JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets © Jot, 2012
Online at http://www.jot.fm

A Seeded Search for the Modularisation of Sequential Software Versions

Mahir Arzoky[a] Stephen Swift[a] Allan Tucker[a]

James Cain[b]

a. School of Information System, Computing and Mathematics, Brunel University, Uxbridge, UK

b. Quantel Limited, Newbury, UK

Abstract Software module clustering is the process of partitioning the structure of the software system using low-level dependencies in the source code in order to understand and improve the system's structure. A software clustering tool, Munch, was used to modularise sequential source code software check-ins to assess the degree of major changes. It uses a search-based software engineering technique. This paper employs a seeding technique, based on results from previous modularisations, to speed up the process and reduce the running time. In order to evaluate the efficiency of the modularisation we conducted a number of experiments on our uniquely large and comprehensive real-world dataset. The results of the experiments present strong evidence to support the seeding strategy.

Keywords clustering; modularisation; refactoring; seeding; time-series; fitness function; EVM; EVMD.

1 Introduction

Large software systems tend to have complex structures. Their structures are often difficult to comprehend due to the large number of modules and inter-relationships that exist between them. As systems continue to evolve, their structure becomes more complex and harder to track. Thus, software systems need to be regularly maintained to cope with constantly evolving requirements.

Refactoring is a common technique that can be used to improve the internal attributes of a system to make it easier to maintain without changing its external behaviour [Fowle99] [Strogg07]. Refactoring can improve maintainability, enhance performance and reduce the complexity of certain code units, if applied correctly. Unfortunately, it is not practical to refactor a software system without taking into account the cost and deadlines of the project. There is significant value in being able to predict where refactoring occurs. Predicting the

Mahir Arzoky, Stephen Swift, Allan Tucker, James Cain. A Seeded Search for the Modularisation of Sequential Software Versions. In Journal of Object Technology, vol. 11, no. 2, 2012, pages 6:1–22. Available at doi: 10.5381/jot.2012.11.2.a6

likely changes a system will undergo, based on previous development time, makes it possible to estimate developer effort required and to allocate resources appropriately.

As the modular structure of a software system tends to decay over time, it is important to modularise. Modularisation is the process of partitioning the structure of the software system into subsystems. Subsystems group together related source-level components in order to assist with understanding the system. They consist of source code resources that provide a service to part of the system. Subsystems include resources such as modules, classes and other subsystems. Subsystems can be organised hierarchically in order to allow developers to navigate through the system at various levels of details. They can facilitate program understanding. Modularisation also makes the problem at hand easier to understand, as it reduces the amount of data needed by developers.

However, poorly partitioned software is widely considered to be a source of problems for understanding [Consta79]. Due to the complexity associated with understanding source-level components of a system, manual decomposition of a software system into meaningful subsystems can be a time-consuming process. To address this issue, fast and effective tools that automatically decompose a software system into a set of meaningful subsystems were developed. Automated tools are used to generate useful information on system structure. These tools analyse low-level dependencies in the source code and cluster them into meaningful subsystems.

Directed graphs can be used to make the software structure of complex systems more comprehensible [Mitch02]. Software structure can be depicted as one or more directed graphs. Directed graphs can be described as language-independent, whereby components such as classes or subroutines of a system are represented as nodes and the inter-relationships between the components are represented as edges. Such graphs are referred to as Module Dependency Graph (MDG). However, creating an MDG of the system does not always make it easy to understand the system's structure. Thus, graphs could be partitioned to make them more accessible and easier to understand. Dependence information from system source code is used as input information. A file is considered as a module and the reference relationship between files is considered to be a relationship. This is known as the software clustering problem. Mancoridis et al [Manco98] were the first to use MDG as a representation of the software module clustering problem.

Creating meaningful partitions of an MDG is not an easy task as the number of possible partitions can be very large, even for smaller systems. In addition, a small difference between two partitions can produce very different results [Manco99]. A good partition of a system would produce independent subsystems that contain highly interdependent modules. Clustering helps developers to better understand the structure of complex systems by providing them with a high level view of the system structure. It is widely considered that it is easier to develop and maintain well-modularised software systems [Consta79].

For various search algorithms [Micha00], search-based software engineering has been shown to be highly robust. There have been a large number of studies [Harma02] [Harma05] [Manco02] [Mitch02] using the search-based software engineering approach to solve the software module-clustering problem. In previous studies, techniques that automatically cluster a system's MDG were introduced. They treat clustering as an optimisation problem, in order to find good partitions. A number of various heuristic search techniques, including Hill Climbing, Genetic Algorithms and Simulated Annealing were used to explore the large solution space of all possible partitions of an MDG.

The aim of the paper is to perform efficient modularisation on source code check-ins, taking advantage of the fact that the dataset is time-series. The nearer the source code in time, the more similar they are expected to be, and thus the more similar the modularisation is expected to be. We are not treating our dataset as separate modularisation problems, but instead we are using the previous results of modularisation to give us a head start. This

paper extends the work of Arzoky et al [Arzok11], which introduced the seeding technique to modularise the time-series dataset. Although a few studies have looked at using the concept of seeding for clustering, for example [Sures10] [Swift04], none have looked at using seeding to modularise sequential source code software versions. The aim is to use code structure and sequence to obtain more effective modularisation and also to locate the possible occurrence of major changes, in particular refactorings.

The paper is organised as follows. Section 2 describes the experimental methods, which includes the implementation of heuristic search algorithms, fitness functions used, and metrics used to measure the quality and similarity of the modularisation. Section 3 describes the subject systems and the experimental procedure. Section 4 discusses the results and presents trends and characteristics observed. Finally, section 5 draws conclusions and outlines future work for the project.

2 Experimental Methods

This section explores the clustering algorithm and the fitness function implemented, describes an external coupling metric used for measuring the modularisation's quality and also explains an agreement metric used for comparing two clustering arrangements.

2.1 Clustering Algorithm

This work significantly extends that of Arzoky et al [Arzok11] and follows Mancoridis et al and Mitchell [Manco02] [Mitch02], who first introduced search-based approach to software modularisation. The clustering algorithm was re-implemented from available literature on Bunch's clustering algorithm [Manco98] to form a tool called Munch. Munch is a rapid prototype implemented to carry out experimentations of different heuristic search approaches and fitness functions. Munch's output is a hierarchical decomposition of the system structure, whereby closely related modules are grouped into clusters that are loosely connected to other clusters.

A heuristic algorithm is required to traverse the space of possible solutions using the fitness function in order to locate the best solution. It uses an MDG as an input and produces a partition of the MDG as an output. It partitions the system into clusters. A cluster is a set of the modules in each partition of the clustering. The software module clustering problem involves finding good quality software modules clusters based on the relationships amongst the modules. It aims to produce a graph partition that minimises coupling between clusters and maximises cohesion within each cluster. Coupling is defined as the degree of dependence between different modules or classes in a system, whereas cohesion is the internal strength of a module or class [Somme95].

The clustering algorithm uses a simple random mutation Hill Climbing approach [Micha00] to guide the search. It is a simple, easy to implement technique that has proven to be useful and robust in terms of modularisation. It was chosen for this paper as it has performed best in reported recent studies. It has outperformed other algorithms in terms of both quality of the solutions and execution time [Pradi11].

