisting of 63 functions in 8 modules. Finally, two systems, *Ccg* and the *Combiner*[5], were merged into one, which was analyzed to assess how robust the approach was when given "unexpected" inputs. The system consists of 118 functions in nine modules and lacks a mental model. The following paragraphs detail the results from these experiments.

5.1 Results for *T2*

T2 is a program that constructs decision trees used for classification. It is perceived to contain six subsystems, with 6, 5, 8, 9, 5, 11 functions respectively and two autonomous functions. In the baseline case, with all the entities grouped together resulting in a recall of 100%, the precision for each subsystem is respectively: 13%, 11%, 17%, 20%, 11%, 24%, and 2% for each of the two

[2] *T2* computes optimal 2-level decision trees. It was written in C by P. Auer and modified by M. Sahami to generate intervals for the MLC++ interface. The code is public domain and can be found at: http://openscience.org/mlc/.

[3] *Plunder* was developed at the School of Computer Science, University of Manchester.

[4] *Ccg* is a C parser developed at the Dept. of Computer Science, University of Durham.

[5] This module is essentially the "combiner" phase of the U. of Arizona Portable Optimizer but was redone in order to be used for optimising the GNU compiler (Copyright (C) 1995 Free Software Foundation, Inc.) by combining instructions.

autonomous functions. The baseline performance would thus be P=1, the total balanced F-score F_t =0.215 and the Performance Index PI=0.167.

The single linkage method produced numerous clusters containing 2 or 3 entities that, according to the mental model, belong to the same subsystem. However, accuracy was greatly reduced when the smaller clusters were merged with other clusters. That is because the method uses the minimum entity-to-entity similarity across two clusters when calculating their similarity. This means that a number of strong entity-to-entity relationships across two clusters do not increase their similarity. There were several points in the clustering process where the formation of a subsystem started accurately, but then deviated from the mental model, due to the small value of the minimum entity-to-entity similarity across the clusters, which would have otherwise been merged. The best performance for subsystem 1 was achieved after 30 steps of clustering, where precision was 67%, recall 33%, and F=0.442. The best performance for subsystem 2 was achieved after 6 steps of clustering, where precision was 100%, recall 40% and F=0.571**.** The best performance for subsystem 3 was achieved after 21 steps of clustering, where precision was 100%, recall 38% and F=0.551. The best performance for subsystem 4 was achieved after 10 steps of clustering, where precision was 100%, recall 22% and F=0.361. The best performance for subsystem 5 was achieved after 18 steps of clustering, where precision was only 50%, recall 40% and F=0.444. The best performance for subsystem 6 was achieved after 8 steps of clustering, where precision was 100%, recall 36% and F=0.529. The best overall performance of this method was reached after 21 steps, where P=3.44, F_t=0.591 and PI=0.283 and similarity was above 0.0135.

The complete linkage method did not produce meaningful results, but simply one large cluster into which all other entities were merged one by one. This effect was also observed in the analysis of *Plunder* and *Ccg*, and it was also described in the literature survey on *file clustering* conducted in [17]. The best performance for subsystem 1 was the same as the baseline: precision of 13%, recall 100%, and F=0.231 at the top level of the hierarchy after 45 steps of clustering. In other words, subsystem 1 was not captured, expect from when all entities were in one cluster. The best performance for subsystem 2 was achieved after 11 steps of clustering, where precision was 100%, recall 40% and F=0.571. The best performance for subsystem 3 was achieved after 30 steps of clustering, where precision was 23%, recall 88% and F=0.365. The best performance for subsystem 4 was achieved after 35 steps of clustering, where precision was 25%, recall 100% and F=0.400. The best performance for subsystem 5 was achieved after 7 steps of clustering, where precision was 38%, recall 60% and F=0.465. The best performance for subsystem 6 was achieved after 12 steps of clustering, where precision was 75%, recall 82% and F=0.783. The best overall performance of this method was reached after 41 steps, where P=2.97, F_t=0.46 and PI=0.215 and similarity was above 0.0080.

The average linkage method initially produced clusters containing 2 or 3 entities, which were consistent with the mental model. Several smaller clusters were subsequently merged with them to form larger clusters, which were also consistent with the mental model. The best performance for subsystem 1 was the same as the baseline: precision of 13%, recall 100% and F=0.231, at the top level of the hierarchy after 45 steps of clustering. The best performance for subsystem 2 was achieved after 16 steps of clustering, where precision was 100%, recall 60% and F= 0.750. The best

performance for subsystem 3 was achieved after 27 steps of clustering, where precision was 100%, recall 50% and F=0.667. The best performance for subsystem 4 was achieved after 37 steps of clustering, where precision was 25%, recall 89% and F=0.390. The best performance for subsystem 5 was achieved after 12 steps of clustering, where precision was 30%, recall 60% and F=0.4. The best performance for subsystem 6 was achieved after 10 steps of clustering, where precision was 78%, recall 64% and F=0.703. The best overall performance of this method was reached after 16 steps, where P= 3.23, F_t=0.652 and PI=0.257 and similarity was above 0.0110.

