Software Clustering based on Dynamic Dependencies

Chenchen Xiao and Vassilios Tzerpos
York University
Toronto, Ontario, Canada
{cxiao,bil}@cs.yorku.ca

Abstract

The reverse engineering literature contains many software clustering approaches that attempt to cluster large software systems based on the static dependencies between software artifacts. However, the usefulness of clustering based on dynamic dependencies has not been investigated. It is possible that dynamic clusterings can provide a fresh outlook on the structure of a large software system.

In this paper, we present an approach for the evaluation of dynamic clusterings. We apply this approach to a large open source software system, and present experimental results that suggest that dynamic clusterings have considerable merit.

1 Introduction

As software systems evolve, they become larger and more complicated. The task of reverse engineering such systems becomes increasingly difficult due to the sheer size of information available. For instance, version 1.3 of Mozilla contains 13,899 source files, and approximately 4.5 million lines of code. Since documentation is often out of date, if existant at all, the task of understanding the architecture of large software systems can be daunting, especially to a newcomer to the project.

Software clustering techniques can break a software system into smaller parts, such as subsystems or modules. This can provide a starting point towards the recovery of the system's structure. Typically, these approaches attempt to cluster a software system's resources based on the static dependencies between them [19, 10, 12, 22]. Some of these approaches view software clustering as an optimization problem [16, 15]. Others utilize techniques from information theory [7], concept analysis [14, 20] and pattern matching [9, 18].

Clustering based on static analysis has been shown to provide meaningful insight on the structure of large software systems. However, the possible benefits of using dynamic analysis facts as input for the clustering algorithms have not been investigated. We identify two main reasons why using dynamic analysis facts can be beneficial:

- Dynamic dependency graphs are commonly weighted. For example, the number of times a particular procedure call or data reference took place during run time can be its weight. Even though weights can be assigned to static dependencies as well, e.g. when a procedure includes multiple references to the same variable, the variance of dynamic weights is significantly larger. Such information can help clustering algorithms identify the important dependencies in the system's structure.
- A well-known cluster analysis fact is that "there is no single correct answer to a clustering problem". Depending on one's point of view, different clusterings might be appropriate. As a result, limiting ourselves to static clusterings seems unnecessary. It is quite likely that dynamic clusterings will also provide interesting insight into a system's structure.

In this paper, we attempt to evaluate the usefulness of providing dynamic dependencies as input to software clustering algorithms. For this purpose, we have extracted static and dynamic dependency graphs from a large software system, version 1.3 of the Mozilla browser. We have also constructed an authoritative decomposition for this system based on an authoritative decomposition that existed for a previous version [11], as well as the help of the Mozilla community. Our experimental results suggest that dynamic clusterings have considerable merit.

The structure of the rest of this paper is as follows. Section 2 presents some of the existing technology that we used for this work. Section 3 outlines our reasons for selecting Mozilla as the example software system, and describes the process of creating an authoritative decomposition for it. Our approach to evaluating clusterings based on dynamic dependencies is presented in Section 4. Experiments that demonstrate the usefulness of dynamic clusterings are presented in section 5. Finally, Section 6 concludes the paper.

2 Background

This section presents the algorithms and tools that we chose for our study. The example software system, Mozilla, is presented in Section 3, along with the process of creating its authoritative decomposition.

We start by introducing the software clustering algorithms that we experimented with. Our choice was only limited by availability of implementation. The algorithms we chose were:

- **ACDC**. This is a pattern-based software clustering algorithm that attempts to recover subsystems commonly found in manually-created decompositions of large software systems [22].
- **Bunch**. This is a suite of algorithms that attempt to find a decomposition that optimizes a quality measure based on high-cohesion, low-coupling. We experimented with two versions of a hill-climbing algorithm, we will refer to as NAHC and SAHC (for nearest- and steepest-ascend hill-climbing) [16].
- **Cluster Analysis Algorithms**. We also experimented with several hierarchical agglomerative cluster analysis algorithms. We used the Jaccard co-efficient that has been shown to work best in a software clustering context [8]. We experimented with four different algorithms: single linkage (SL), complete linkage (CL), weighted average linkage (WA), and unweighted average linkage (UA).

In order to compare the output of these algorithms to the authoritative decomposition of Mozilla, we used the MoJoFM effectiveness measure [25]. This is a measure that is based on MoJo distance [21, 24], but presents the effectiveness of the algorithm as a percentage. The basic premise is that the smaller the distance of a proposed decomposition to the authoritative one, the more effective the algorithm that produced it[1].

Other comparison techniques, such as the ones introduced by Koschke and Eisenbarth [13], and Mitchell and Mancoridis [17] could also have been chosen. Part of our future work will include determining whether the choice of comparison technique affects the obtained results in a significant fashion.