In Hill Climbing, the search process starts from a randomly chosen representation. Modules are rearranged to find better clustering arrangements with a higher fitness function value. Once a 'fitter' representation is found, this becomes the current representation in the search space. This process iterates. Modules from this partition are then re-arranged systematically in order to find an improved partition (with better fitness function). If no

'fitter' representation is found, the search converges and the maximum is found. The pseudo-code of the clustering algorithm is shown in Algorithm 1.

```
Algorithm 1. MUNCH(ITER,M)
Input: ITER- the number of iterations (runs), M - An MDG
 1) Let C be a random (or specified - for seeded) clustering
    arrangement
 2) Let F = Fitness Function (See Section 2.2)
 3) For i = 1 to ITER (number of iterations)
 4)    Choose two random clusters X and Y (X≠Y)
 5)    Move a random variable from cluster X to Y
 6)    Let F'= Fitness Function
 7)    If F' is worse than F Then
 8)       Undo move
 9)    Else
10)       Let F = F'
11)    End If
12) End For
Output: C - a modularisation of M
```

2.2 Fitness Functions

A fitness function is used to measure the relative quality of the decomposed structure of system into subsystems (clusters). In our previous paper [Arzok11], we experimented with two fitness functions: the Modularisation Quality (MQ) metric of Mancoridis et al [Manco98], as implemented in Bunch; and the EValuation Metric (EVM) of Tucker et al [Tucke01], previously applied to problems of time-series data and the clustering of gene expression data. We also introduced EValuation Metric Difference (EVMD), a faster version of the EVM function. EVMD was selected as the fitness function for the modularisations as it is more robust than MQ and faster than EVM [Arzok11].

EVM rewards maximising the cohesiveness of the clusters (presence of intra-module relationships) clustering with a high number of intra-module relationships, but it does not directly penalise inter-clustering coupling. It looks at all possible relationships within a cluster and rewards those that exist within the MDG and penalises those that does not exist within the MDG. In other words, it looks at all possible links and for each cohesion relationship that exist the score is incremented by one, and for each cohesion relationship that does not exist, the score is decremented by one. However, this implies that it may indirectly penalise high coupling; re-arranging modules between clusters can change high coupling between two modules to lower coupling between them, and higher cohesion within one (or possibly both) of them [Harma05].

The objective of these heuristic searches is to maximise the fitness function. As the value of EVM is not normalised, there are no upper limits to the value of the functions. EVM has a global optimum that corresponds to all modules in a single cluster, where modules are all related to each other. The theoretical maximum possible value for EVM is the total number of links (relationships) in the graph, whereas the minimum value is simply the negative of the total number of links.

For the following formal definition of EVM, a clustering arrangement C of n items is defined as a set of sets $\{c_1, \ldots, c_m\}$, where each set (cluster) $c_i \subseteq \{1,\ldots,n\}$ such that $c_i \neq \phi$

and $c_i \cap c_j = \phi$ for all $i \neq j$. Note that $1 \leq m \leq n$ and $n > 0$. Note also that $\bigcup_{i=1}^{m} c_i = \{1,...,n\}$.

Let MDG M be an n by n matrix, where a 1 at row i and column j (M_{ij}) indicates a relationship between variable i and j, and 0 indicates that there is no relationship. Let c_{ij} refer to the j^{th} element of the i^{th} cluster of C. The score for cluster c_i is defined in equation 2.

$$EVM(C,M) = \sum_{i=1}^{m} h(c_i, M) \tag{1}$$

$$h(c_i, M) = \begin{cases} \sum_{a=1}^{|c_i|-1} \sum_{b=a+1}^{|c_i|} L(c_{ia}, c_{ib}) & ,\text{if } |c_i| > 1 \\ 0 & ,\text{Otherwise} \end{cases} \tag{2}$$

$$L(v_1, v_2, M) = \begin{cases} 0 & ,v_1 = v_2 \\ +1 & ,M_{v_1 v_2} + M_{v_2 v_1} > 0 \\ -1 & ,\text{Otherwise} \end{cases} \tag{3}$$

In order to make the process of modularisation faster, another fitness function, EVMD, was defined. It utilises an update formula on the assumption that one small change is being made between clusters. It is a faster way of evaluating EVM, where the previous fitness is known and the current fitness is calculated, without having to do the move. It calculates the value of what the fitness function is going to be. It produces the same results as EVM, but effectively reduces the computational operations from $O(n^{1.5})$ to $O(n^{0.5})$, thus reducing the speed significantly.

For the formal definition of EVMD, let f_{old} be the EVM fitness function. Also, let x be the *from* cluster, y be the *to* cluster and z be the *index*. Function G, defined in equation 5, determines the relationship (from MDG M) that exists between variable v and cluster k. Equation 3 simply checks whether it is a positive or negative influence (i.e. does a relationship exist?).

$$EVMD(f_{old}, C, x, y, z, M) = f_{old} - G(C_x, C_{xz}, M) + G(C_y, C_{xz}, M) \tag{4}$$

$$G(C_k, v, M) = \sum_{i=1}^{|C_k|} L(c_{ki}, v, M) \tag{5}$$

From this point forward EVM will be used when referring to the EVMD metric.

2.3 HS metric

Homogeneity and Separation (HS) is an external coupling metric defined to measure the quality of the modularisation. HS is based on the Coupling Between Objects (CBO) metric, first introduced by Chidamber and Kemerer [Chida94]. CBO (for a class) is defined as the count of the number of other classes to which it is coupled. It is based on the concept that if one object acts on another, then there is coupling between the two objects. Since the

properties between objects of the same class are the same, the two classes are coupled when methods of one class use the methods defined by the other [Chida94].

For the formal mathematical definition of the HS metric, we define a function $P(v,C)$ which returns the cluster number within C that variable (class) v resides.

$$HS(C,M) = \frac{H(C,M) - S(C,M)}{H(C,M) + S(C,M)} \tag{6}$$

$$H(C,M) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1 - \delta(M_{ij},0))\delta(P(i,M),P(j,M)) \tag{7}$$

$$S(C,M) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1 - \delta(M_{ij},0))(1 - \delta(P(i,M),P(j,M)) \tag{8}$$

We use Kronecher's Delta function $\delta(i,j)$, which is defined as follows:

$$\delta(i,j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{9}$$

HS is a simple and intuitive coupling metric that calculates the ratio of the proportion of internal and external edges. As shown in equation 6, HS is calculated by subtracting the number of links within clusters from the number of links that are between clusters, and then dividing the answer by the total number of links (to normalise it). In other words, it looks at all the links within the MDG, finding all the pairs that are not equal to 0. If the two variables are in the same cluster, H is incremented, and if they are in different clusters, S is incremented. The more links between the clusters the worse the modularisation, as only internal links are modularised (and not external ones). A value of +1 is returned if all the links are within the modules, a value of −1 is returned if all links are external coupling, and 0 is produced if there is an equal number.

2.4 Weighted-Kappa

Weighted-Kappa (WK) [Altma97] is an agreement metric defined for the comparison of two clustering arrangements. It not only measures similarity but also takes into account the degree of disagreements. WK is used to rate the agreements of the clustering arrangements of the time-series modularisation.

For the two clustering arrangements, rows represent one observer, whereas columns represent the other. Order is not of importance. WK is computed in terms of a matrix of observations; in this case a two-by-two contingency table is constructed. There is a maximum of four outcomes to a single paired observation, two observers and two observations (same cluster, different cluster). For two of the four outcomes, the observers agree with either on same cluster or on different cluster. For the other two outcomes, the observers do not agree. One observes that nodes are in the same cluster, and the other observes that they are in different clusters. For each of the four matrix elements the total number of occasions on which one of the four possible outcomes occurs is calculated.

On the leading diagonal, the two agreement outcome totals are recorded: same cluster and different cluster. The two possible disagreement outcomes are recorded in the other two elements of the matrix. If the value of the matrix is zero in all but the leading diagonal,

observers agree completely, which means that the clustering arrangements are identical. If the leading diagonal consists of only zero elements, then the clustering arrangements are in complete disagreement about all pair of nodes. If some non-leading diagonal elements are non-zero, then the clustering arrangements are not identical.