The optimal results achieved by each linkage type along with the baselines values for P, F_t, and PI are summarised in Fig. 4. We can observe here, that single linkage outperformed both average and complete linkage when applied to *T2*, in terms of Performance P and the Performance Index PI, but it was average linkage performing better in terms of the total balanced F-score F_t; complete linkage clearly ranked last, in terms of every global accuracy indicator.

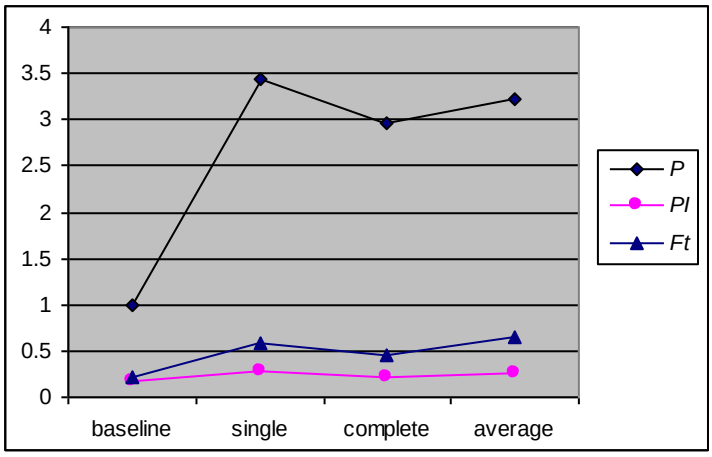

Figure 4: Summary of *DMCC* results for *T2*

A more comprehensive summary of the results is given in Table 5, which details how the single, complete and average linkage methods faired in comparison to the baseline performance. The table shows results for performance P, Performance Index PI and total balanced F-score F_t for the full system (Table 5-a) or precision p, recall r, performance P, and balanced F-score F for the six subsystems (Table 5 b & c). Numbers in bold indicate the highest value per column.

Table 5: *DMCC* results for *T2* and its six subsystems

	T2		
	P	PI	F_t
Baseline	1.00	0.167	0.215
Single	**3.44**	**0.283**	0.591
Complete	2.97	0.215	0.460
Average	3.23	0.257	**0.652**

(a)

	Subsystem 1				**Subsystem 2**				**Subsystem 3**			
	p	r	P	F	p	r	P	F	p	r	P	F
Baseline	0.13	**1.00**	0.13	0.231	0.11	**1.00**	0.11	0.196	0.17	**1.00**	0.17	0.296
Single	**0.67**	0.33	**0.22**	**0.442**	**1.00**	0.40	0.40	0.571	**1.00**	0.38	0.38	0.551
Complete	0.13	**1.00**	0.13	0.231	**1.00**	0.40	0.40	0.571	0.23	0.88	0.20	0.365
Average	0.13	**1.00**	0.13	0.231	**1.00**	0.60	**0.60**	**0.750**	**1.00**	0.50	**0.50**	**0.667**

(b)

	Subsystem 4				Subsystem 5				Subsystem 6			
	p	*r*	*P*	*F*	*p*	*r*	*P*	*F*	*p*	*r*	*P*	*F*
Baseline	0.20	**1.00**	0.20	0.327	0.11	**1.00**	0.11	0.196	0.24	**1.00**	0.24	0.386
Single	**1.00**	0.22	0.22	0.361	**0.50**	0.40	0.20	0.444	**1.00**	0.36	0.36	0.529
Complete	0.25	**1.00**	**0.25**	**0.400**	0.38	0.60	**0.23**	**0.465**	0.75	0.82	**0.62**	**0.783**
Average	0.25	0.89	0.22	0.390	0.30	0.60	0.18	0.400	0.78	0.64	0.50	0.703

(c)

Examining the results, it becomes apparent that, in terms of Performance P, complete linkage gives better "local" results in three out of six subsystems when single linkage gives better results in just one out of six subsystems (Table 5 b & c). However, single linkage is the best overall performer at the system level (Table 5-a). Similarly, complete linkage gives better "local" results in three out of six subsystems, in terms of balanced F-score *F*, when average linkage gives better results in two out of six subsystems (Table 5 b & c). Again, when it comes to overall performance, average linkage is best at the system level and complete linkages is the worst (Table 5-a). This indicates that local optima can be achieved by complete linkage but for global optimum one should select either single or average linkage.

Overall, it can be claimed that experimenting with *T2* not only indicated the feasibility of the approach, but also highlighted its strengths in capturing the logical modularity of the system, as it was up to 3.44 times, in terms of Performance *P*, up 3.03 times, in terms of total balanced F-score F_t, and up 1.54 times, in terms of Performance Index *PI*, more accurate than a "wild guess", which would place all the entities in a single cluster.