For the static analysis of Mozilla, we chose Swagkit [1], a suite of reverse engineering tools developed by the Software Architecture Group (SWAG) at the University of Waterloo. It has been used in a number of reverse engineering projects in order to extract, abstract and visualize software architectures. The main feature of Swagkit that was of interest to us is the inclusion of the CPPX fact extractor for C/C++ source code [2]. The features of CPPX that made it the right choice for our study were:

1. **Open source**. The extractor and its source code can be obtained freely.
2. **Quality**. CPPX uses GNU C++ as its front end. Since parsing and semantic analysis of C++ programs is done by *gcc*, CPPX accepts all dialects of C++ that *gcc* can accept.
3. **Performance**. CPPX runs only slightly slower than the *gcc* compiler.
4. **Output format**. Three different output formats are supported. They are TA, VCG and GXL. TA is a fact file format that contains tuples representing relationships between entities, as well as attributes representing various properties of these entities. VCG is a format that can be used for visualization purposes, while GXL is the well-known Graph Exchange Language [3]. Our experiments utilized the TA output.

Finally, selecting the right profiling tool was an important decision for the dynamic analysis part of our work. The Mozilla community provides several profiler tools [4], such as *gprof* and *jprof*.

gprof is a popular profiler tool for C/C++ software system. However, we found that its output did not contain enough information for the purposes of our experiments. It turned out that *gprof* cannot handle threads or shared libraries. Since Mozilla is a multi-threaded program, using *gprof* would lose important execution information.

The Mozilla community has developed another profiler tool especially for the analysis of the performance of Mozilla, called *jprof*. Since *jprof* does not suffer from the problems mentioned above, we selected it for our work. The information we extracted from the output of *jprof* was the number of times a particular function call between two source files occurred at run time.

In the next section, we present our example software system.

3 Mozilla and its Authoritative Partition

In order to evaluate the usefulness of dynamic dependency graphs for clustering purposes, we needed to find a software system with particular properties:

1. The size of the system must be considerable, so that an attempt to cluster it is meaningful.
2. The source code must be readily available.

[1]A Java implementation of MoJoFM is available for download at: http://www.cs.yorku.ca/~bil/downloads

File type	KLOC	# of files	# of files compiled under Linux
.h	1,504	7,911	1,871
.c	1,039	1,830	305
.cpp	1,976	4,158	1,383
Total	4,519	13,899	3,559

Table 1. Mozilla 1.3 source distribution

3. It should be easy to find parsing and profiling tools.
4. A comprehensive test suite that will ensure adequate coverage for the dynamic facts is required.

Many open-source systems have most of these properties. However, the last requirement disqualifies the majority of them, since they do not normally come with a test suite. Thankfully, Mozilla [5] fit all our requirements.

Mozilla is a huge open source software system with more than four million lines of code written in C/C++. As presented in Section 2, we had access to all necessary parsing and profiling tools. Moreover, a test suite, known as the Mozilla "smoketests" is available on the Mozilla website. It covers a large part of the system's functionality. Several "modules" of Mozilla also publish test suites. By combining all these together, we were able to create a test suite that covered a significant portion of the source code.

The latest version available when this work started was Mozilla 1.3. This was the version selected for this study. Various facts about the source distribution of this version are presented in Table 1.

Our next task involved the creation of an authoritative partition for Mozilla 1.3, so that the output of the various clustering algorithms can be evaluated. Authoritative partitions are usually created by a system expert who understands the system's structure well. The Mozilla community has not undertaken such a task. However, Michael W. Godfrey and Eric H.S. Lee [11] created such a partition for release M9 of Mozilla. As a result, we did not have to create an authoritative partition for version 1.3 from scratch. Instead, we updated the authoritative partition for version M9.

There is significant size difference between the two versions. The size of version M9 was 108MB compared to 262MB in version 1.3. In order to investigate whether big structure changes happened between the two versions, we investigated material such as the Mozilla Development Roadmap [6], and checked the system's version history.

Furthermore, we enlisted the help of the Mozilla developers by posting the top-level structure of the authoritative partition for version M9 to the Mozilla newsgroups. Gervase Markham, a community member, replied: "Massive amounts of things have changed in the more than 2 years between M9 and 1.3. The top-level divisions you have are probably still roughly right, though." As a result, we decided to keep the same top-level structure for the authoritative partition of version 1.3 as well.

Subsystem	Number of files	KLOC
HTML layout engine	1,251	478
HTML parser	46	29
Image processing library	95	91
JavaScript engine	168	122
Networking	246	74
Tools	872	390
User Interface	368	151
Programming Utilities	121	40
Platform independent layer	101	49
Cross platform interface	291	106
Total	3,559	1,530

Table 2. Mozilla 1.3 authoritative partition

Source files that can be found in both versions were placed in the subsystem they belonged to in version M9, since it is unlikely that a file was altered that drastically to warrant a move to a different subsystem. Files that are no longer there in version 1.3 were simply removed. Our main task was to place new source files to the appropriate subsystem. For this purpose, we adjusted the structure of the authoritative partition by adding or deleting susbsystems accordingly. We identified three different types of new source files:

<u>New files in old directories</u>

A feature of the authoritative partition of M9 was that all files belonging in the same directory were clustered together (in some cases, a cluster corresponding to a particular directory may have been split into more fine-grained clusters). We kept this property for the authoritative partition of version 1.3. All new files found in pre-existing directories were placed in the same cluster as their pre-existing siblings.