The WK value ranges from −1.0 (for total dissimilarity of clusters) and 1.0 (for identical clusters). A high WK value suggests that the two arrangements are similar, whereas a low value suggests that they are dissimilar. A value of approximately 0 is normally observed for two random clusters. An interpretation table of the WK values is shown in Table 1.

Table 1. Agreement strength of Weighted-Kappa.

Weighted-Kappa (WK)	Agreement Strength
$-1 \le WK \le 0$	Very Poor
$0 < WK \le 0.2$	Poor
$0.2 < WK \le 0.4$	Fair
$0.4 < WK \le 0.6$	Moderate
$0.6 < WK \le 0.8$	Good
$0.8 < WK \le 1.0$	Very Good

3 Experimental Design

This section describes the creation and pre-processing of the source data for this paper. It also describes a simple metric for calculating the similarity between subsequent graphs and explains the modularisation experiments conducted for this paper.

3.1 Data Creation

The large dataset used for this paper consists of information about different versions of a software system over time. It was provided by the international company Quantel Limited. Quantel is one of the world's leading developers of high performance content creation and delivery system across television and film post production. It supplies products to many of leading media companies, such as Fox, Sky, BBC and ESPN. Furthermore, they have been a recipient of many prestigious awards such as Oscars, Emmys and the MacRobert Award, presented by the Royal Academy of Engineering.

The data source for this paper is from processed source code of an award winning product line architecture library that has delivered over 15 distinct products. The entire code base currently runs to over 12 million lines of C++. It has been developed for over ten years and has taken over several person centuries of developer effort. The subset we analyse in this paper is the persistence engine used by all products, comprising of over 0.5 million lines of C++ [Cain09].

The Debug Symbol Information Program Databases (PDB files) are data files that contain all the type information in a system; they are produced by Microsoft Visual C++. Debuggers can interpret global, stack and heap locations and map them back to the types they represent. This file format is undocumented by Microsoft [Pietr02]. However in March 2002 an API released by Microsoft allowed access to (some of) the debug type information without undue reverse engineering [Schre01]. The PDB files for each version of the code were archived, and were analysed using bespoke software that interfaced with the PDB files

using the DIA SDK. Explanations on extracting type information using DIA SDK are in [Cain04].

The PDBs were checked into a revision control system. Data was collected over the period 17/10/2000 to 03/02/2005, with 503 PDBs in total. To ensure anonymity, all class names (types) in all the PDBs were sorted into an alphabetically sorted master class table. This was used as a global index to convert each class name to a globally unique ID. A total of 6120 classes exist in the system, however, not all classes exist at the same time slice; there are between 29 and 1626 active classes at any one time. Active classes are the classes that exist at a particular point in time. Hence, classes generally appear at certain time point, and then "disappear" at a later time point. Some of the appearances and disappearances of these classes are because when a class is renamed, it will appear in the dataset as a new class with a new identifier. At this time, we have no way to detect this phenomenon, but we look to resolve this as part of future research.

The dataset consists of five time-series of directed graphs with integer edge weights; the absence of an edge weight implies a weight of zero. We are currently performing the experiments using un-weighted (binary) graphs. The whole process of seeding and experiments will be the same for weighted and un-weighted. Only the fitness function will need to change for weighted graphs.

Each graph originally consisted of a 6120 by 6120 relationship matrix. It is highly sparse, as there are only between 29 and 1626 classes at any one point. An initial analysis showed that none of the graphs over the five types of relationships were fully connected; each graph consisted of numerous disconnected sub-graphs. This may seem unusual, as for it to be part of the same application each class should be indirectly related to all other classes. However, this is true if each type of graph is combined for each time slice, but not when each type of relationship is considered on its own. Table 2 describes how each graph represents a relationship between classes. For this paper, graphs of the five types of relationship were merged together to form the 'whole system' for particular time slices.

Table 2. Class relation types.

Class relationship	Description
Attributes	Data members in a class
Bases	Immediate base classes
Inners	Any type declared inside the scope of a class
Parameters	Parameters to member functions of a class
Returns	Return from member functions of a class

In [Arzok11] we modularised the full graphs in the dataset, knowing that around 55% of the dataset code is not Quantel's data, and thus should not be clustered. We have the classification table of the 6120 classes and what they represent. Classes not produced by Quantel consist of Standard Template Library (STL), Windows COM Interface class and component from a 3rd party library. The STL is a generic C++ library that consists of container classes, algorithms and iterators; it is used to implement standard data structures such as queues, lists and stacks. It is difficult to modularise source code that uses library functions due to the amount of coupling involved, the Quantel code uses a large number of *Strings* and *Vectors*. This produced lower HS and WK values than anticipated. Thus, for this paper we removed all the modules that are not produced by Quantel. Across the entirety of the lifespan of the software system there were 2801 classes produced by Quantel. However, they were not all active at any one time (as mentioned before), Figure 1 only displays active classes that are produced by Quantel (2801 classes) at each software check-in. Due to the removal of classes that were not produced by Quantel, the number of

classes at most graphs changed. As shown from Figure 1, there are now between 202 and 1193 active classes at any one point.

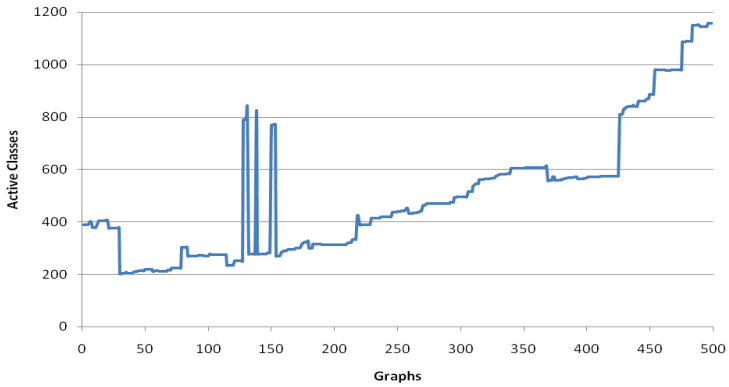


Figure 1 — [Quantel's active classes at each software check-in]

At points 128 and 132, the dataset seems to gain and lose a large number of classes. This behaviour repeats 2 more times in the dataset at points 138 and 139, and 150 and 154, as can be seen from Figure 1. We confirmed these results with Quantel and they have informed us that these major changes coincide with a new major release of a COTS (i.e. Commercial-Off-The-Shelf) library. They had a major update to a core piece of their software. The three spikes are when they have carried out each new release of it. We were informed by the manufacturer that this extensive class library had new functionalities including new classes.

3.2 Absolute Value Difference (AVD)

From the experimentation conducted in [Arzok11] we predicted and showed that within two weeks of development there were no significant changes to the source code that made two successive graphs very different (for seeding not to be possible). We also expected that if one graph was similar to the next, then modularisation would also be similar. To empirically find out whether this relationship existed we produced the matrix of each graph, and by subtracting the matrices of two successive graphs from each other and taking the absolute value of the results we produced a set of results showing the similarity between the graphs. Equation 10 shows how the absolute value difference was calculated for each graph, where X and Y are two n by n binary matrices (MDGs). A value of 0 indicates that two matrices are identical, whereas a large positive value indicates that they are different. A value between 0 and a large number gives a degree of similarity.

$$AVD(X,Y) = \sum_{i=1}^{n}\sum_{j=1}^{n}\left|X_{ij} - Y_{ij}\right| \tag{10}$$

Figure 2 shows the results of the AVD for the full dataset of 503 graphs. The results produced show that the majority of the graphs have very low AVD, as there were only a few days of development between each check-in. In fact, 46 per cent of the graphs have an AVD of 0. Sudden peaks and drops can also be seen in values, which could possibly indicate where major changes or refactoring work occurred.