5.2 Results for *Plunder*

Plunder is a small program that is used to create and maintain a register of residents in a hotel. According to an expert's mental model this program contains two main subsystems, consisting of 6 and 7 functions each respectively, and 8 autonomous functions, which do not belong to any particular subsystem. In the baseline case, with all the entities grouped together, resulting in 100% recall, the precision for each subsystem is respectively 29% and 33% while precision for the autonomous entities is 4.8%. The baseline performance thus is P=1, F_t=0.167 and PI=0.21.

The single linkage method successfully produced a subsystem abstraction. The most meaningful clustering for subsystem 1 was formed with 100% precision, 83% recall and F=0.907, after 12 clustering steps. For subsystem 2, the most meaningful clustering was achieved after 10 steps, with 87% precision, 100% recall and F=0.930. The best overall performance of this method was achieved after 12 steps; at this stage, the overall precision was 95%, as only one out of 21 entities was misclassified. It can be observed in Fig. 5, how functions 0,1,2,3,4 and 5 (in bold) that belong to subsystem 1, eventually gravitate towards the cluster on the left, while functions 6,9,10,11,12,13 and 14 (in italic) which belong to subsystem 2, eventually gravitate towards the cluster in the middle, and 7 out of 8 autonomous functions (7,8,15,16,18,19,20) remain separate. Similarity was 0, P=8.74, F_t=0.891 and PI=0.83.

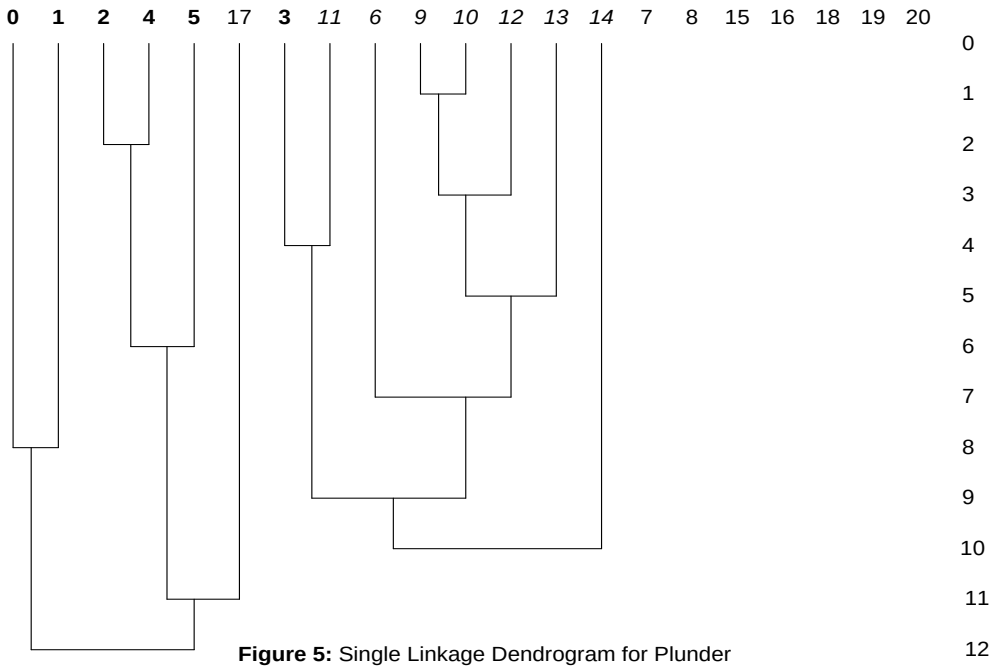

Figure 5: Single Linkage Dendrogram for Plunder

In the complete linkage method subsystem 2 is formed consistently with the mental model early in the clustering process. Subsystem 1 began to form consistently with the mental model, but later it deviated from it. The most meaningful clustering for subsystem 1 was formed with 100% precision, 66% recall and F=0.795, after only 4 clustering steps. For subsystem 2, the most meaningful clustering was achieved after 6 steps, with 100% precision, 71% recall and F=0.830. After just 6 clustering steps the overall precision was 71% but it did not get any better. Similarity was above 0.5668 at this point, while P=8.96, F_t=0.933 and PI=0.676.

Using the average linkage method both subsystems were reproduced accurately. The most meaningful clustering for subsystem 1 was formed with 100% precision, 83% recall and F=0.907, after 11 clustering steps. For subsystem 2, the most meaningful clustering was achieved after 10 steps, with 88% precision, 100% recall and F=0.936. The optimum level appeared after 11 clustering steps, where similarity was above 0.0162. At this point, all entities that were perceived not to belong to any particular subsystem were not placed in any subsystem. The overall precision was 95%, as only one entity out of 21 was misclassified, while P=9.71, F_t=0.984 and PI=0.911.

The optimal results achieved by each linkage type along with the baselines values for P, F_t and PI are summarised in Fig. 6. We can observe here that average linkage outperformed both single and complete linkage when applied to *Plunder*.