<u>New files in new sub-directories</u>

There exist subdirectories in version 1.3 that, even though they are in an existing directory, they serve a different function than other directories in the same parent directory. In this case, we either created new subsystems for them or placed them in an existing subsystem depending on the information we obtained from available documentation or the *module owners*.

<u>New files in new top-level directories</u>

By comparing the directory structures of Mozilla1.3 and Mozilla M9, we found the following new top-level directories in Mozilla1.3: *accessible, content, db, directory, docshell, embedding, jpeg, string, uriloader* and *tools*. New subsystems were created for all these directories. These subsystems were placed under the appropriate top-level subsystem based on information provided by the Development Roadmap and the module owners.

Table 2 provides more information on the obtained authoritative partition for version 1.3 including the size and a short description of each top-level subsystem.

4 Experiment Setup

The purpose of this paper is to investigate the effectiveness of software clustering when dynamic dependencies are used as the basis for clustering. This will be done through a number of experiments with a variety of software clustering algorithms. In this section, we present the setup for these experiments. We begin with dynamic analysis.

4.1 Dynamic analysis process

In order to extract dynamic information from Mozilla during execution, we add instrumentations when we compile the source code. Information about function invocations taking place during runtime is written in the output of *jprof*. A specially developed script replaces functions with the file that declares them, so that we can have information about dependencies between files. This script produces output in TA format, our dynamic *fact base*. We then apply a variety of filters before starting the clustering process. The purpose of these filters is explained in Section 4.3.

Next, we apply a clustering algorithm in order to group related files into clusters. Finally, we apply the MoJoFM measure in order to assess the effectiveness of the automatic decomposition with respect to the authoritative one. Figure 1 depicts the pipeline used for our dynamic analysis experiments.

The main challenge that the dynamic analysis committee faces is that it is hard to obtain a comprehensive test suite that ensures complete coverage. In fact, in many cases, and especially when dealing with systems of the size of Mozilla, a much smaller coverage is deemed sufficient.

The test suite for Mozilla was able to provide dynamic dependencies among 1,023 source files from a total of 3,559. In other words, our dynamic coverage rate was 28.74%. Although we would like to improve on this number, the size of the dynamic dependency graph was clearly sufficient for clustering purposes.

In order to ensure that the comparison to the authoritative decomposition is meaningful, we reduced it to the 1,023 source files that were part of the dynamic dependency graph. The term "authoritative decomposition" will refer to the reduced one from now on. Similarly, when clustering based on static dependencies we concentrated on the same set of files as outlined below.

4.2 Static analysis process

The static analysis process is simpler than the dynamic one. All static dependencies are extracted by Swagkit. An important question is when to remove information about source files other than the 1,023 ones in the dynamic dependency graph. We identified two possibilities:

- Remove information about the extra files from the static dependency graph before running the clustering algorithm. We will call this "static process A". Its pipeline is shown in Figure 2.
- Run the clustering algorithm on the full dependency graph. Remove information on the extra files from the algorithm's output. We will call this "static process B". Its pipeline is shown in Figure 3.

In Section 5, we present experiments that compare these two types of processes.

4.3 Filtering

One of the main features of dynamic dependency graphs is that they are weighted. For example, the number of times a particular procedure call or data reference took place during run time can be the weight of the corresponding edge. Even though weights can be assigned to static dependencies as well, e.g. when a procedure includes multiple references to the same variable, the variance of dynamic weights is significantly larger. Such information should be of great value to a clustering process.

Unfortunately, most of the software clustering algorithms presented in the literature do not take edge weights into account. In order to overcome this problem, and be able to take advantage of edge weights, we implemented the filtering approach described below.

Only edges with particular weights pass through the filter. In other words, edges with "undesirable" weights are removed from the dependency graph. This way we can emphasize the importance of exactly the set of edges we are interested in. We identify two interesting types of filters:

- **1 - k**: This filter allows only edges with weight less than a particular threshold. Successive filters of this kind can help us investigate the importance of edges with large weight.
- **k - max**: This filter allows only edges with weight larger than a particular threshold (max refers to the maximum weight in the complete dynamic dependency graph). Successive filters of this kind can help us investigate whether removing edges of small weight is beneficial to the clustering process.