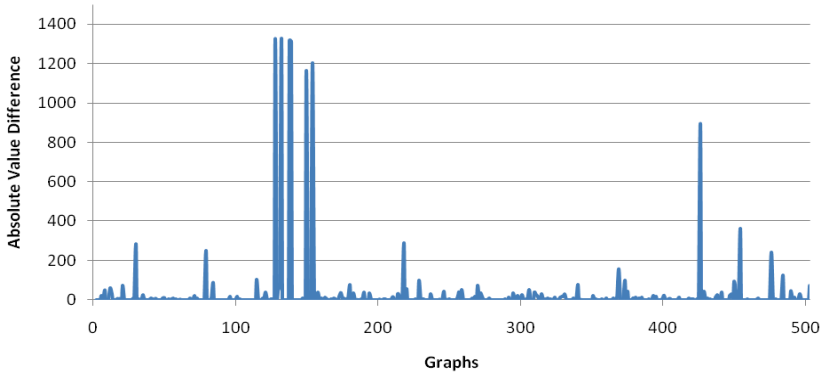


Figure 2 — [Plot showing the AVDs of the full dataset]

3.3 Experimental procedure

The main aim of the experiments is to modularise a dataset using a number of techniques which are explained below.

As described in Section 3.1, the full dataset consist of 503 sets of graphs with each graph containing 5 types of relationships combined together to form the 'whole' system at a particular time slice. There are roughly 2-3 days' gaps between each check-in, giving a total time span of 4 years and 4 months for the full dataset.

Five sets of experiments were designed for this paper. The main difference between the experiments is the number of iterations they run for and their starting clustering arrangements; otherwise it is the same program. Figure 3 shows a representation of the relationships between the five experiments.

The five experiments described below were conducted only once for the full dataset of 503 graphs. Then, the same five experiments were repeated ten times but only for the first 50 graphs of the same dataset.

Experiment 1 (C) - We modularised the dataset for 8 million iterations each. The starting clustering arrangement consisted of every variable in its own cluster. It assumes that all classes are independent; there are no relationships.

Experiment 2 (S) - We modularised the dataset using results of the previous clustering arrangement from C. Instead of creating a random starting arrangement for the modularisation, the clustering arrangement of the preceding graph (produced from C) was used to give it a head start. For example, for modularising the fourth graph, the results from the full modularisation of the third graph were used. Graphs were modularised for 80,000 iterations apart from the first graph, which was run for the full 8 million iterations.

Experiment 3 (SS) - We modularised the dataset using the preceding results of the modularisation. Instead of creating a random starting arrangement when the modularisation process starts, the clustering arrangement of the preceding seeded graph was used. For example, for modularising the fourth graph, results produced from the third seeded graph were used as the starting arrangement. The first graph was run for 8 million iterations, as it has no preceding graph. All other graphs were modularised for 80,000 iterations only.

Experiment 4 (SD) - The dataset was also modularised using the results produced from the modularisation of the preceding graph. However, unlike the other experiments, the number of iterations was not fixed. It varied depending on the similarity of the graphs. The AVD was calculated (as described in Section 3.2) for all the graphs and was used as a scalar for controlling the number of runs. The more similar the two graphs (low AVD), the

less runs needed. The more different two successive graphs (high AVD) are, the higher the number of iterations. Identical graphs with 0 AVD run for 0 number of iterations. Equation 11 was used for calculating the number of iterations of each graph.

$$ITER = AVD \times 8000 \tag{11}$$

Experiment 5 (SSD) - We modularised the dataset using the preceding results of the modularisation as in SS, while using equation 11 to calculate the number of iterations as in SD.

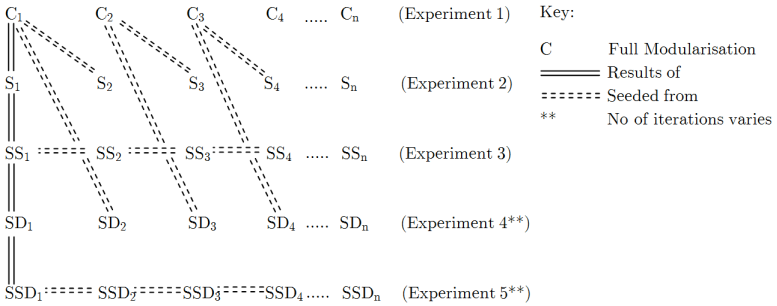


Figure 3 — [The relationships between the experiments]

4 Results and Discussion

This section presents and discusses the results of the experiments for one run of the full dataset and ten repeats of a sample of the dataset.

Without loss of generality, the amount of time it takes the modularisation program to run is proportional to the number of fitness function calls. In our experiments the number of fitness function calls is referred to as the number of iterations, and the time it takes the program to run is proportional to the number of iterations.

4.1 Full dataset experiments

Figure 4 shows a plot of the EVM values produced for the five experiments. Where the results overlap on the plot, the same EVM values are produced despite the fact that S and SS were run for 1% of the original time of C. This proves that the seeding technique works to a fair degree of accuracy. For S and SS a clear increasing trend of EVM is observed from the plot, which is due to graphs increasing in size. It seems to correlate with the plot in Figure 1, which shows an increase in the number of active classes throughout the project, apart from few peaks and drops, which may possibly suggest where radical extensions or refactoring events have taken place.

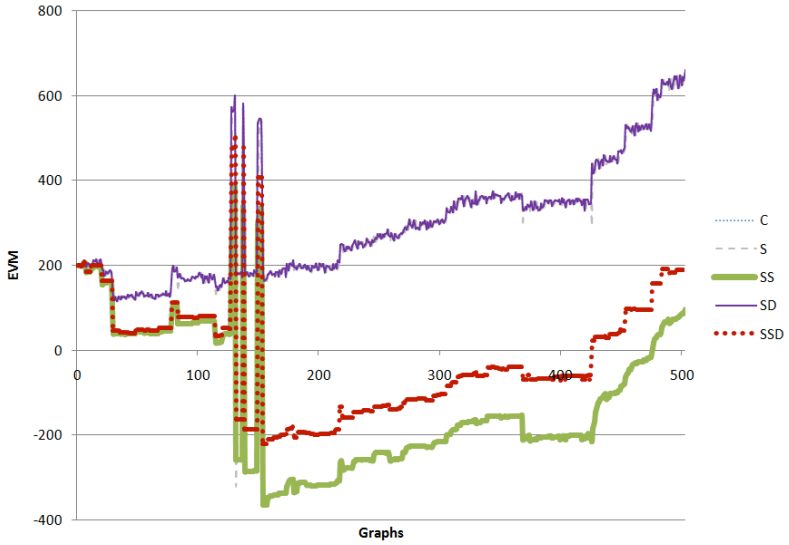


Figure 4 — [EVM results of the full dataset for the five experiments.]

The peaks from the results correlate with the spikes from the AVD graph, shown in Figure 2. The results produced allow us to find out how different two modularisations are without actually running the modularisation.

It is also interesting to see from the plot how the seeding strategy breaks down when there are large changes. The EVM values of few graphs from S are negative, due to the major differences among these graphs (major changes being made to the code). The starting clustering arrangements of these particular graphs were very poor, and they needed a longer running time for the modularisation. However, for SS it can be seen that after the major changes were made to the code, EVM values dropped to very low negative values (showing how the structure crumbled).

