Data Mining Source Code to Facilitate Program Comprehension: Experiments on Clustering Data Retrieved from C++ Programs

Yiannis Kanellopoulos and Christos Tjortjis
Department of Computation, UMIST, PO Box 88, Manchester, M60 1QD, UK
Y.Kanellopoulos@postgrad.umist.ac.uk, christos@co.umist.ac.uk

Abstract

This paper presents ongoing work on using data mining to discover knowledge about software systems thus facilitating program comprehension. We discuss how this work fits in the context of tool supported maintenance and comprehension and report on applying a new methodology on C++ programs. The overall framework can provide practical insights and guide the maintainer through the specifics of systems, assuming little familiarity with these.
The contribution of this work is two-fold: it provides a model and associated method to extract data from C++ source code which is subsequently to be mined, and evaluates a proposed framework for clustering such data to obtain useful knowledge. The methodology is evaluated on three open source applications, results are assessed and conclusions are presented. This paper concludes with directions for future work.

1. Introduction

A well documented problem faced by maintainers when understanding a software system is the lack of familiarity with it, combined with the lack of accurate documentation [11]. Several techniques and methods have been proposed in order to facilitate this time consuming activity [3], [7], [9].

The work presented in this paper is part of a wider research effort investigating the applicability and suitability of using data mining to facilitate program comprehension and maintenance [4], [13], [15],[15]. This effort aims at developing a methodology for semi-automated program comprehension incorporating data mining. A fundamental underlying assumption is that the maintainer may have little or no knowledge of the program which is analysed.

The work presented here aims to help maintainers to recognise parts of C++ code that have common characteristics, facilitating program understanding. This work focuses on extracting data from C++ code which are clustered in order to identify logical, behavioural and structural correlations amongst program components. C++ was selected as it is widely used but is more complicated to comprehend, compared to other programming languages, like COBOL. As an object oriented language, it can be analyzed in either a more detailed, technical level (member data and member functions analysis), or in a more abstract level (class analysis).

The objectives of this work are:

i) to define the input model needed to extract data from C++ code and populate a database. This requires defining program entities and their attributes.
ii) to propose a pre-processing method that extracts data from code using the input data model.
iii) to assess the feasibility of the methodology in producing valid, useful and novel patterns and knowledge about a software system.

The remaining of this paper is organised as follows. Section 2 reviews previous solutions in the domain of data mining for program comprehension. Section 3 outlines the proposed methodology for pre-processing C++ source code, the input data model and the steps of this methodology. Section 4 assesses the accuracy of the output of this method, analyses its results and outlines deductions from its application. Finally, conclusions and directions for future work are presented in section 5.

2. Background

Software maintenance is the most difficult stage in software lifecycle, often performed with limited understanding of the design and the overall structure of a system because of commercial pressures [11]. Fast, unplanned modifications, based on partial understanding of a system, give rise to increased code complexity and deteriorated modularity, thus resulting in 50%-90% of the maintainers' time to be spent on *program comprehension* [14]. Furthermore it is recognised that there are no

explicit guidelines given a program understanding task, nor there are good criteria to decide how to represent knowledge derived by and used for it [2].

Data mining and its ability to deal with vast amounts of data, has been considered a suitable solution in assisting software maintenance often resulting in remarkable results [1], [6], [8], [10], [12], [17], [17]. Our approach similarly uses data mining to get insights into systems design and structure [4], [13], [15], [15].

The following paragraphs briefly review some of the most prominent solutions in the area of data mining for software maintenance and compare these to our approach.

2.1 Using Clustering to Produce High-Level System Organisations of Code

This solution proposes a collection of algorithms which facilitate the automatic recovery of the modular structure of a software system from its source code [8]. It creates a hierarchical view of the organisation of the system based mainly on the components and the relationships that exist in the source code.

First it represents the system modules and the module-level relationships as a module - dependency graph. Then it partitions this graph so that the high - level subsystem structure can be derived from the component level relationships extracted from the source code. Based on the concepts of cohesion and coherence three parameters are introduced: intra-connectivity, inter-connectivity and modularisation quality.

The basic goal of this modularisation technique is to automatically partition the components of a system into clusters (subsystems) so that the resultant organisation concurrently minimises inter-connectivity while maximising intra-connectivity. The underlying assumption is that a well-designed system is organised into cohesive clusters that are loosely interconnected. The main drawback of this solution is that as the number of files exceeds 20, calculation time is greatly increased.

2.2 A Software Evaluation Model Using Component Association Views

This solution proposes a model for the evaluation of the architectural design of a system based on the association between the components of the system [12]. It allows measurement of system modularity, as an indication of the design quality and its decomposition into subsystems. For this reason the following three association views of a system are generated:

i) Control passing: It represents the correlation among the system components based on function invocation.
ii) Data exchange: It epitomises the correlation among the system components based on aggregate data types (except integer, real, boolean and string) that are either passed as parameters between two functions or are referenced by a function.
iii) Data sharing: It signifies the correlation among the system components based on sharing the global variables by the functions.