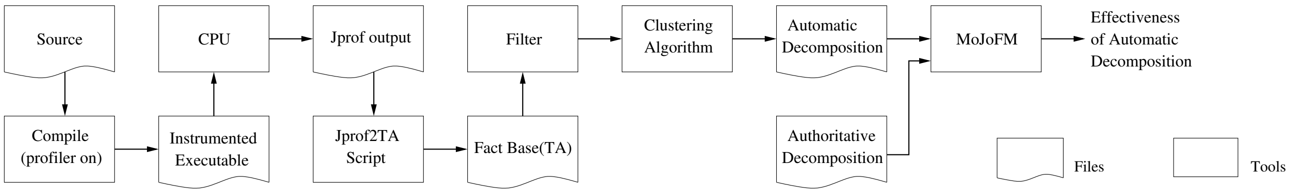


Figure 1. Dynamic analysis process

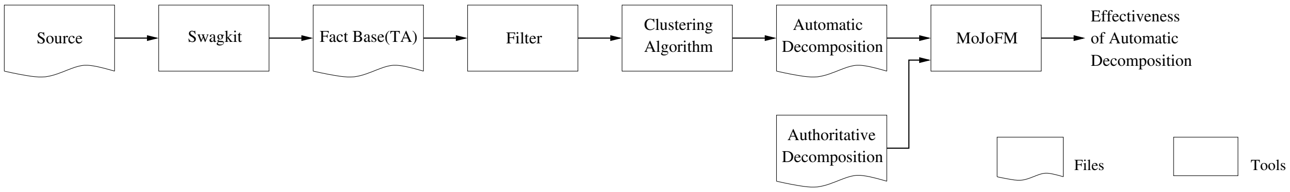


Figure 2. Static process A.

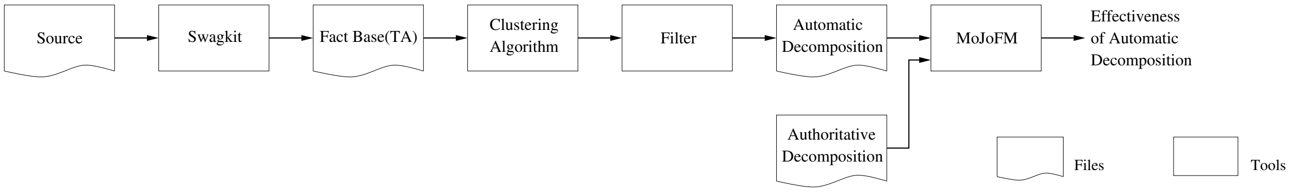


Figure 3. Static process B.

We should note that if a filter removes all edges attached to a particular node, this node is also removed from the graph.

The static processes in Figures 2 and 3 indicate that there are filters for the static fact base as well. These filters are only there to ensure that the static and dynamic clusterings produced refer to the same set of source files. As a result, they do not remove edges based on weight (there are no weights in the static dependency graph), but they remove exactly the same set of nodes that the corresponding dynamic filter removes.

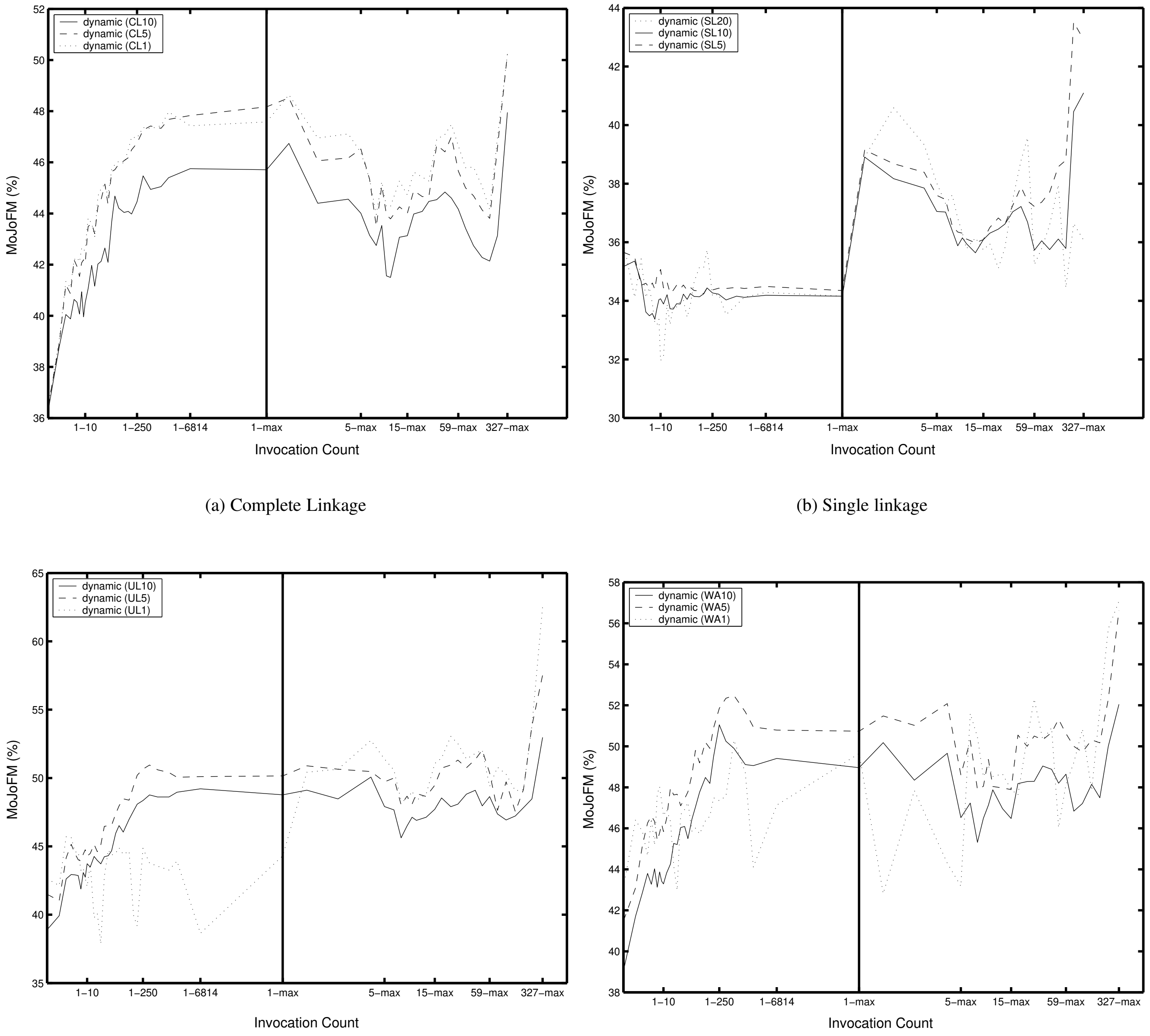
In the next section, we present experimental results using the processes outlined above.