Note that it is not possible to differentiate C from the plot, as it overlaps with SD. SD produces the same results as C because the graphs are the same and the full modularisation results from C are used as the starting clustering arrangements. The average percentages of the fitness function calls for C and SD are 8 million and 232,095, respectively. Thus, in terms of running time, SD was more than 34 times faster than C, despite the fact that they both produced identical results. However, in the real world, we would not have the full modularisation results. Thus, SSD was conducted combining SS and SD together to produce a run that runs as SS, but the iterations are computed as in SD. From Figure 5 it can be seen that SSD produces better EVM values than SS. This illustrates how introducing the scalar to control the number of iterations produces better results. However, it still produces values that are considerably lower the EVM values of C. We look to investigate this further as part of future work.

Figure 5 shows a plot of the HS values for the five experiments. From the plot a gradually decreasing trend of HS values can be observed. It seems that HS values are gradually getting worse throughout the life of the project. The HS results of C, S and SS seem to be very similar, overlapping for most of the time, even though S and SS were run for only 1% of the original time for C. Thus, results from experiment S have 100 fold improvements in time with less apparent loss in performance. We are not stating that we can perform better than C, but instead we are saying that if we could reproduce the same results as C, but in a shorter period of time through seeding, then we are saving 100 times the amount of run time for the majority of the results.

Note that it is not possible to differentiate C from the plot, as it overlaps with SD. SD produced HS results that are identical to C, despite the fact that it was considerably faster to run, whereas SSD, which also was more than 34 times faster than C, produced HS results better than C. This suggests that the seeding strategy works very well.

For produced modularisations, the negative HS values indicate that the inter-module edges are more than the intra-module edges. As predicted in [Arzok11], excluding all the classes not produced by Quantel from the modularisation significantly improved the HS results of the modularisation across the experiments. Nevertheless, the HS results produced are still low; we believe that this is because the system contains more coupling between the modules than within the modules.

In addition, it seems that large changes or refactoring events occurred a few times throughout the life of the project. The plot illustrates that there is a reduction of coupling to a certain degree during these events.

There is a noticeable trend between the HS of C observed from Figure 5 and the number of active classes from Figure 1. To find out whether there is a relationship between the two, we correlated them. A value of -0.841 is produced, which indicates a very high negative correlation. This shows that as the number of classes increase, the HS measure of how good coupling is decreases.

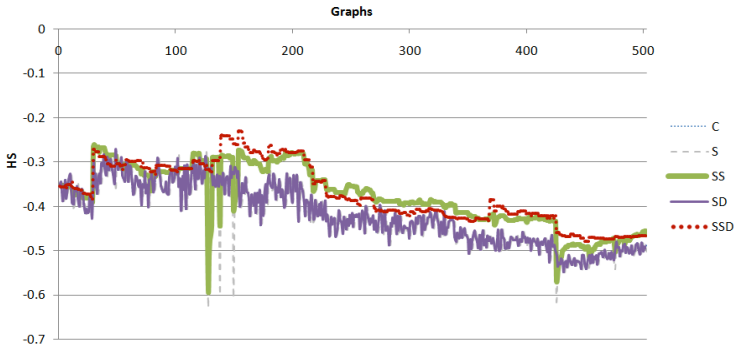


Figure 5 — [HS results of the full dataset for the five experiments]

Figure 6 shows a plot of the WK values for the clustering results of the 1st graph compared to the i^{th} clustering results for C. From the plot a decreasing trend of WK values can be observed. The WK values are initially between 0.3 and 0.6 which are considered to have moderate or fair agreement strengths, according to Table 1. These WK values become poorer over the lifespan of the project. This illustrates the deterioration of the original structure of the system over time. From the plot, it can be seen that the WK values vary widely; we believe that this is due taking the results of a single run of the Hill Climbing algorithm.

Figure 7 displays a plot showing the WK results of the modularisations produced by C and SS. WK of individual successive graphs seems to vary. However, a gradual drop of WK values can be observed from the graph. It also shows a compounded error gradually building up throughout the seeding strategy. However, there seem to be some structures in the results to be seeded through; presumably the same core structure is maintained all the way through the result while the rest degrades.

Figure 8 displays the WK values of C and SSD. A drop of WK values can be seen from the plot. A compounded error gradually building up throughout the seeding strategy can also be observed from the plot. Like Figure 7, there still seems to be some structure in the results being seeded through. However, we expected the WK values to be higher than this, and we look to investigate this further.

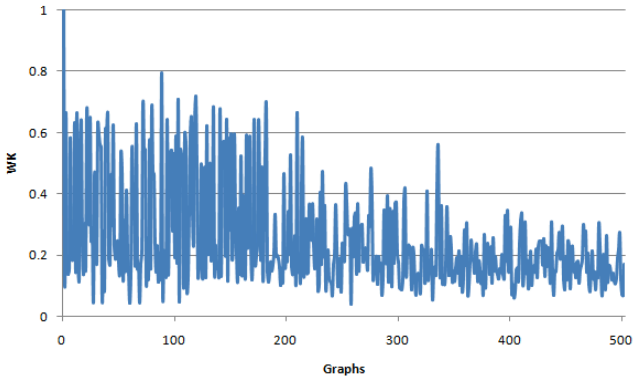


Figure 6 — [WK results between C_1 and and C_i for the full dataset]

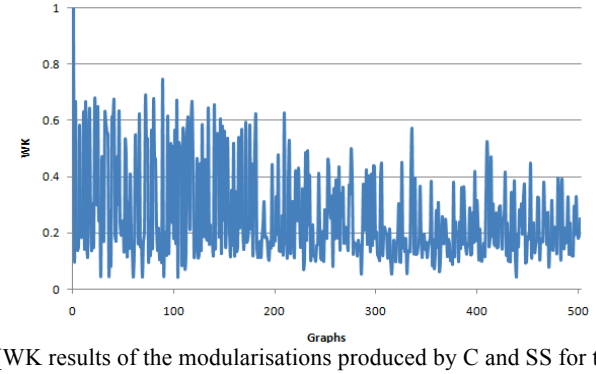


Figure 7 — [WK results of the modularisations produced by C and SS for the full dataset]

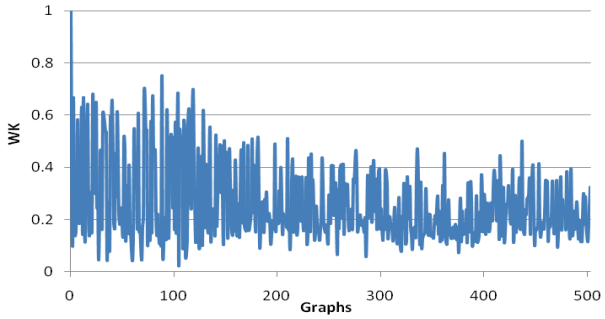


Figure 8 — [WK results of the modularisations produced by C and SSD for the full dataset]

4.2 50 graphs experiments

One of the well known problems with the Hill Climbing algorithm is that it can run into and get stuck in Local Maximums. In order to show whether this is happening or not, a practitioner often runs a number of repeat experiments. However, the main problem with this type of data is the running time of the experiments. Although we have recently upgraded to an enterprise level server, Experiment C still requires 5 days to run. To do ten repeats, 50 days of running time for experiment C alone is needed. Nonetheless, to work out the consistency and variability of the Hill Climbing, the modularisation of the first 50 graphs were repeated ten times.

Figures 9, 10, 11, 12 and 13 show the plots of the average, minimum, maximum and standard deviation of the EVM values for each of the five experiments. These EVM values were collected from 10 repeats of the modularisations. The average, minimum and maximum of the five experiments seem to be similar, which shows that there is some consistency in the results. Also, the standard deviation results from the plots seem to be fairly low throughout the five experiments, which indicate that the results produced from each run is close to the mean and thus represents consistency. In addition, the EVM values from the plot seem to be very similar to the EVM values of the first 50 graphs of Figure 4.

