ISBN : 978-1-60558-426-3 http://doi.acm.org/10.1145/1506216.1506228

Computing Dynamic Clusters

Philippe Dugerdil, Sebastien Jossi
Dept. of Information Systems
HEG-Univ. of Applied Sciences of Western Switzerland
7 route de Drize, CH-1227 Geneva, Switzerland
+41 22 388 17 00
philippe.dugerdil@hesge.ch, sebastien.jossi@hesge.ch

ABSTRACT

When trying to reverse engineer software, execution trace analysis is increasingly used. Though, by using this technique we are quickly faced with an enormous amount of data that we must process. While many solutions have been proposed that consist of summarizing, filtering or compressing the trace, the lossless techniques are seldom able to cope with millions of events. Then, we developed a dynamic clustering technique, based on the segmentation of the execution trace that can losslessly process such a large quantity of data. In order to compute the clusters of classes we use a maximal clique computing algorithm. After having presented our technology we show experimental results highlighting that it is robust with respect to the segmentation parameters. Finally we present the tool we developed to compute dynamic clusters from execution traces.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement - *Restructuring, reverse engineering, and reengineering.*

General Terms: Design, Experimentation, Algorithm.

Keywords

Reverse-engineering, software architecture, software clustering, dynamic analysis.

1. INTRODUCTION

To extend the life of a legacy system, to manage its complexity and decrease its maintenance cost, one option is to reengineer it. However, reengineering initiatives that do not target the architectural level are more likely to fail [1]. Consequently, many reengineering initiatives begin by reverse architecting the legacy software. The trouble is that, usually, the source code does not contain many clues on the high level components of the system [2].

However, it is known that to "understand" a large software system, which is a critical task in reengineering, the structural aspects of the software system i.e. its architecture are more important than any single algorithmic component [3].

ISEC'09, February 23–26, 2009, Pune, India.

Therefore, a good architecture is one that allows the observer to "understand" the software. To give a precise meaning to the word "understanding" in the context of reverse-architecting, we borrow the definition by Biggerstaff et al. [4]: "A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program". The first step is therefore to recover the architecture of the software and then try to explain this architecture in comparison to the functionality of the code.

Many techniques have been proposed to recover the architecture of legacy software by computing clusters of classes i.e. sets of classes that are tightly coupled [16][18][19][21][23]. While most of the published work rest on the static analysis of the code, we recently proposed to focus on the execution trace to find clusters of dynamically correlated classes [26]. These clusters represent the classes that work closely together when executing a given scenario associated to a use-case. Since the use-cases are associated to business function, we have a way to associate the clusters of classes to business functions. However, the execution trace files of all but trivial programs are generally very large. For example, in one of our experiments, we got a file with more than 7 millions of events (method calls). Although many authors try to cope with the quantity of information to process by compressing the trace using more or less sophisticated techniques [25], we have developed another technique: trace segmentation. In the latter, the trace is split into contiguous segment of equal size and we observe the presence or absence of each of the classes in each of the segments. A class is said to be present in a segment if there is at least one call to one of its methods in the segment. Let us define the number of segments in the trace as Ns and the binary occurrence vector V_C for a given class C as a vector whose size is Ns and whose i^{th} component indicates the presence (1) or absence (0) of the class in the i^{th} segment (figure 1). An occurrence vector can then be associated to each class of the system and for each scenario. From the occurrence vectors of the classes we can compute the *dynamic correlation* between the classes of the system in the context of the executed scenario. In fact, if two classes are simultaneously present or absent in the same segments, then they are considered as correlated.

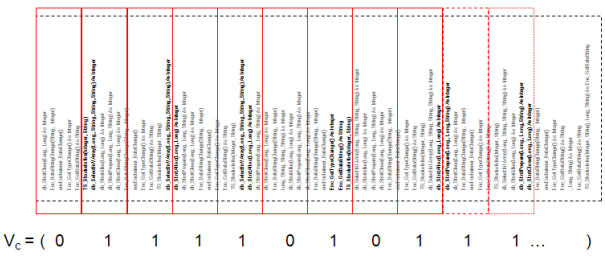

Figure 1: Trace segmentation and occurrence vector

The correlation between any two classes C_1, C_2 is given by [26]:

$$\text{correlation}(C_1, C_2) = \frac{V_1.V_2}{\sum_{i=1}^{Ns} V_1[i] \oplus V_2[i]} \text{x100}$$

Where V_1, V_2 are the occurrence vectors of the classes C_1 and C_2, $V_1.V_2$ is the usual dot product for vectors and $V_1[i] \oplus V_2[i]$ is the Boolean OR operator between the corresponding components of both vectors. Since the dot product as well as the $\oplus$ operator between vectors are symmetric, this correlation function is symmetric: correlation(C_1,C_2) = correlation (C_2,C_1).

The value of the correlation is represented by an integer between 0 and 100. Two classes are considered strongly correlated if their correlation is higher or equal to some predefined threshold T. Since the scenarios are instances of use-cases that represent business functions, the clusters represent sets of classes working closely together to implement some business function. In this paper we first present a detailed account of our technique and the algorithm used to identify the functional components in legacy software. Then we present an assessment of the robustness of our technique through several experiments. This paper is organized the following way. Section 2 presents our definition of a component and the correlation graph. Section 3 presents the maximal clique computation algorithm that we use to compute the clusters from the execution trace. Section 4 presents the result of its application on a large execution trace, analyzes the robustness of our technique and discusses its performance. Section 5 presents the trace analysis tool we developed in Java and shows examples of use. Section 6 concludes the paper and gives some hints on future work. Section 7 discusses the related work.