5 Experiments

We conducted four different types of experiments. In all experiments, we applied successive filters of both kinds presented above. The first part of the experiment included successive **1-k** filters, where k ranged from 1 to max. The **1-max** filter is, of course, the identity filter. For the second part of each experiment, successive **k-max** filters were applied. The value of k was chosen so that the resulting dependency graph was not reduced to less than 20% of its original size.

5.1 Cut-point heights

The first set of experiments had to do with determining an appropriate cut-point height for the hierarchical algorithms. We run all four of them with a variety of cut-point heights. The results are presented in Figure 4. As can be seen from these diagrams, the cut-point height that performed the best in our case was that of 0.05 (marked on the graphs as CL5, SL5 etc.) For this reason, this is the cut-point height that we will use for the rest of the experiments.



(a) Complete Linkage

(b) Single linkage

(c) Unweighted average

(d) Weighted average

Figure 4. Cut-point height comparison results.

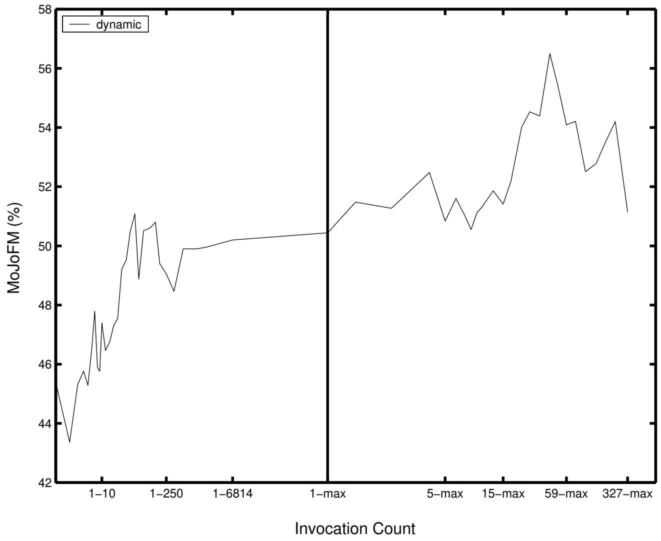
5.2 Dynamic analysis process

In this set of experiments, we experimented with all clustering algorithms presented in Section 2, and observed the effect of the various filters to the effectiveness of the obtained decompositions. The results of these experiments are shown in Figure 5. Note that the x-axis is in logarithmic scale (this is the case for all experimental results presented in this section).

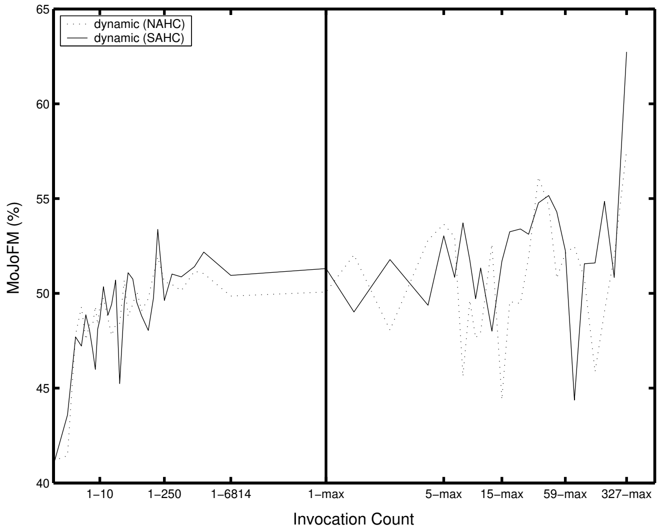
There are some interesting observations to be made:

As k increases in the first part of the graph, the effectiveness of the obtained decomposition also increases. This clearly indicates that removing edges of large weight from the dependency graph is detrimental to the clustering result.