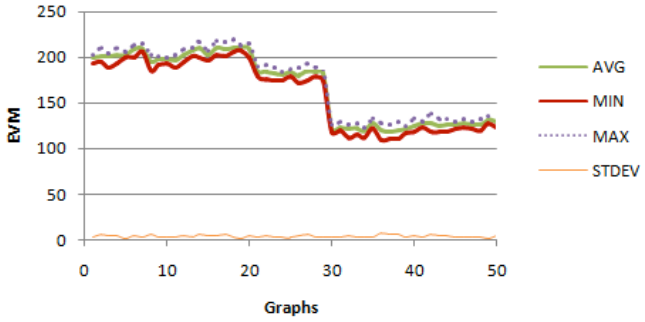


Figure 9 — [Avg, min, max and SD of EVM valuess for 10 repeats of C]

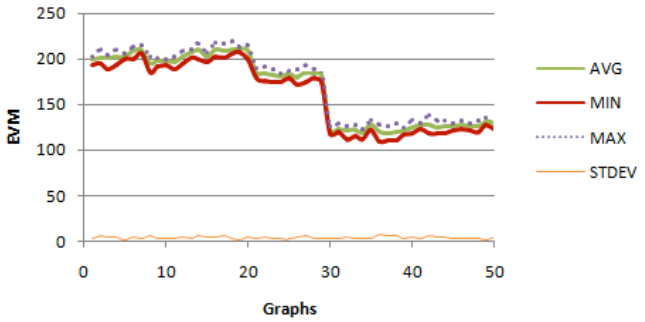


Figure 10 — [Avg, min, max and SD of EVM valuess for 10 repeats of S]

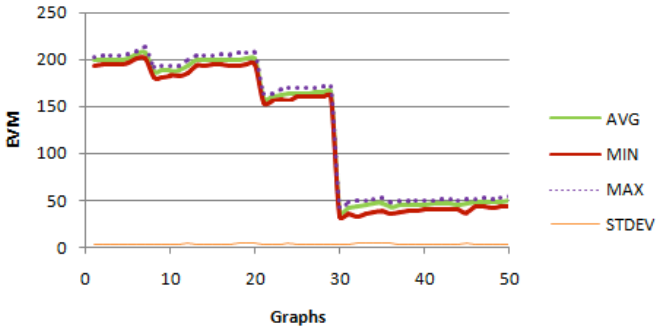


Figure 11 — [Avg, min, max and SD of EVM valuess for 10 repeats of SS]

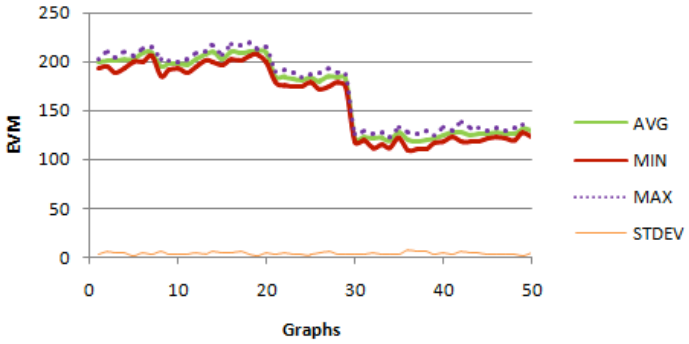


Figure 12 — [Avg, min, max and SD of EVM valuess for 10 repeats of SD]

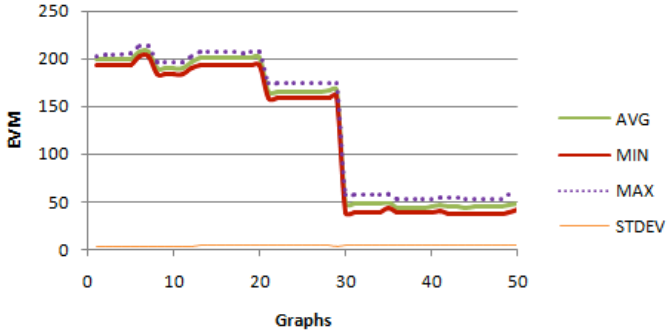


Figure 13 — [Avg, min, max and SD of EVM valuess for 10 repeats of SSD]

Figure 14 shows a plot of the average HS values of the first 50 graphs for 10 repeats of the five experiments. Note that C cannot be seen from the plot as it overlaps with SD, and S is not very noticeable as it overlaps with SS. From the plot a very gradually decreasing trend of HS values can be initially noticed. A sudden increase in HS values is then observed. We believe that this increase in HS values is due to the removal of nearly 200 classes from the system at that software check-in (Figure 1).

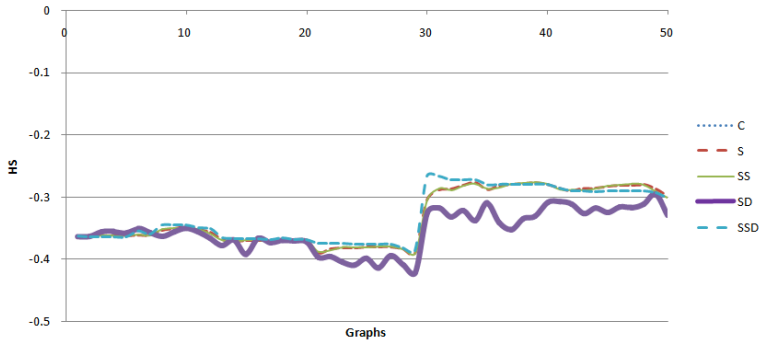


Figure 14 — [Average HS results for 10 repeats of the five experiments]

Figure 15 shows the average WK results between the first graph and the i^{th} graph for 100 comparisons, as there are 10 C_1 and 10 C_i. A clear decreasing trend can be observed from the graph illustrating the gradual decay of the system over time. The WK values went down from 0.6 to 0.4 in the span of 3 months, and there still seems to be structure and similarity between the graphs. This justifies the seeding strategy implemented for the paper.

Figure 16 shows the average WK results between C and SS for the 10 repeats. The first repeat of C was used for seeding the first repeat of SS. Since choosing the first repeat to

seed from is as valid as choosing a random repeat, WK was only calculated for the 10 repeats between C and SS. A gradual smooth drop of WK values can be observed from the plot and therefore there seems to be some structures in the results that are being seeded through. The WK values do not vary widely as in Figure 7, due to the number of repeats conducted.