In this approach the software system is modelled as an attributed relational graph with system entities as nodes and data-control-dependencies as edges. At this point, the application of data mining techniques, like association rules helps the decomposition of the graph into domains of entities based on the association property. The next step is to populate a database of these domains. This approach is based on the concept of the association between the components of a system. There are however other characteristics that play crucial role in grouping system components, such as the number of member data or functions in a class. These can be discovered by using other data mining techniques like clustering.

2.3 A Method for Legacy Systems Maintenance by Mining Data Extracted from Code

This approach used data mining to facilitate software maintenance and reliability assessment. It addressed C/C++ and COBOL legacy systems aiming at understanding low/medium level concepts and relationships involving components at the function, paragraph or even line of code level [4], [13], [16].

This approach consists of three distinct phases: a) data extraction, b) data mining application and c) result evaluation. There were different challenges in each phase. These involved the definition of an appropriate data model which captures as much information about the code as possible, the construction of a database suitable for data mining, the selection and customisation and application of data mining algorithms and the assessment of the outcomes by domain experts. The approach deals with both COBOL and C/C++ programs and varies according to the differences between these languages.

For C programs, we used functions as entities, and attributes defined according to the use and types of parameters and variables, and the types of returned values. We then applied clustering to identify sub-sets of source code that are grouped together according to custom-made similarity metrics.

For COBOL programs we used paragraphs as entities, and binary attributes depending on the presence of user-defined and language-defined identifiers. In this case we derived association rules in order to establish inter-group and intra-group relationships.

Results represent the syntactic and semantic content of the source code. Code is represented by means of models or graphs, like variable relationship model, a variable-block relationship model or even models that convey a meaning similar to Data Flow Diagrams and flow charts. Programs are abstracted into groups containing interrelated entities and are grouped together.

This solution addresses systems both at medium and at low level and confirms that data mining can produce structural views of source code thus facilitating legacy systems understanding. There were however issues that had to do with correlations across system components such as programs and files. This deficiency was dealt with by the methodology proposed in this paper.

3. Description of the Proposed Framework

The framework proposed here, was developed for pre-processing C++ source code at the program level and consists of the following parts:

i) The input model, which involves the specification of program entities and their attributes.
ii) The pre-processing method.

This section outlines the main characteristics of the pre-processing and cluster analysis system.

3.1 Input Model

The definition of the input model requires the specification of program entities and their attributes. This facilitates clustering, a data mining technique, which imposes requirements on the type and number of attributes, the lack of a distinction between predictive and predicted attributes and so on [5].

Entities were selected to be applicable to all programs. That means that they need to be clearly named in a program. For instance, functions and classes are candidate entities as they exist in most C++ programs. Candidate entities also need to model a large proportion of the code, thus ensuring that any subsequent analysis covers a large part of a system.

Moreover entities need to contain a common set of attributes in order to achieve homogeneity. This allows for entities' comparison on the basis of their attributes, which is the main principle of clustering. The number of selected attributes need also be sufficient in order to avoid misleading comparisons or discovering accidental similarities. Selected attributes were both binary and qualitative, as they are predominant in a source code application domain [4].

Four types of entities were eventually selected: Classes, member functions, parameters and member data. The following schema (Fig. 3.1) outlines the proposed input model in terms of entities and respective attributes. Each entity is described by attributes thus formulating database tables.

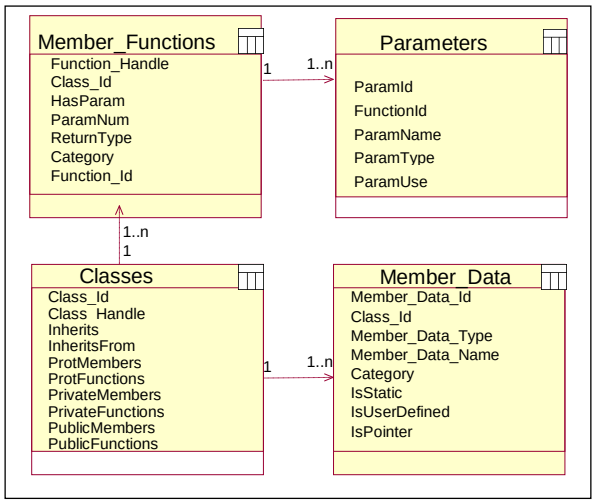

Fig. 3.1: The proposed input model

3.2 Pre-Processing Methodology

The pre-processing method extracts data from source code and stores these into appropriate tables. There were two major requirements for this:

i) Output should be stored in way facilitating clustering
ii) Data processing should be fast.

We use a top-down approach by processing top-level program data first, such as class information, and then lower-level data such as member functions, their parameters, and member data.