A more comprehensive summary of the results is given in Table 6, which details how the single, complete and average linkage methods faired in comparison to the baseline performance. The table shows results for performance P, Performance Index PI and total balanced F-score F_t for the full system (Table 6-a) or precision p, recall r, performance P, and balanced F-score F for the two subsystems (Table 6.3 b). Numbers in bold indicate the highest value per column.

Results show that average linkage achieves higher Performance P both at the system and the subsystem level, while single linkage only fairs well at the subsystem level. The same is largely true in terms of the balanced F-score F: average linkage is the best both at the system and the

subsystem level, while single linkage only fairs well at the subsystem level. So, for *Plunder* average linkage was shown to be best both locally and globally.

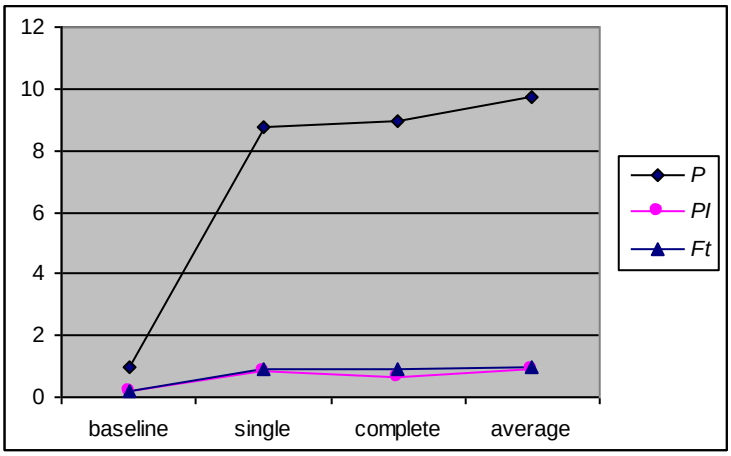

Figure 6: Summary of *DMCC* results for *Plunder*

Table 6: *DMCC* results on *Plunder* and its two subsystems

(a)

	Plunder		
	P	*PI*	F_t
Baseline	**1.00**	0.210	0.167
Single	8.74	0.830	0.891
Complete	8.96	0.676	0.933
Average	**9.71**	**0.911**	**0.984**

(b)

	Subsystem 1				**Subsystem 2**			
	p	*r*	*P*	*F*	*p*	*r*	*P*	*F*
Baseline	0.29	**1.00**	0.29	0.444	0.33	**1.00**	0.33	0.500
Single	**1.00**	0.83	**0.83**	**0.907**	0.87	**1.00**	0.87	0.930
Complete	**1.00**	0.66	0.66	0.795	**1.00**	0.71	0.71	0.830
Average	**1.00**	0.83	**0.83**	**0.907**	0.88	**1.00**	**0.88**	**0.936**

These results are consistent, although not entirely identical to these observed by experimenting with *T2*. The underlying message appears to be that of selecting either single or average linkage with a similarity threshold of 0.01, when aiming at getting globally accurate system decomposition. It should be emphasised however that experiments with *Plunder* were primarily aimed at confirming the applicability of *DMCC* to C++ systems. Its small size does not allow for conclusions from the clustering to be generalised.

The accuracy of *DMCC* was up to 9.71 times better in terms of Performance *P*, 5.89 times in terms of total balanced F-score F_t and 4.34 times better in terms of Performance Index *PI*, compared to the baseline. This experiment showed also the applicability and suitability of the approach for C++, as it accurately captured the structure of the system.

5.3 Results for *Ccg*

Ccg is a C parser consisting of a rather large number of modules and functions. As stated before, there was no mental model available, and the program was mainly used to evaluate the scalability

of the proposed approach and its suitability for larger software systems. All three methods of linkage (single, complete, average) were used again. However, complete linkage was already known from the previous experiments, to result in one big cluster which eventually "absorbs" all the others, thus limiting its potential to reveal the true subsystem structure; this was once again shown to be the case. Due to the absence of a mental model, one can only observe the results in relation with the previous findings of this approach and interpret them, just like a maintainer unfamiliar with the program would do.

The single linkage method resulted in an even spread of clusters. At the top level there were 13 clusters and 2 autonomous entities. Imposing a threshold to the similarity at 3 levels, namely 0.0025, 0.005 and 0.01 resulted in 13 clusters and 10 autonomous entities, 15 clusters and 21 autonomous entities, 5 clusters and 49 autonomous entities, respectively.

The complete linkage method resulted, as expected, in a single large cluster at the top level. Imposing a threshold to the similarity at 3 levels, namely 0.0025, 0.005 and 0.01 resulted in 3 clusters and 5 autonomous entities, 7 clusters and 19 autonomous entities, 2 clusters and 47 autonomous entities, respectively. Comparing the clusters derived by this method to the ones derived by the complete linkage, one may conclude that initially the clusters formed are similar, but later they deviate as the single linkage forced entities to be grouped together in one large cluster.