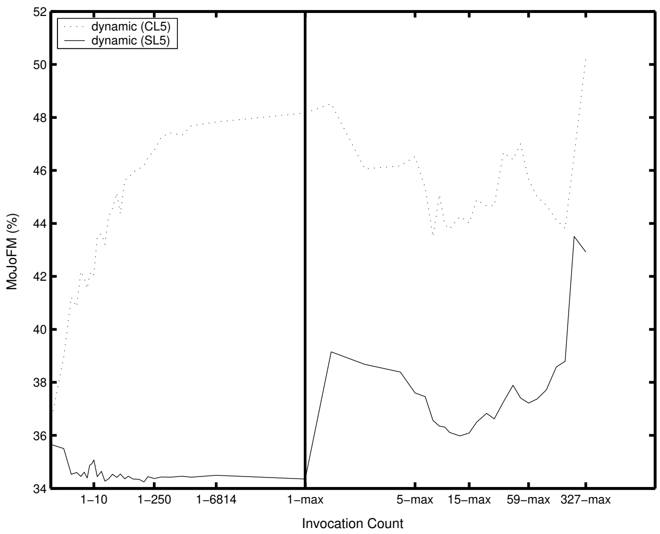
On the other hand, removing edges of small weight from the dynamic dependency graph (as shown in the second part



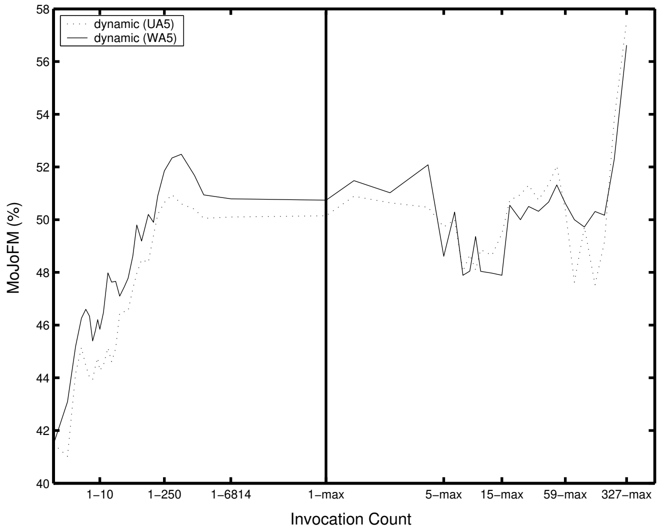
(a) ACDC



(b) Bunch



(c) Complete and single linkage



(d) Weighted and unweighted average

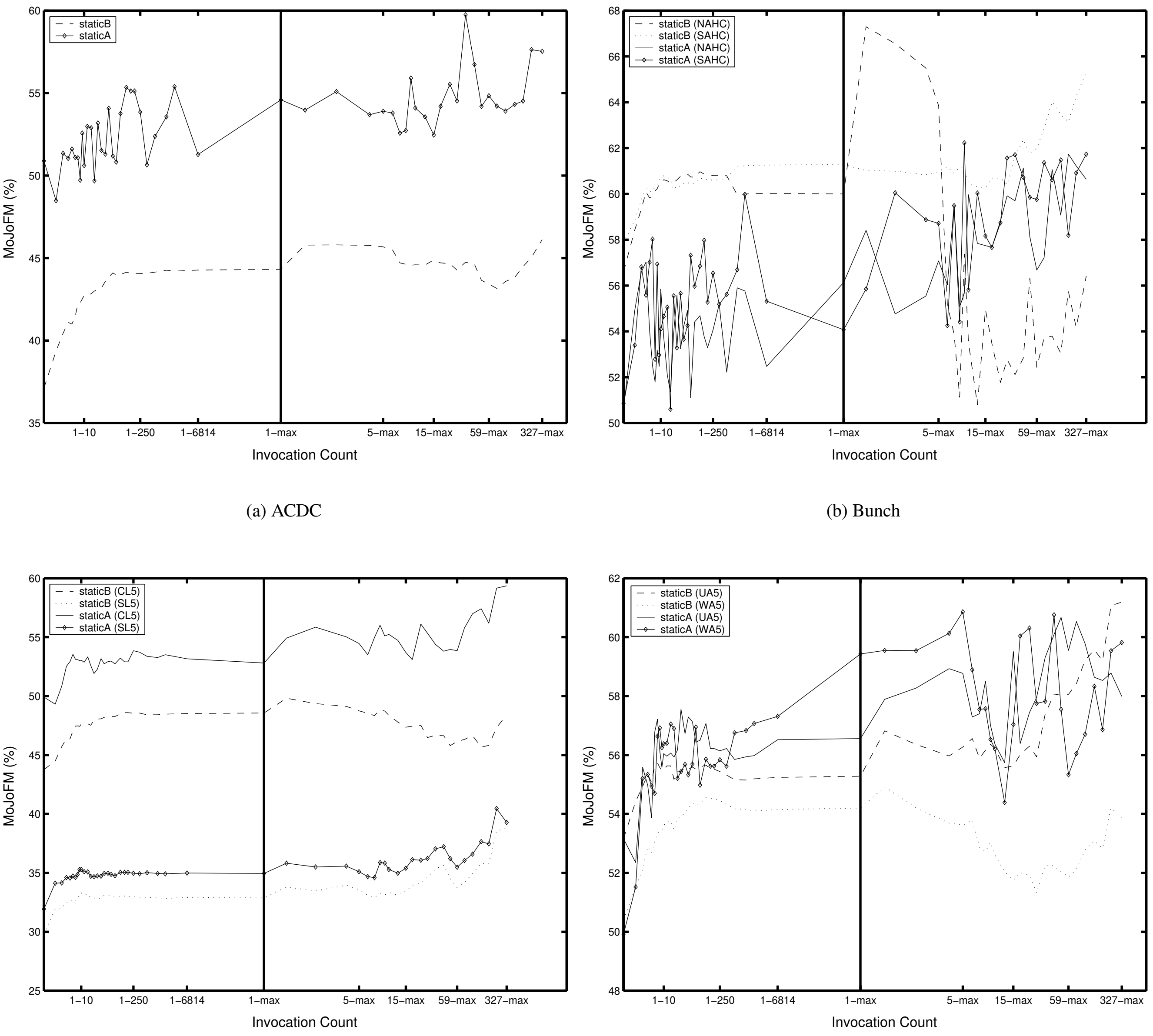
Figure 5. Dynamic analysis process results.