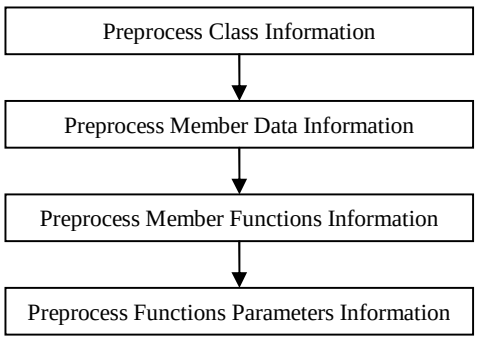

Fig. 3.2: Pre-processing methodology

More specifically, we first extract information that describes the class entity, such as class handle, its super-class name if it exists, and the number of the member data and functions. Then information that describes the member data of a class is extracted, including variable name, type and category, such as public, protected, private, as well information describing whether the

variable is static, a pointer or user-defined. After that, we extract information related to class member functions, such as name, return type and category (public, protected, private), as well as number of parameters if any. Finally information related to the parameters of class member functions is extracted including name, type and use (by value, by reference). An outline of this methodology is illustrated in Fig. 3.2.

3.3 Aspects of the Proposed Framework

Sections §3.1-.2 presented two fundamental concepts about the proposed framework: the input model and the pre-processing methodology. This section describes aspects of the framework regarding outcome utilisation. More specifically all the information required by the methodology, as defined by the input data model, can be found at the header files of standard C++ systems. We scan these files and populate relevant database tables. We then use IBM's Intelligent Miner™ demographic clustering on these tables to identify patterns concerning the system structure and its components' general characteristics. These characteristics can be qualitative like the name of the superclass that a class inherits from, the category (public protected, private) of a member function and so on. They can also be quantitative, such as the number of member function parameters. We have experimented using various clustering schemes in order to identify correlated entities such as classes, functions and member data, based on similarities on their attributes as defined by the input model. Results are briefly presented and discussed in section 4.

3.4 Technical aspects of the Framework

This section provides a description of the technical aspects of the framework and the system developed to implement it. The main parts of this system are:

i) The Pre-processing Application that parses C++ code. This consists of a G.U.I front-end and a pre-processing back-end that extracts data from code and stores them in a database.
ii) The database management system that stores data to be mined.
iii) The data mining clustering tool.

Fig. 3.3 highlights the structure of this system. At first, the pre-processing application parses the C++ code in order to extract the data defined by the input model described in §3.1. The next step is using a data mining tool to perform clustering. The system also includes clustering results analysis.

All in all, our approach used a customised process model widely used in Knowledge Discovery in Databases (KDD). The main difference is that the original data collection is made of source code rather than more "conventional" tuples [5]. Fig. 3.4 illustrates this process.

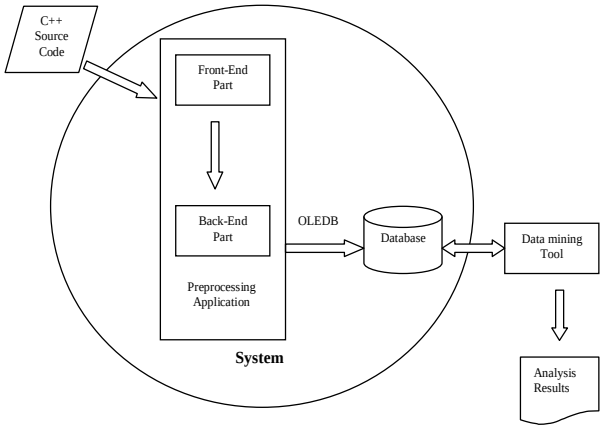

Fig. 3.3: An overview of the system

The first step of this process (*Extraction*) involves parsing the code to extract data modeling program entities and their attributes. The second step (*Transformation*) transforms the extracted data in order to store these in relational database tables suitable for clustering. The third step (*Data Mining*) applies clustering in search for patterns of interest. Patterns are then interpreted and analyzed. The process is an iterative one and interim results or findings can be feedback to a previous stage.

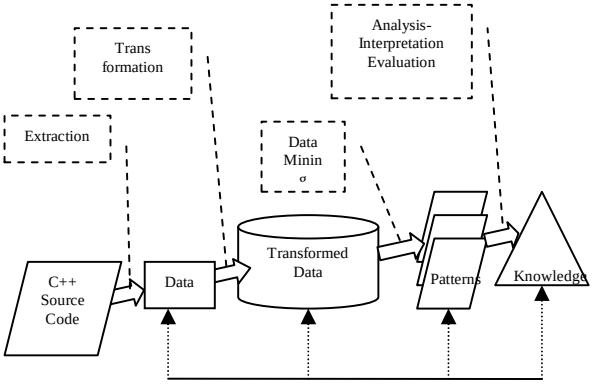

Fig. 3.4: Knowledge discovery in source code

4. Result Evaluation

The proposed framework was evaluated in terms of accuracy and ability to capture knowledge relevant to software maintenance activities, using three open source applications. Two of the applications, CAccessReports and CompDB, are created with the help of Microsoft Foundation Classes (MFC) and can be downloaded from [19]. The other application, FlightGear Flight Simulator, is an open source flight simulator that can be downloaded from [19]. The actual structure of these applications is

compared to the outcome of the analysis of their respective input models. The output should be valid, novel and useful to the system maintainer. The following sub-sections discuss separately the outcomes of our empirical experimentation with these applications.