Using the average linkage method, which was known to give reliable results, as long as similarity is kept above 0.01, it was decided to impose this threshold at first and examine the outcomes. 3 clusters were then produced leaving out 53 autonomous entities. All three clusters contained functions belonging to different modules. However, by examining closely the code for these functions, one can see that they have similar functionality and are related, like for example functions *make_declaration* and *dec_list_insert*, in Fig. 7. 3 larger clusters were formed, and 49 autonomous entities were left out, when the similarity threshold was set to 0.005. Consistency to the above principal of related functions grouped together was preserved. Setting the threshold to 0.0025 led to the formation of 13 clusters, leaving out 20 autonomous entities. This time some functions grouped together appeared not to be closely related to each other. At the top level there is a single cluster grouping all the entities together. Interestingly, when applying a threshold of 0.0013 the model derived is nearly identical to the one derived by the single linkage method with a threshold of 0.0025. There are 10 identical clusters and three clusters of the latter method containing the same entities as the ones in the last cluster of the former method.

One may conclude, given the results from *T2* and *Plunder* that the average linkage method is indeed the most accurate one, and used in conjunction with the single linkage can give a good overview of a system's decomposition. The approach described earlier was shown to be scalable, although it could be tested on yet larger programs. It can also be claimed that using the models derived by the single and the average linkage methods, facilitated understanding *Ccg*, however the claim is subjective and its justification beyond the scope of this work.

```
DECLARATION_PTR make_declaration (int type, DEC_SPEC_PTR dec_spec, DEC_LIST_PTR
dec_list, DECLARATION_PTR next)
{  DECLARATION_PTR helpp;
  if ((helpp = (DECLARATION_PTR) malloc(sizeof (DECLARATION))) == NULL){
    fprintf(stderr, "Out of Memory\n");
    return (NULL);
  }
  helpp->type = type;
  helpp->dec_spec = dec_spec;
  helpp->dec_list = dec_list;
  helpp->next = next;
  return(helpp);}

DEC_LIST_PTR dec_list_insert (DEC_LIST_PTR list, DEC_LIST_PTR element)
{  DEC_LIST_PTR help;
  help = list;
  if (element == NULL)
    return (list);
  if (list == NULL)
    return (element);
  while (help->next != NULL)
    help = help->next;
  help->next = element;
  return (list);}
```

Figure 7: Excerpts from *Ccg* source code

5.4 Results for *Ccg* mixed with *Combiner*

A final experiment was run this time mixing the entities derived from *Ccg* with the ones derived from another program, the *Combiner*. The *Combiner* is in fact a C program which was originally meant to be used to test the parsing capabilities of *Ccg*. It consists of a single module of 55 functions. The idea behind this experiment was to attempt to "confuse" *DMCC* and test its response when inputted two merged systems instead of one. The objective of this experiment was to validate the ability of the system to cope with "unexpected" input.

The single linkage method avoided grouping functions from the two different programs together even at the top level. There was only one exception, when two functions were clustered together at the very final step when the similarity had dropped to 0.001. Other than that, this method resulted in a clustering scheme for *Ccg* very similar to the one produced by the previous experiment. *Combiner*'s entities were grouped in separate clusters, which appeared to be meaningful.

Using the complete linkage method ended up mixing entities of the two programs soon after similarity dropped below 0.005, but even before that, clusters failed to portray similarity in a comprehensive manner, unlike the other two methods.

Using the average linkage method the two systems were kept separate, as long as the similarity threshold was kept above 0.001. More precisely 54 out of the 55 functions in *Combiner* were put together in clusters consisting of functions belonging only to *Combiner* while the last one was kept autonomous. In the same manner, except from 7 autonomous functions that belonged to *Ccg*, all the 56 remaining functions were grouped in clusters consisting of functions belonging only to *Ccg*. Eventually the method resulted on a single cluster at the top level. However, the results of this method were once again consistent to the ones derived by the single linkage method.

This final experiment produced evidence that *DMCC* could be used to retrieve the original constituent components of a single system which contains a number of such merged components. It has also confirmed the complete linkage method's known weakness of forcing clusters together when little actual similarity exists.

5.5 Evaluation

DMCC was evaluated using data extracted from C/C++ systems of various sizes, of up to 118 functions (or up to 18 modules). Experimental results showed that a multi-layered, high-level abstraction of system, as a number of subsystems containing "similar" functions, can be achieved by clustering program functions into groups.

The accuracy of the results was evaluated by comparing the produced subsystem abstractions with experts' mental models. The abstractions were shown to be accurate, capturing subsystems consistently with the mental models. Pair wise values of precision and recall ranged between (13%, 100%) and (88%, 100%). The highest precision achieved was 100%, the highest recall 100% and the highest balanced F-score *F* was 0.936. The lowest values for precision, recall and balanced F-score were respectively: 13%, 22/%, 0.231, but *DMCC* never performed worse than the baseline accuracy.