of the graphs) does not cause a significant problem. In fact, in many cases the resulting decomposition has improved considerably. This seems to indicate that the weights in the dynamic dependency graphs are significant to the clustering process. It is quite possible that if software clustering algorithms are developed to take advantage of these weights, they will produce better results than the existing algorithms.

5.3 Comparison of the two static processes

This set of experiments compares the two static processes presented in Section 4.2. As can be seen in Figure 6, static process A performs consistently better than process B, with the exception of the two Bunch algorithms.

This can be explained by the fact that in static process A the removal of extraneous nodes takes place before the



(a) ACDC

(b) Bunch

(c) Complete and single linkage

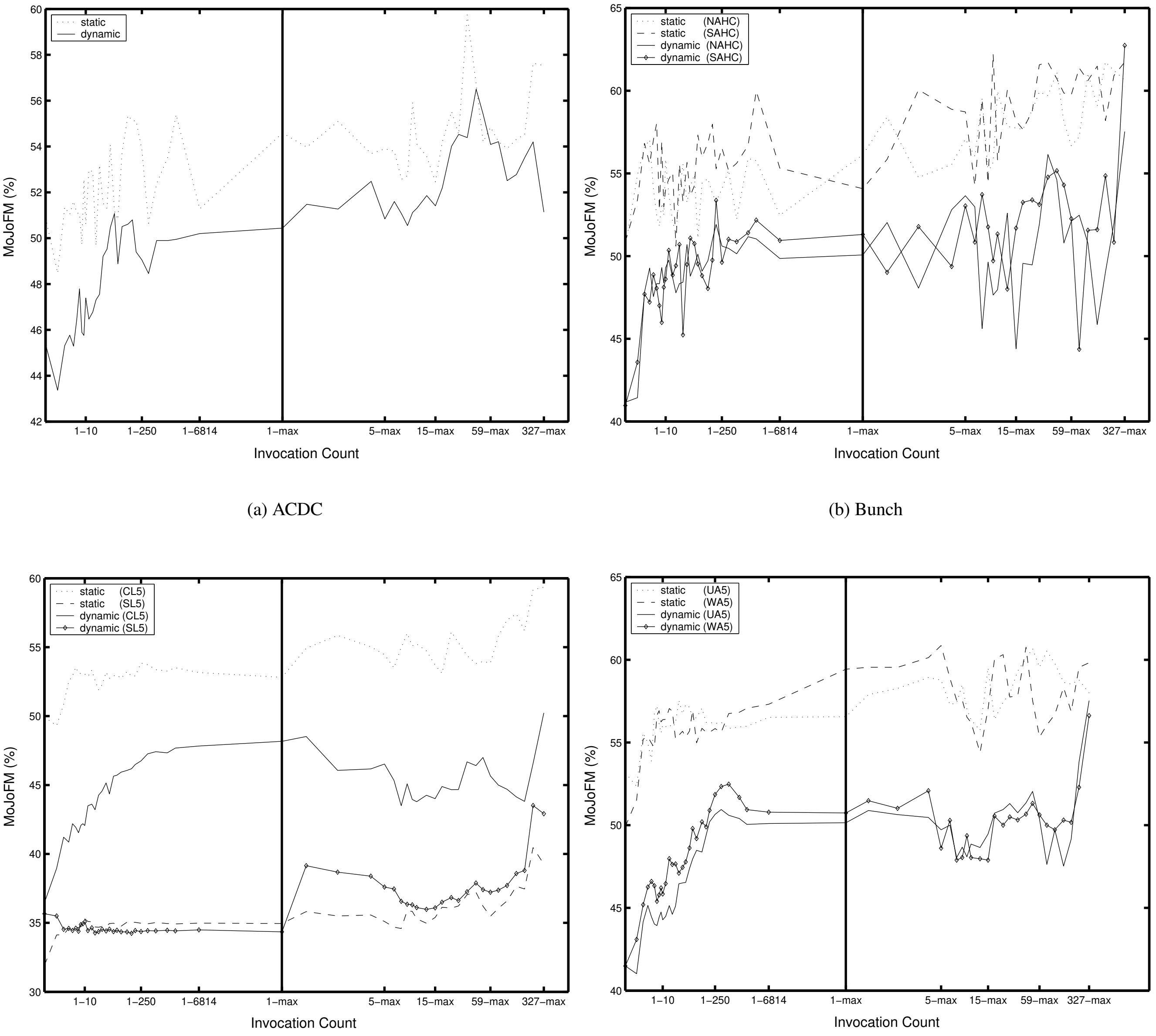
(d) Weighted and unweighted average

Figure 6. Results for the comparison of the two static processes.

clustering algorithm is run. The clustering algorithm is then allowed to produce a meaningful decomposition of the remaining nodes. In static process B, the extraneous nodes are removed after the algorithm has run with no consideration for the status of the obtained decomposition.

The fact that the behaviour was different in the case of Bunch was quite intriguing. This might be explained by the fact that Bunch is a randomized tool, and therefore unstable [23]. However, further investigation is required in this regard.

Since static process A performs better in most of the cases, it is the process we chose for the comparison of static and dynamic results.



(a) ACDC

(b) Bunch

(c) Complete and single linkage

(d) Weighted and unweighted average

Figure 7. Comparison of static and dynamic clusterings.

5.4 Comparison of static and dynamic clusterings

In this set of experiments, we compared the clusterings produced by the dynamic process to the ones produced by the static one. The results can be seen in Figure 7.

The static process performs usually better. However, there are cases where the dynamic clusterings are the ones with the higher quality. Also, the average difference between the static and dynamic clusterings was less than 5%.

While this result may seem disappointing at first, there are many reasons to explain it:

- The clustering algorithms we experimented with were developed with static dependencies in mind. It is possible that biases exist in their design or implementation. Furthermore, if they were designed to handle weighted dependencies we might have seen better results.

- The authoritative decomposition used for these experiments was definitely biased towards the static view. A large part of it was based on directory structure, clearly a static decomposition criterion. It is still quite likely that the dynamic clusterings produced during these experiments are at least as helpful as the static ones. Further investigation is required to establish whether this is true.