4.1 The First Case Study

CAccessReport is a small-medium size application with 53 public classes, and 2812 functions that have 1614 parameters in total.

4.1.1 Class Analysis. The classes of this application have many similarities as almost all of them (52 out of 53) inherit from one class: COleDispatchDriver and have only public member functions. Therefore, only attributes describing the number of public functions and the class handle were of importance in formulating clusters. As a result clusters are characterised only by the number of their member functions.

4.1.2 Member Functions Analysis. There are two significant characteristics of the member functions of this program: the first is that all of them are public and the second is that almost half of them contain parameters. They were grouped in three clusters.

The first cluster, representing 45.82% of the population, consists of public functions with parameters. These functions either have no return type or they return void. Therefore, it can be concluded that this cluster includes the constructors of the system's classes and functions that usually set values to these. The second cluster, representing 34.12% of the population, consists of public functions that have no parameters. Half of these functions return the type CString, which encapsulates a character string. The third cluster, representing 20.06% of the population, consists of public functions, 11.17% of which have no parameters at all, while the remaining 88.83% have. Almost half of these functions return the following types VARIANT, which is a self-described data type that facilitates data passing, and LPDISPATCH, which accesses the underlying pointer of the COleDispatchDriver object.

4.1.3 Member Functions Parameters Analysis. Most of the member function parameters are passed by value. They were grouped in three clusters:

The first cluster (42.44% of the population) consists of parameters that are passed by value and originate from the following types: LPCSTR, which is a constant pointer to a string, LPDISPATCH and pointers of type VARIANT. The second cluster (41.57%) also consists of parameters that are passed by value, most of which originate from the types bool, short and long. The third cluster (15.99%) consists of parameters that are passed by reference and they are constants of type VARIANT. Constants of this type are used by functions which aim to specify data that cannot be passed by reference in any other way.

4.1.4 Conclusions. There are no major differences amongst components of this program. The majority of its classes inherits from the same superclass, has no member data and member functions are public. Differences are mainly quantitative, such as the number of public functions or parameters. The only differences that are qualitative rather than quantitative are attributed to the member functions return types of and their parameters' types. Interestingly, every parameter passed by reference is a constant of type VARIANT. Thus, it can be deducted that functions using such parameters are bound during run-time and that data are not known in advance.

4.2 The Second Case Study

CompDB is a small size application. It consists of 18 public classes, 64 member data and 256 functions that have 235 parameters.

4.2.1 Class Analysis. Entities extracted from this program formed three clusters.

The first cluster represents 38.89% of the population, and consists of classes that all inherit from another class. Their respective superclasses are:

i) CStatic, which encapsulates the static control.
ii) CView, which a view class is derived from.
iii) CMDIFrameWnd, which provides a main frame window for Multiple Document Interface (MDI) applications.
iv) CMDIChildWnd, which provides child windows for an MDI application.

Classes in this cluster are related logically, as they represent components of the document/view architecture implemented by this program.

The second cluster, represents 33.33% of the population, and consists of classes, amongst which, two do not inherit and four do. The respective superclasses of those who inherit are:

i) CStringArray, which is an array of the String type.
ii) CListBox, which encapsulates the list box control.
iii) CDocument, which is the class where the document of an MFC application (like CompDB) derives from.
iv) CListCtrl, which displays a graphical list items.

The classes in this cluster do not have a strong logical correlation. There is only one class representing a component included in the document/view architecture, two others represent control classes, and another represents a shape of the MFC collection.

The third cluster represents 27.78% of the population, and consists of classes that all inherit. Their respective superclasses are:

i) CDialog, which implements Windows dialogs.
ii) CButton, which is a standard Windows pushbutton.
iii) CWinApp, which represents the standard Windows Application.

The classes in this cluster do not have a strong logical correlation. There are three classes representing dialog controls, one represents the Windows application and another represents a control.

4.2.2 Member Data Analysis. The member data of this program's classes are either public or protected. Three clusters were formed.

The first cluster represents 59.38% of the population and consists of protected members, none of which is a pointer. Almost half of the member data of this class (Fig. 4.1) belong to two classes. The types of the member data vary. The more predominant are:

i) int
ii) CString, which encapsulates a character string.
iii) CFont, which wraps the Windows font object and API functions for creating and managing fonts.
iv) CGridCtrl, which is a control.