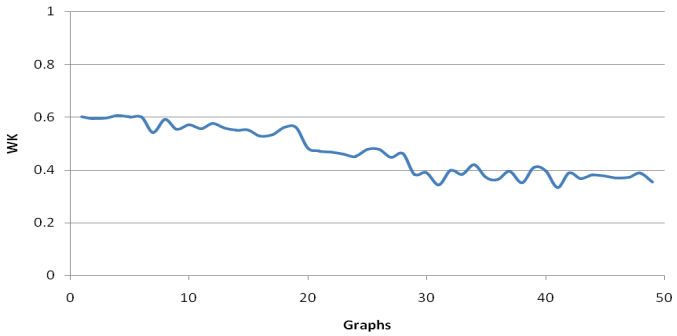


Figure 15 — [Average WK results between C_1 and and C_i]

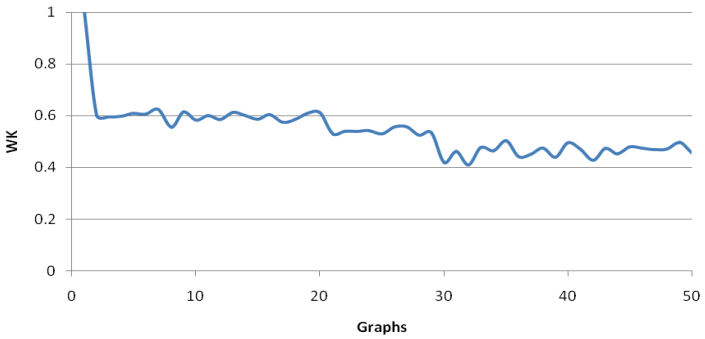


Figure 16 — [Average WK results of the modularisations produced by C and SS]

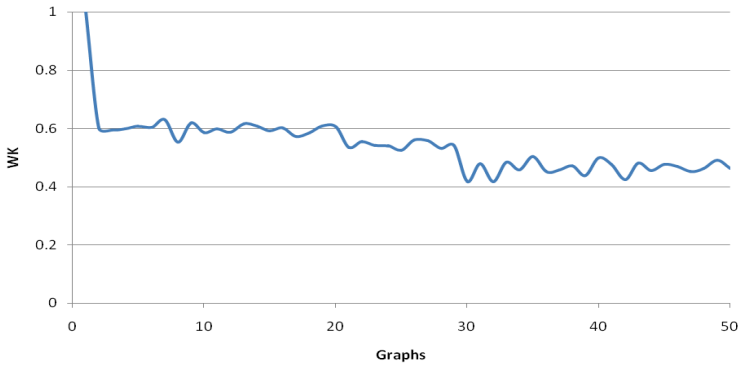


Figure 17 — [Average WK results of the modularisations produced by C and SSD]

Figure 17 displays the average WK results between C and SSD for the 10 repeats. As in Figure 16, there is a gradual drop of WK values and a compounded error building up throughout the seeding strategy. The WK values of C and SSD were generally the same as the WK values of C and SS, however, on a number of graphs WK values are higher. This illustrates that using the scalar to control the number of iterations (depends on the similarity between graphs) was a much more robust way of conducting the experiment, as it not only reduces the overall running time of the experiment but also provides enough iterations to

reach the optima. Also, WK values do not vary widely as in Figure 8, due to the number of repeats conducted.

Figure 18 displays the average WK results of every pair-wise comparison of the 10 repeats for each of the five experiments. An average WK value of 0.6 between the graphs shows the clustering arrangements of the runs to be reasonably consistent to each other. However, there still needs to be repeats as Hill Climb is a stochastic method and can produce varying results.

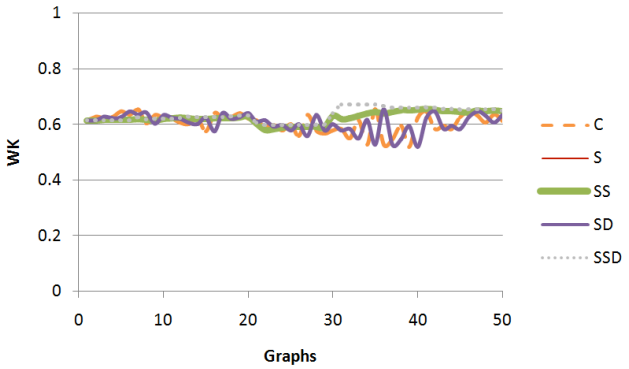


Figure 18 — [Average WK results of all pair-wise comparision of the 10 repeats]

5 Threats to Validity

The problem with Hill Climbing is that there is a risk of the search reaching only the local maxima and thus a large number of runs are needed. However, due to the amount of running time involved when conducting the experiments, only one run was conducted on the full dataset. However, a smaller sample of only 50 graphs was repeated ten times.

We are aware that Hill Climbing is not necessarily the most effective search technique, however since the main aim of this paper was to explore the use of seeding techniques to reduce runtime, it was suitable for our purposes and quick to implement. Now that this initial "proof of concept" part of our project has been successful, we aim to explore the use of other Heuristic Search techniques such as Genetic Algorithms and Simulated Annealing.

Table 2 currently contains data relating to returns, parameters, attributes, inners and bases. These were relatively easy to extract using the DIA SDK kit, however, obtaining the method information i.e. a method using another class as a local variable was more difficult. This type of information is at a much deeper level in the data structure and is significantly more difficult to obtain. Also, the data extraction process was implemented at Quantel, and they provided us with these information. We would also like to acknowledge that we will only be able to get from the source code the structural relationship. Conceptual design might not be fully appreciated, as MDG is only an approximation of some of the structure.

We are aware that our fitness function might not be a good indicator for the quality of the modularisation and as a result we have used an external metric of validity, HS, which is based on CBO metric [Chida94]. It is essentially a count of the coupling links; the difference between intra and inter coupling.

A drawback of this type of work is that we have only used empirical ways of evaluating the software metric. Thus, we would also like to acknowledge that there is an absence of qualitative evaluation of the clusters in the paper and we aim to obtain these as part of future work. We aim to discuss the meaningfulness of the clusters with our industrial collaborator on a case by case basis and report these results.

6 Conclusions and Future Work

For this paper, we did not treat our dataset as 503 separate modularisation problems, but instead we took advantage of the fact that out dataset is time-series. We used results of previous time slices to speed up the search process of the next time slice. We reduced the duration of the process by a factor of 100, and modularised what would take 100 hours in 1 hour, to a good degree of accuracy.

The results produced allow us to find out how different two modularisations are without actually running the modularisation. Due to the large correlation between subsequent graphs, we can use quick statistics of the AVDs to determine the similarity. This reduces the computational complexity from hours to seconds. However, this statistic will not inform us where the modules should be and what is related together.

For this paper we used the similarity of the graphs to control the number of iterations we ran the seeded modularisation for. The AVD of the graphs was analysed and used as a scalar to determine the run time. This technique caters for the fact that when there are major extensions or refactoring events, (almost) full modularisation is needed. Using this seeding strategy we managed to produce results identical to the full modularisation of graphs (when the full modularisation results are available to seed from) while reducing the running time by more than 34 times. Without having the full modularisation results we still managed to produce HS values using the seeding strategy that are higher than the full modularisation of graphs. However, the EVM values were lower than expected, and we look to investigate this as part of future work.

Currently, the AVD values of graphs is used for specifying the number of iterations of the modularisation, however these values are not normalised. Thus, as part of future work, we look to convert them into probability values that will inform us of the significance of the values that we seed from subsequent changes.

If successive modularisations are very different, this may suggest that the program has been radically refactored or radically extended. Due to the nature of our data, we cannot check whether these changes are refactoring or just an extension *pro tem*; we can only determine whether one class uses another.

We suspect that refactoring is occurring and not simply other development because we were informed by Quantel that they massively refactor and that this is a practice they encourage all staff to strive towards. Their senior systems architects are proactive in promoting and pushing refactoring techniques. Thus, by finding areas of major change it will either be new functionality or refactoring. The aim of our project is to be able to identify one or the other, but, we currently do not have the data to distinguish between the two. At the moment being able to identify areas of interest is useful, as it allows us to indicate the potential locations of refactoring in the code. Although we cannot prove that refactoring activities are occurring, we know the locations of where refactoring does not occur, on the basis that there are very few changes.

We aim to investigate this further by correlating the results produced with information from the developers, perhaps even their commit comments in the version control system. We might need to find the number of non-comment lines of code for each class. If there have been major changes in the dependency graph and the number of code lines remained roughly the same, then there had not been major functionality added. We also aim to look at source code analysis software that would try to detect Fowler's 72 refactorings [Fowle99], in order to detect whether something has occurred.