- The dynamic dependency graph we obtained was of significant size but by no means complete. We believe that a fuller dynamic dependency graph might give the dynamic clusterings a better chance.

In view of all these factors, we believe that the dynamic analysis process has performed quite well. We hope that software clustering algorithms will be developed to take advantage of the distinct nature of dynamic clusterings.

6 Conclusions

This paper presented an approach for the evaluation of clusterings based on dynamic dependencies between software artifacts. We also presented several experiments that demonstrated that dynamic clusterings have merit. It is our belief that developing software clustering algorithms that take advantage of the distinct nature of dynamic dependencies is an avenue of research worth investigating.

References

[1] `http://www.swat.uwaterloo.ca/~swagkit`.

[2] `http://swag.uwaterloo.ca/~cppx/`.

[3] `http://www.gupro.de/GXL/`.

[4] `http://www.mozilla.org/performance/tools.html`.

[5] `http://www.mozilla.org`.

[6] `www.mozilla.org/roadmap.html`.

[7] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *Proceedings of the Tenth Working Conference on Reverse Engineering*, pages 334–344, Nov. 2003.

[8] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, Oct. 1999.

[9] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering*, pages 3–14, Mar. 2004.

[10] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[11] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Second International Symposium on Constructing Software Engineering Tools*, June 2000.

[12] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, Aug. 1985.

[13] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 201–210, June 2000.

[14] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, May 1997.

[15] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 2003.

[16] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[17] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the International Conference on Software Maintenance*, pages 744–753, Nov. 2001.

[18] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the International Conference on Software Maintenance*, pages 408–419, Nov. 2001.

[19] R. W. Schwanke and M. A. Platoff. Cross references are features. In *Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[20] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 170–179, Oct. 1997.

[21] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, Oct. 1999.

[22] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, Nov. 2000.

[23] V. Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 211–218, June 2000.

[24] Z. Wen and V. Tzerpos. An optimal algorithm for MoJo distance. In *Proceedings of the Eleventh International Workshop on Program Comprehension*, pages 227–235, May 2003.

[25] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the Twelfth International Workshop on Program Comprehension*, pages 194–203, 2004.