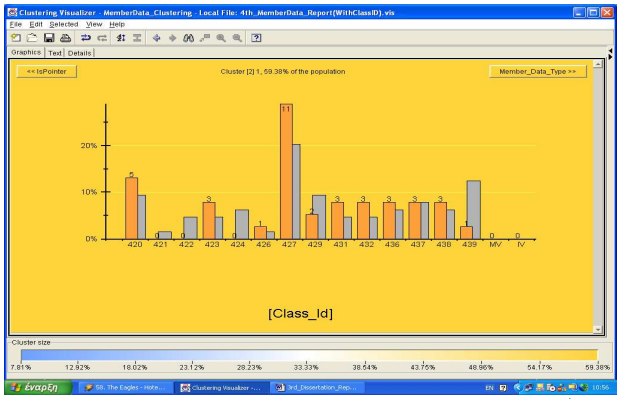

Fig. 4.1: Member data classes of CompDB, 1st cluster

The second cluster represents 32.81% of the population and consists of public members none of which is a pointer. Member data of this cluster mostly belong to three classes. There is a clear logical connection among member data of this cluster as the majority of it belongs to classes that are derived from the CDialog class.

Types of the member data vary. The more predominant are:

i) enum
ii) CString, which encapsulates a character string.
iii) CButton, which wraps a standard Windows pushbutton.

The third cluster represents 7.81% of the population and consists of public and protected members which are all pointers. This is the most important logical relation between the member data of this cluster, which only belongs to two classes (Fig. 4.2).

The types of the member data are different. The more predominant are:

i) CPen, which wraps the Windows pen object and includes API functions for creating pens as member functions.
ii) CBrush, which wraps the Windows brush object and API functions for creating brushes.

4.2.3 Member Functions Analysis. Class member functions were grouped in three clusters.

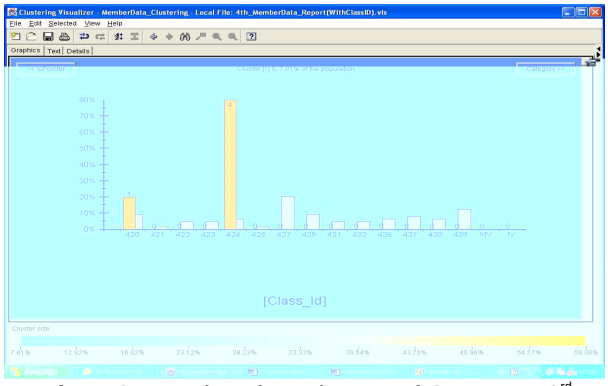

Fig. 4.2: Member data classes of CompDB, 3rd cluster

The first cluster represents 34.38% of the population and consists of public and protected functions. The return types of these vary, the most predominant being afx_msg void and afx_msg int. Most of the member functions of this cluster belong to four classes. The second cluster represents 33.20% of the population. The return types of these vary, the most predominant being void and bool. Most of the member functions of this cluster belong to four classes, two of which are the same as in the first cluster. The third cluster represents 32.42% of the population. Almost half of the functions of this cluster do not have a return type. This indicates that they are either the constructors or the destructors of the classes they belong to. Among member functions that have a return type, the most predominant one is bool. Most of the member functions of this cluster belong to four classes, three of which are the same as in the second cluster and only one similar to these in the first cluster.

4.2.4 Member Functions Parameters Analysis. Member function parameters of classes were grouped into three clusters:

The first cluster represents 42.98% of the population and consists of parameters passed by value. The return types of these vary, the most predominant being pointers of type char, int and pointers of type CDC, which is a class that encapsulates device-context support. The second cluster represents 42.55% of the population and consists of parameters passed by value. The return types of these vary, the most predominant being char, UINT, which is an unsigned 32-bit integer and COLOREF, which is a 32-bit integer that holds a colour. The third cluster represents 14.47% of the population and consists of parameters passed by reference. The return types of these vary, the most predominant being CDUMPCONTEXT, which is a class that its objects provide several diagnostic messages and _CONNECTIONPTR, which is a class that its objects are pointers to a Connection Interface.

4.2.5 Conclusions. We found the following logical correlations among classes of this program: first, there are four classes deriving from the class CView. These classes have also structural (member data) and behavioural (member functions) similarities. They have the same size as there are no significant differences between the number of their member data and functions. Their member data have also common data types. Their member functions have also common return types and similar parameter numbers.

Another category of logically related classes are two classes that are derived from CWnd. They also have structural similarities, as their member data have common data types. However, their member functions seem to have similarities but in a lower rate than the member data.

Three classes that are derived from the class CDialog are also logically related and present structural and behavioural similarities. Their member data and functions have many similarities, such as common data types (for the member data), number of parameters and return types (for the member functions).

Overall we identified logical, structural and behavioural correlations involving 9 out of 18 classes of this program. This can be useful for maintenance purposes. For example, if changes in the data type of a member data of one class are to be made, then related classes should be looked at as well. We may also presume that by finding logical correlations amongst classes of a system, it is also likely to find structural and behavioural correlations. This can be useful for software understanding and maintenance purposes. Classes' ability to inherit is an important factor affecting the logical correlation within a system. If some classes share the same superclass, then it is more likely to have similar structure and behaviour.