We currently have a granularity of 10 classifications for each of the classes in the dataset. Quantel has indicated the classes that they have developed, 4 out of the 10 classification are Quantel's code. Classes that they have not developed include STL and

components from a third-party library. Eliminating these classes significantly improved the results, producing higher HS and WK results. However, the classifications provided are too general, and one of our aims, as part of future work, is to strive towards a much more granular look at the data.

In addition, we aim to compare the techniques and approaches mentioned in this paper against more systems and perform a more systematic comparison.

This paper mainly concentrates on speeding up the process of modularisation, but also attempts to identify areas in the code that has been radically extended or refactored. It measures the degree of change and not the reason behind the change. However, as part of future work, we look to identify the major changes occurring and to classify them as refactoring or extension accordingly. We also look to apply Search Based Software Engineering and Intelligent Data Analysis techniques to our large dataset to try and predict changes and potential refactorings.

Acknowledgments

The authors would like thank Quantel Ltd for providing us with their unique dataset. We would also like to thank all of the reviewers for their useful comments and suggestions.

References

[Altma97] D. G. Altman: *Practical Statistics for Medical research*, Chapman and Hall, 1997.

[Arzok11] M. Arzoky, S. Swift, A. Tucker, J. Cain: "Munch: An Efficient Modularisation Strategy to Assess the Degree of Refactoring on Sequential Source Code Checkings", IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp.422-429, 2011. doi: 10.1109/ICSTW.2011.87.

[Cain04] J. Cain: "Debugging with the DIA SDK", Visual System Journal, http://www.developerfusion.com/article/84368/debugging-with-the-dia-sdk/.

[Cain09] J. Cain, S. Counsell, S. Swift, and A. Tucker: "An Application of Intelligent Data Analysis Techniques to a Large Software Engineering Dataset", Advances in Intelligent Data Analysis VIII: 8th International Symposium on Intelligent Data Analysis (IDA09), Lecture Notes in Computer Science, 2009. doi: 10.1007/978-3-642-03915-7_23.

[Chida94] S. R. Chidamber and C. F. Kemerer: "A metrics suite for object oriented design", IEEE Transactions Software Engineering, vol. 20, no. 6, pp. 476-493, 1994. doi: 10.1109/32.295895.

[Consta79] L. L. Constantine and E. Yourdon: Structured Design, Prentice Hall, 1979.

[Fowle99] M. Fowler, K. Beck, J. Brant, W. Opdyke and R. Don: Refactoring: Improving the Design of Existing Code, Object Technology Series, 1st edn, Massachusetts: Addison-Wesley, 1999.

[Harma02] M. Harman, R. Hierons, and M. Proctor: "A new representation and crossover operator for search based optimization of software modularization", Proc. Genetic and Evolutionary Computation Conference

(GECCO 02), Morgan Kaufmann Publishers, pp. 1351–1358, 2002. doi: 10.1.1.144.5252.

[Harma05] M. Harman, S. Swift, and K. Mahdavi: "An empirical study of the robustness of two module clustering fitness functions", Genetic and Evolutionary Computation Conference (GECCO 2005), pp. 1029–1036, 2005. doi: 10.1145/1068009.1068184.

[Manco98] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen and E. R. Gansner: "Using automatic clustering to produce high-level system organizations of source code", International Workshop on Program Comprehension (IWPC'98), IEEE Computer Society Press, pp. 45-53, 1998. doi: 10.1.1.14.3376.

[Manco99] S. Mancoridis, B. S. Mitchell, Y. F. Chen, and E. R. Gansner: "Bunch: A clustering tool for the recovery and maintenance of software system structures", Proc. IEEE International Conference on Software Maintenance, IEEE Computer Society Press, pp. 50–59, 1999. doi: 10.1109/ICSM.1999.792498.

[Manco02] S. Mancoridis, M. Traverso: "Using Heuristic Search Techniques to Extract Design Abstractions from Source Code", Proc. Genetic and Evolutionary Computation Conference (GECCO 02), Morgan Kaufmann Publishers, 2002. doi: 10.1.1.6.4923.

[Micha00] Z. Michalewicz and D. B. Fogel: How to Solve It: Modern Heuristics, Springer-Verlag, 2000.

[Mitch02] B. S. Mitchell: A Heuristic Search Approach to Solving the Software Clustering Problem, PhD Thesis, Drexel University, Philadelphia, 2002.

[Pietr02] M. Pietrek: "Under the Hood", MSDN Magazine, vol. 17, no. 3, 2002.

[Pradi11] K. Praditwong, M. Harman and X. Yao: "Software Module Clustering as a Multi–Objective Search Problem", IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 264–282, 2011. doi: 10.1109/TSE.2010.26.

[Somme95] I. Sommerville: Software Engineering, 5th edition, Addison-Wesley, 1995.

[Schre01] S. Schreiber: Undocumented Windows 2000 Secrets: A Programmer's Cookbook, Addison-Wesley, 2001.

[Strogg07] K. Stroggylos and D. Spinellis: "Refactoring does it improve software quality?", WoSQ 07: Proceedings of the 5th International Workshop on Software Quality, IEEE Computer Society, 2007. doi: 10.1109/WOSQ.2007.11.

[Sures10] L. Suresh, J. B. Simha and R. Velur: "Seeding cluster centers of K-means clustering through median projection", CISIS, pp.217-222, 2010. doi: 10.1109/CISIS.2010.133.

[Swift04] S. Swift, A. Tucker, V. Vinciotti, N. Martin, C. Orengo, X. Liu, et al: "Consensus clustering and functional interpretation of gene-expression data", Genome Biology, vol. 5, no. 11, 2004. doi: 10.1186/gb-2004-5-11-r94.

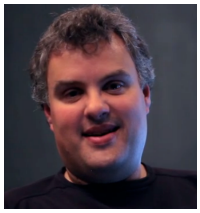
[Tucke01] A. Tucker, S. Swift, and X. Liu: "Variable Grouping in multivariate time series via correlation", IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, vol. 31, no. 2, pp. 235–245, 2001. doi: 10.1109/3477.915346.

About the author(s)

Mahir Arzoky is a PhD student at the School of Information Systems, Computing and Mathematics of Brunel University, UK. He received his BSc in Computer Science from the same institute in 2009. His research interest includes search based software engineering, data clustering and refactoring.

Stephen Swift received a BSc degree in Mathematics and Computing from the University of Kent, UK, an MSc in Artificial Intelligence from Cranfield University, UK and a PhD degree in Intelligent Data Analysis from University of London (Birkbeck College). He is a Lecturer within the Department of Computer Science, Brunel University, UK. He spent four years in industry as a web designer and programmer and has four years postdoctoral research experience. His research interests include heuristic search, data clustering and evolutionary computation. He has applied his research to various areas including software engineering, bioinformatics and healthcare.

Allan Tucker is a Senior Lecturer within the Department of Computer Science, Brunel University. He received his BSc in Cognitive Science from the University of Sheffield, UK and his PhD in Computer Science from the University of London (Birkbeck College). His research interests include machine learning and data-mining of software, biomedical and ecological data. He is on the board for the Artificial in Medicine Conference and is a member of the ICES working group on fish ecology and a member of the British Ecological Society specialist interest group on Predictive Ecology.



James Cain is Principal Software Architect at Quantel Limited. He has over fifteen years of experience as a professional developer. While with Quantel, he has been instrumental in the design of a product line architecture and framework that has generated over a dozen tools. These have subsequently been used on the production of award winning films and media and a news and sports server system used 24 hours a day at the BBC and Sky. His PhD in Computer Science from Reading University looked at the effects of scale on programmer productivity when developing large software systems.