Experimentation with *Plunder* and *T2* demonstrated that the approach could successfully be applied to both C and C++ programs. It was indicated however, that the more the entities and the subsystems the less accurate the approach. This is not surprising, given that the larger the system the higher the entity misclassification probability. It was also shown that complete linkage is not very suitable for capturing a system's modularity but can give insights into the subsystems that come up accurately enough during the early stages of clustering.

It was the average, and to a lesser degree the single linkage method which gave better results and captured accurately the structure of subsystems. In fact, experiments strongly suggested that the best results can be achieved using these methods, whilst setting the similarity threshold to approximately 0.01; below this threshold clusters tend to be less accurate. At this stage, one can get an accurate overview of the system and its decomposition into subsystems, which of course is only the first step towards understanding it.

DMCC was not only shown to be applicable to C code (*T2*, *Ccg*, *Combiner*) and to non object-oriented aspects of C++ code (*Plunder*), but also scalable and robust. It was shown to be scalable as it can handle not only small systems (*Plunder*) but also medium sized (*T2*) and larger systems (*Ccg*, *Combiner*). Robustness was shown when *DMCC* dealt successfully with code originating from two different systems (*Ccg* merged with *Combiner*). *DMCC* was shown to produce accurate results

wherever there was an expert's mental model to compare to, or at least meaningful results wherever there was no such model. In other words, it can be reasonably expected that *DMCC* can assist software maintainers to understand the structure of unfamiliar systems.

DMCC produces systems' overviews, which *aid comprehension*. Grouping program components into subsystems can *reduce the perceived complexity* thus facilitating maintainers. Complexity can be detected by identifying subsystems which consist of comparatively large number of functions. Large, complex and strongly interrelated subsystems are likely to be fault-prone [34].

DMCC forms clusters of functions using the similarity between attributes, which are derived based on the use and types of variables or parameters and the types of returned values. It should be graspable how this approach can facilitate maintenance tasks, such as *code modifications*. For example, if the maintenance engineer planned to change the use of parameters within the body of a function it would be advisable to check whether other "similar" functions are affected. This can facilitate impact analysis and risk assessment related with fast code modifications.

DMCC could also be used in *perfective maintenance*, when attempting to improve systems cohesion and coherence, in other words when the goal is to increase system modularity. This could happen by relocating functions into modules where they "naturally" belong. For example, a function may have limited similarity with functions in the same module but high similarity with functions in another module: in this case moving it to this other module should be considered. Alternatively, one could consider adjusting what a function does, in a manner reflecting consistently the overall functionality of the module it belongs to; increasing similarity within the module should result from this adjustment.

6. CONCLUSIONS AND FUTURE WORK

This work was concerned with software maintainers' need to understand a system before performing maintenance tasks if they are unfamiliar with it; a system overview is obviously highly desirable [1]. Earlier work has produced evidence that data mining techniques can derive high level system overviews [16]. However, it is clear, that such overviews are necessary both at the level of files and programs and at a lower level, such as the function level. Furthermore, deriving system abstractions could indicate interconnections between system parts in a manner that could expose potential risks when modifying the system.

Innovative advanced data analysis has also been found to support software development teams, by increasing cognition and decision making [40]. This is achieved by gathering input raw data to calculate and visualize more advanced metrics and find correlations between them. High-level data are then shown to product owners to increase their cognition, situational awareness, and decision-making capabilities.

The proposed approach enhances the existing ways of using data mining in support of program comprehension in that it addresses components at the program level, while other methods operate at the system level [11], [12], [15], [16], and [20]. It bears similarities with other non data mining based methods in that it produces dependency relationships between software artefacts, like the dependency tracer proposed in [35] or the approach for decomposing legacy systems into objects [36].

DMCC is more similar to ISA [15], MDG [16] and ASDC [11] in that it requires only general computer science knowledge as they seek an understanding in a rather model free, opportunistic fashion. That differentiates it from other program comprehension techniques which require specific domain knowledge, and as a results significant pre-processing effort [37], [38].

This work focused on addressing the issue of assessing the feasibility of clustering program functions depending on the types and use of a number of parameters. It also investigated the appropriateness of different clustering strategies. Finally, it explored the suitability of the approach for different types of programming languages and sizes of programs. A tool was developed for this reason.

The tool was used to obtain the subsystem abstraction of 3 programs. The accuracy of the results was evaluated by comparing the subsystem abstractions with experts' mental models for two of the programs. The produced results were found to be meaningful in most cases. In any case, even when inaccurate results occur, it could be argued that this happens partly because of possible inconsistencies of the expert's mental model with the actual organisation of the program.

In addition to evaluating the accuracy of the results, the feasibility of this approach was evaluated by assessing the *complete* process of obtaining a subsystem abstraction of a program. Overall, this investigation demonstrated the potential of the proposed approach for obtaining a subsystem program abstraction and for identifying interrelations between modules and functions. Admittedly further improvements would be required before the approach produces optimum results. Limitations for the approach were identified and suggestions made on how to further develop it and improve its accuracy.