At this point it has to be underlined that finding logical correlations between the classes of a system is not the only way for a maintainer to understand the system. There are classes that do not have any logical correlation, but they can have either structural or behavioural similarities. Examples of this are classes which have member data of the same type (structural similarity). Therefore, if one is to change the type of the member data of one, it is likely that a similar change will be needed for the other as well.

4.3 The Third Case Study

FlightGear Flight Simulator is a medium size application with 147 classes, 1450 member data and 2155 functions that have 1614 parameters in total.

4.3.1 Class Analysis. Entities extracted from this program formed 5 clusters.

The first cluster represents 46.26% of the population, and consists of classes that do not inherit. By their handles we understand that there is a logical correlation among these, as most belong to the three system modules. The second cluster represents also 46.26% of the population and consists of classes that do inherit. Their respective superclasses are: FGSubSystem, which provides an interface that all subsystems should implement, FGInterface and instr_item, which represents an aircraft instrument. The third cluster represents 3.12% of the population and consists of classes that all inherit. Their respective superclasses are FGATC, which is the base class for the various actual classes, and fgCallback, which is a wrapper class that treats method and function pointers as objects. The fourth cluster represents 2.34% of the population and its basic characteristic is that the classes that belong to it do not have any protected members and most of them inherit from FGElectricalComponent, which is a base class for other electrical components. The last cluster represents 2.02% of the population and consists of classes that do not inherit and have more than 27 public function members.

4.3.2 Member Data Analysis. Clustering member data entities formed three clusters.

The first cluster represents 38.34% of the population and consists of private member dataone third of which are pointers. Most of them belong to class FGControls, which defines a standard interface to all flight simulation controls. The types of the member data vary. The more predominant are Bool, Int, SGPPropertyNode and Float.

The second cluster represents 35.66% of the population and consists of private data members, almost none of which are pointers. Most of them belong to classes FGInterface, which defines shared flight model

parameters and FGNavCom, which is a class that manages navigation communications instances. Member data types vary, the more predominant being Double, SGPPropertyNode and FG_VECTOR_3.

The third cluster represents 26.00% of the population and consists of public and protected member data most of which belong to classes FGILS, FGDME, FGApproach and fgLIGHT. The types of data members also vary and the more predominant are Double, Bool and SGPPropertyNode.

4.3.3 Member Functions Analysis. Class member functions were grouped in three clusters.

The first cluster represents 54.48% of the population and consists of mostly public member functions without parameters. Return types of these vary, the most predominant being double and void. Most of the member functions of this cluster belong to seven classes.

The second cluster represents 42.41% of the population. It consists of public classes that have parameters. Return types of these vary, the most predominant being void and bool. Most of the member functions of this cluster belong to two classes, FGInterface, which defines shared flight model parameters, and FGNavCom, which is a class that manages navigation communications instances. These classes are identical to these that class member data of the second cluster belong. This is depicted in Fig. 4.3-4.4.

The third cluster represents 3.11% of the population and consists of private member functions that have parameters. That is why it represents such a small percentage of the population. Almost half of the functions of this cluster return void. Most of the member functions of this cluster belong to four classes.

4.3.4 Analysis of the Parameters of Member Functions. Member function parameters of classes were grouped in three clusters:

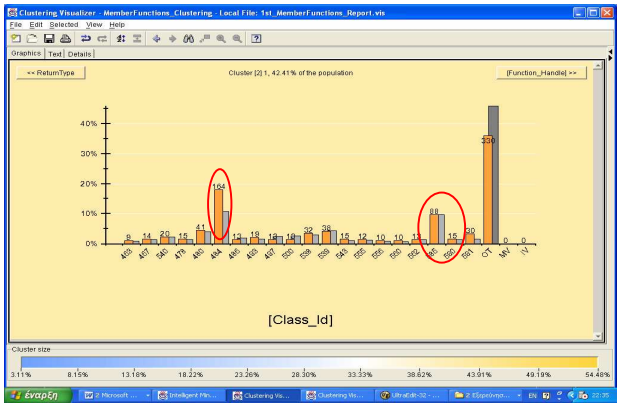

Fig. 4.3: Member function classes in the 2nd cluster

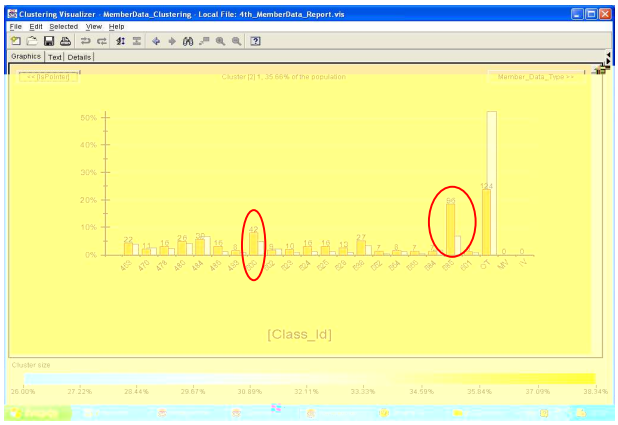

Fig. 4.4: Member data classes in the 2nd cluster

The first cluster represents 46.10% of the population and consists of parameters passed by value. The return types of these vary, the most predominant being double and bool. The second cluster represents 45.16% of the population and consists of parameters passed by value. The return types of these vary, the most predominant being Int, string and const WeatherPrecision, which is a user-defined type. The last cluster represents 8.74% of the population and consists of parameters passed by reference. The return types of these vary, the most predominant being Const String, Const Double and Int.

4.3.5 FlightGear Flight Simulator: Conclusions. We found several types of correlations amongst classes of this program. The first correlation is among classes that belong to the same module. They have both structural (member data) and behavioural (member functions) similarities, as their member data have common data types and their member functions common return types and parameter numbers like the previous category. For example classes hud_card and HudLadder belong to the Cockpit module. There are also classes that present either structural or behavioural similarities. For example classes' FGNavCom and FGAD member data have similarities, such as common data types. On the other hand, their member functions are dissimilar as their number of parameters and return types differ.

As mentioned in §4.2.5, class inheritance is also an important factor affecting logical correlations within a system. If some classes share the same superclass, then they are likely to have similar structure and behaviour. For example, classes FGInterface and FGNavCom, which inherit from class FGSubsystem, do not belong to the same module but present both structural and behavioural similarities, as their member data has common data types and their member functions have also common return types and similar parameter numbers. Figure (Fig. 4.5) depicting the system's base classes illustrates this point.

Here, the most important base class is FGSubsystem as it has the largest number of sub-classes. Therefore, any change in this class may impact on large part of the system.

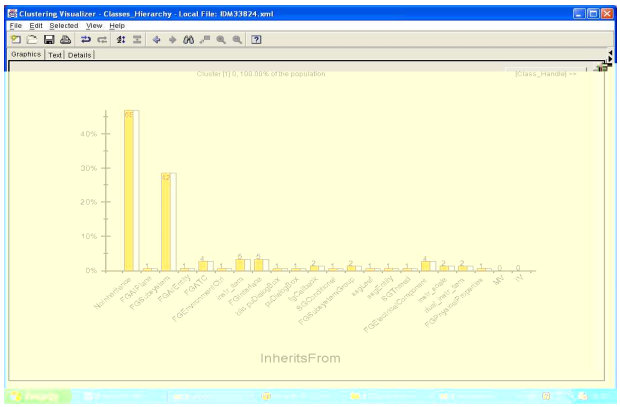

Fig. 4.5: Superclasses of the FlightGear

It is also clear (Fig. 4.5) that almost half of the system classes do not have a superclass, which indicates that finding logical correlations among system classes is not the only way for a maintainer to understand the system. There are classes that do not have any logical correlation, but they can have either structural or behavioural similarities or both as it was also presented in §4.2.5 (conclusions section of CompDB application). All in all conclusions drawn on FlightGear Flight Simulator's classes confirm these presented in §4.2.5.

5. Conclusions and Future Work

This section presents conclusions drawn by evaluating the proposed methodology and comparing it to similar ones. Directions for future work are also discussed.

5.1 Conclusions for the Framework

Firstly, the proposed framework uses clustering and the respective input model that supports it to assist program comprehension. It is specifically designed for C++ code and was tested using various sizes of datasets. Three open source systems were used as case studies: the first with 53 files of 3600 LOC and 224KB in total, the second with 18 files of 500 LOC and 70KB in total, and the last with 147 files of 5000 LOC and 512 KB in total.

The proposed solution, is semi automated unlike the one at §2.1. As soon as the pre-processing method terminates, a separate data mining clustering tool is used. The strength of this framework is that it is compatible to existing clustering tools making the customisation of clustering algorithms unnecessary. Data are stored in a database which can be easily connected with any data mining tool regardless of the technique to be used (clustering, association rules and so on). This framework uses only clustering unlike the solution in §2.2; what is more, low and medium level relations, such as among functions and classes respectively, can be identified, according to the level of detail that the user requires.

5.2 Conclusions for the Pre-Processing Methodology

An important part of the proposed framework consists of the input model and the pre-processing methodology. Various requirements were set and met, as indicated by the empirical evaluation. Data from source code, the header files in particular, are retrieved and stored in a way facilitating clustering. Suitable attributes were selected in order to avoid an unnecessary increase of records' similarity. For instance, the *Category* attribute of the *Member Data* and *Classes* entities was used in order to define the category of the data members and the classes. An earlier design catered for three binary attributes instead of a single one.

The methodology can process a substantial number of files reasonably fast. For example it took less than 15 minutes to process 53 header files, of a total size of 224KB, of the CAccessReport application. Finally, clustering outputs indicated the validity of the approach. Patterns found reflect the structure of all three examined systems. Such patterns are also useful to a maintainer when attempting system understanding. For example, it is likely to observe structural and behavioural correlations by discovering logical correlations amongst classes. However, discovered patterns were not entirely novel. Some of them have been theoretically described before, but were verified by clustering nonetheless. A maintainer might have expected results like in the previous example. An unexpected, novel pattern identified was that either structural or behavioural similarities can be found in classes that are not logically correlated, perhaps reflecting programming styles.