The process of extracting the input data from the program and pre-processing it needs further automation. We plan to investigate the use of compilers and parsers to this end. So far, we have managed to acquire a subset of the input data using a parser. The approach needs to be further specified and enriched to facilitate as much automation of the derivation of the input data model as possible. Alternatives, including using empirical data to assign weights (e.g., from empirical studies, or from learning algorithms) can be explored. It would also be desirable if this approach could be incremental in a manner accommodating changes in software, as the assumption of static software is unrealistic.

Furthermore, the indication that the approach may lose accuracy as programs grow larger, needs further investigation. Other directions for future work include fine tuning of tool-related parameters for optimal results, allow processing of program files containing non-syntactic constructs such as textual macros [39], extending the input data model with more attributes such as function calls, and also exploring the possibility of utilising features related to classes in object oriented programming languages, such as C++ or C# [41].

ACKNOWLEDGEMENTS

The author would like to thank Emeritus Professor of Software Engineering Malcolm Munro at the Dept. of Computer Science, University of Durham, for providing the source code of Ccg.

REFERENCES

1. C. Tjortjis and P.J. Layzell, “Expert Maintainers’ Strategies and Needs when Understanding Software: A Qualitative Empirical Study”, *Proc. IEEE 8th Asia-Pacific Software Engineering Conf.* (APSEC 01), IEEE Comp. Soc. Press, pp. 281-287, 2001.
2. I. Sommerville, *Software Engineering*, 10th ed., Addison-Wesley, 2016.
3. P. Grubb, A.A. Takang, *Software maintenance: concepts and practice*, 2nd ed., World Scientific, 2003.
4. B. Eddy, “Structured source retrieval for improving software search during program comprehension tasks”, *Proc. ACM SIGPLAN Conf. Systems, Programming, and Applications: Software for Humanity* (SPLASH '14), pp. 13-15, 2014.
5. R. Brooks, “Towards a theory of the Comprehension of Computer Programs”, *Int’l. Journal of Man-Machine Studies*, Vol. 18, no. 6, pp. 543-554, 1983.
6. E. Soloway and K. Ehrlich, “Empirical Studies of Programming Knowledge”, *IEEE Transactions Software Engineering*, Vol. 10, no. 5, pp.595-609, 1984.
7. S. Letovsky, “Cognitive Processes in Program Comprehension”, *1st Workshop Empirical Studies of Programmers*, Ablex Publishing Norwood, pp 58-79, 1986.
8. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, “Mental Models and Software Maintenance”, *Empirical Studies of Programmers*, Albex, Norwood NJ, pp. 80-98, 1986.
9. A. Lakhotia, “A Unified System for expressing Software Subsystem classification techniques”, *Journal of Systems and Software,* Vol. 36, no 3, pp. 211-231, 1997.
10. T. Kunz and J. P. Black, “Using Automatic Process Clustering for Design Recovery and Distributed Debugging”, *IEEE Transactions Software Engineering*, Vol. 21, no. 6, pp. 515-527, 1995.
11. V. Tzerpos and R. Holt, “Software Botryology: Automatic Clustering of Software Systems”, *Proc. 9th Int’l Workshop Database Expert Systems Applications* (DEXA98) IEEE Computer Society Press, pp. 811, 1998.
12. N. Anquetil and T. C. Lethbridge, “Experiments with Clustering as a Software Remodularisation method”, *Proc. 6th Working Conf. Reverse Engineering* (WCRE 99), IEEE Comp. Soc. Press, pp. 235-255, 1999.
13. R.W. Schwanke and S.J. Hanson, Using Neural Networks to Modularize Software, *Machine Learning*, 15, pp. 137-168, 1994.
14. M. Shtern, V. Tzerpos, “Methods for selecting and improving software clustering algorithms”. *Software Practice and Experience*, Vol. 44, no. 1, pp. 33-46, 2014.
15. C. M. DeOca and D. L. Carver, “Identification of Data Cohesive Subsystems using Data Mining Techniques”, *Proc. Int'l Conf. Software Maintenance* (ICSM 98), IEEE Comp. Soc. Press, pp.16-23, 1998.
16. S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen and E. R. Gansner, “Using Automatic Clustering to Produce High-Level System Organisations of Source Code”, *Proc. 6th Int'l Workshop Program Understanding* (IWPC 98), IEEE Comp. Soc. Press, pp. 45-53, 1998.
17. T. A. Wiggerts, “Using Clustering Algorithms in Legacy Systems Remodularization”, *Proc. 4th Working Conf. Reverse Engineering* (WCRE 97), IEEE Comp. Soc. Press, pp. 33-43, 1997.