5.3 Future Work

We consider the following various alternatives in order to enhance the proposed framework:

Enriching the input model and improving the algorithm

The proposed framework processes information derived from header files only. In order to achieve better understanding of a system, it might be better to process elements from the *.cpp files also, such as:

i) The names of the included header files that are referenced at the beginning of each *.cpp file. It

could be another way to discover logical correlations amongst the classes of a system, other than inheritance.

ii) The definitions of the constants that are used in the *.cpp file. This can help to find more structural similarities among classes of a system, even in case they do not present any logical correlations.

Integration of data mining algorithms

The proposed methodology does not integrate any data mining algorithms. It pre-processes C++ code data and uses existing commercial tools to perform clustering. However it may be useful if custom data mining algorithms were integrated in this framework. This would result in a complete system for automated program and system comprehension. On the other hand that would deprive the framework of its current flexibility and adaptability.

Using other data mining techniques for better program comprehension

The input model is designed to facilitate clustering, which aims at grouping records together based on their similarity. However, using other data mining techniques, which can discover other types of patterns, may give a more complete picture of a program. For instance, association rules identify items in a record that imply the presence of other items in the same record.

References

[1] N. Anquetil and T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization method", *Proc. 6th Working Conf. Reverse Engineering* (WCRE 99), IEEE Comp. Soc. Press, Oct. 1999, pp. 235-255.

[2] F. Balmas, H. Wertz and J. Singer, "Understanding Program Understanding", *Proc. 8th Int'l Workshop Program Comprehension* (IWPC 00), IEEE Comp. Soc. Press, 2000, pp. 256.

[3] G. Canfora, L. Mancini, and M. Tortorella, 'A Workbench for Program Comprehension during Software Maintenance', *Proc. 4th Int'l Workshop on Program Comprehension* (IWPC 96), IEEE Comp. Soc. Press, 1996, pp. 30-39.

[4] K. Chen, C. Tjortjis and P.J. Layzell, "A Method for Legacy Systems Maintenance by Mining Data Extracted from Source Code", *Case studies of IEEE 6th European Conf. Software Maintenance and Reengineering* (CSMR 02), 2002, pp. 54-60.

[5] U. M. Fayyad, G. Piatetsky-Shapiro and P. Smyth. "From Data Mining to Knowledge Discovery: An Overview", in *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996.

[6] T. Kunz and J. P. Black, "Using Automatic Process Clustering for Design Recovery and Distributed Debugging", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, June 1995, pp. 515-527.

[7] P. Linos, Z. Chen, S. Berrier, and B. O'Rourke, "A Tool For Understanding Multi-Language Program Dependencies", *Proc. IEEE 11th Int'l Workshop Program Comprehension* (IWPC 03), IEEE Comp. Soc. Press, 2003, pp. 64-72.

[8] S. Mancoridis, B.S. Mitchell, Y. Chen and E.R. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures", *Proc. Int'l Conf. Software Maintenance* (ICSM 99), IEEE Comp. Soc. Press, 1998, pp. 50-59.

[9] A. Von Mayrhauser and A.M. Vans, 'Program Understanding Behavior During Adaptation of Large Scale Software', *Proc. 6th Int'l Workshop Program Comprehension* (IWPC 98), IEEE Comp. Soc. Press, 1998, pp.164-172.

[10] C.M. de Oca 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, 1998, pp.16-23.

[11] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, Wiley Computer Publishing, 1996.

[12] 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, 2000, pp. 129-140.

[13] C. Tjortjis and P.J. Layzel, "Using Data Mining to Assess Software Reliability", *Suppl. Proc. IEEE 12th Int'l Symposium Software Reliability Engineering* (ISSRE 01), 2001, pp. 221-223.

[14] 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, 2001, pp. 281-287.

[15] C. Tjortjis, N. Gold, P.J. Layzell and K. Bennett, "From System Comprehension to Program Comprehension", *Proc. IEEE 26th Int'l Computer Software Applications Conf.* (COMPSAC 02), IEEE Comp. Soc. Press, 2002, pp. 427-432.

[16] 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, 2003, pp. 125-132.

[17] V. Tzerpos and R. Holt, "Software Botryology: Automatic Clustering of Software Systems", *Proc. 9th Int'l Workshop Database Expert Systems Applications* (DEXA 98), IEEE Comp. Soc. Press, 1998, pp. 811-818.

[18] T. A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization", *Proc. 4th Working Conf. Reverse Engineering* (WCRE 97), IEEE Comp. Soc. Press, 1997, pp. 33-43.

[19] http://www.codeguru.com/mfc_database/

[20] http://www.flightgear.org