18. B.S. Mitchell and S. Mancoridis, “On the Automatic Modularization of Software Systems Using the Bunch Tool”, IEEE Trans. Software Eng., Vol. 32, no. 3, 2006, pp. 193-208.
19. C. Xiao, V. Tzerpos, “Software Clustering on Dynamic Dependencies”, *Proc. 9th European Conf. Software Maintenance and Reengineering* (CSMR 05), IEEE Comp. Soc. Press, pp. 124-133, 2005.
20. K. Sartipi, K. Kontogiannis and F. Mavaddat, “Architectural Design Recovery Using Data Mining Techniques”, *Proc. 2nd European Working Conf. Software Maintenance Reengineering* (CSMR 00), IEEE Comp. Soc. Press, pp. 129-140, 2000.
21. D. Rousidis, C. Tjortjis, “Clustering Data Retrieved from Java Source Code to Support Software Maintenance: A Case Study”, *Proc. IEEE 9th European Conf. Software Maintenance Reengineering* (CSMR 05), IEEE Comp. Soc. Press, pp. 276-279, 2005.
22. Y. Kanellopoulos, C. Tjortjis, “Data Mining Source Code to Facilitate Program Comprehension: Experiments on Clustering Data Retrieved from C++ Programs”, *Proc. IEEE 12th Int’l Workshop Program Comprehension* (IWPC 2004), IEEE Comp. Soc. Press, pp. 214-223, 2004.
23. C. Tjortjis, L. Sinos and P.J. Layzell, “Facilitating Program Comprehension by Mining Association Rules from Source Code”, *Proc. IEEE 11th Int’l Workshop Program Comprehension* (IWPC 03), IEEE Comp. Soc. Press, pp. 125-132, 2003.
24. Y. Kanellopoulos, C. Makris and C. Tjortjis, “An Improved Methodology on Information Distillation by Mining Program Source Code”, Data & Knowledge Engineering, Vol. 61, no. 2, pp. 359-383, 2007.
25. D. Papas and C. Tjortjis, “Combining Clustering and Classification for Software Quality Evaluation”, Lecture Notes Computer Science, Springer-Verlag, Vol. 8445, pp. 273-286, 2014.
26. O. Maqbool, H.A. Babri, A. Karim, M. Sarwar, “Metarule-guided association rule mining for program understanding”, *IEE Proc. Software*, Vol. 152, No. 6, pp. 281-296, 2005.
27. A.T. Misirli, A.B. Bener, B. Turhan, “An industrial case study of classifier ensembles for locating software defects”, *Software Quality Journal*, Vol. 19, No. 3, pp. 515-536, 2011.
28. H. Tribus, I. Morrigl, S. Axelsson, “Using Data Mining for Static Code Analysis of C”, *Proc. 8th Int’l Conf. Advanced Data Mining and Applications* (ADMA 2012), LNAI 7713, pp. 603-614, 2012.
29. M. R. Anderberg, *Cluster Analysis for Applications*, Academic Press Inc., 1973.
30. A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, 1988.
31. I. Witten, E. Frank, M. Hall, and C.J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th Ed., Morgan Kaufmann, 2016.
32. P. H. A. Sneath and R. R. Sokal, *Principles of Numerical Taxonomy*, W. H. Freeman & Co., 1973.
33. B.S. Everitt, S. Landau, M. Leese, D. Stahl, *Cluster Analysis*, 5th ed., John Wiley & Sons, 2010.
34. T. Khoshgoftaar, E. Allen and R. Shan, “Improving Tree-based Models of Software Quality with Principal Components Analysis”, *Proc. 11th Int’l Symposium Software Reliability Engineering* (ISSRE 2000), IEEE Comp. Soc. Press, pp. 198-209, 2000.

35. A.R. Fasolino and G. Visaggio, “Improving Software Comprehension through an Automated Dependency Tracer”, *Proc. 7th Int'l Workshop Program Understanding* (IWPC 99), IEEE Comp. Soc. Press, pp. 58-65, 1999.
36. G. Canfora, A. Cimitile, A. De Lucia and G.A. Di Lucca, “Decomposing legacy systems into objects: an eclectic approach”, *Information and Software Technology*, Vol. 43, pp. 401-412, 2001.
37. T.J. Biggerstaff, B.G. Mitbander and D.E. Webster, "Program Understanding and the Concept Assignment Problem", *Communications of the ACM*, Vol. 37, No. 5, pp. 72-82, 1994.
38. N.E. Gold, “Hypothesis-Based Concept Assignment to Support Software Maintenance”, *PhD. Thesis*, Dept. of Comp. Sc., University of Durham, 2000.
39. D.C. Atkinson and W.G. Griswold, Effective pattern matching of source code using abstract syntax patterns, *Software-Practice and Experience*, Vol. 36 no. 4, pp. 413-447, 2006.
40. M. Choraś, R. Kozik, D. Puchalski, R. Renk, “Increasing product owners’ cognition and decision-making capabilities by data analysis approach”, *Cognition, Technology & Work*, May 2019, Vol. 21, no. 2, pp 191–200.
41. S. Arshad, C. Tjortjis, “Clustering Software Metric Values Extracted from C# Code for Maintainability Assessment”, SETN 16, Article No. 24, ACM Int’l Conf. Proc. Series, 2016.