2. CORRELATIONS GRAPH

Definition: *component***.**

Let $\mathcal{C}$ be a set of classes and correlation: $\mathcal{C} \times \mathcal{C} \rightarrow [0..100]$ be our correlation function between two classes. A *component* is a maximal subset K of $\mathcal{C}$ such that all classes in K are mutually strongly coupled. In other words a component K must satisfy the following two constraints:

1. $\forall x,y \in K$, correlation(x,y) $\geq$ T
2. $\neg\exists z \in \mathcal{C} \setminus K \mid \forall x \in K$ correlation(x,z) $\geq$ T

Where T is a predefined correlation threshold corresponding to a high correlation between classes.

Corollary: starting with a correlation function and a correlation threshold T the computation of the components is unique.

Proof: trivial by definition, since one computes the *maximal* subsets of $\mathcal{C}$.

Definition: a *functional component* is a component computed from an execution trace, or a set of execution traces, corresponding to the execution of use-case scenarios.

The component is called *functional* because it is involved in the implementation of well defined business function (as represented by the use-case [28]). Besides, the classes implementing a functional component should be highly cohesive and strongly coupled. Following the execution of a use-case, we can compute the correlation matrix of the classes. In such a matrix, each cell represents the correlation between the classes represented by the row and column headers. This matrix is obviously symmetric. Figure 2 presents such a matrix where the highly correlated classes (i.e. whose mutual correlation value is higher or equal to a given threshold T) are highlighted. Besides, in all our experiments, we saw that some classes occurred in almost all the segments of the execution trace. These classes are therefore not specific to any step in the scenario and perform some utility work. They are similar to the utility classes in the work of Hamou-Lhadj and Lethbridge [29]. Since we need to discover the components that implement specific business functions these classes, that we call "temporally omnipresent classes", are filtered out before proceeding with the computation of the clusters [26].

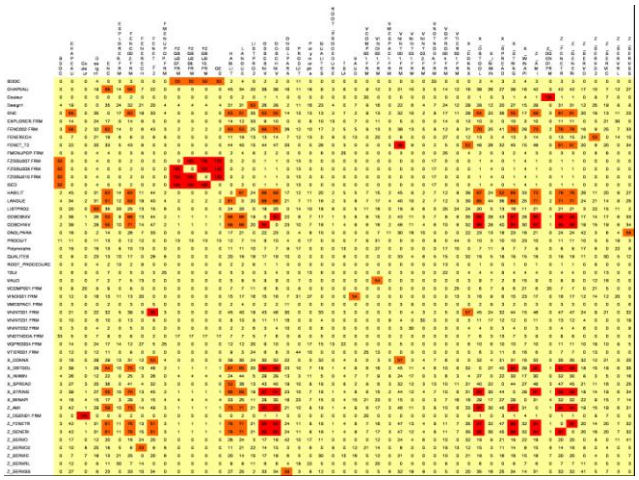

Figure 2. Classes correlation matrix

Let us define G = ($\mathcal{C}$,R) a weighted graph whose set of nodes $\mathcal{C}$ is the set of classes identified in an execution trace and whose edges are defined by the correlation R between these classes (figure3).

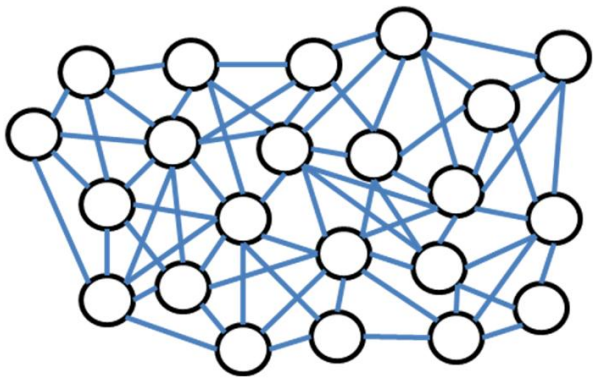

Figure 3. Correlation graph

The weight of an edge is the strength of the correlation between the connected nodes. The computation of a component is then equivalent to the computation of a maximal clique in the partial subgraph G' of G whose edges' weight is greater or equal to the threshold T. Depending on T, G' may take the form of a set of separated connected component as presented in figure 4.

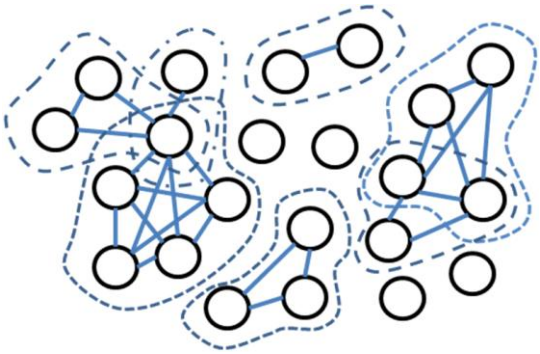

Figure 4. Connected subgraphs of G and components

Since a given class can be part of several maximal cliques, it can be part of several components. In figure 4 we surrounded the classes forming components with a dotted line.

3. CLUSTERING ALGORITHM

Let us have:

- $\mathcal{C}$ the set of classes whose methods are invoked in the execution trace.
- correlation: $\mathcal{C} \times \mathcal{C} \rightarrow [0..100]$ the correlation function
- $\mathcal{G}$ the corresponding correlation graph.

The computation of all the clusters in $\mathcal{C}$ amounts to finding all the maximal subgraphs whose nodes are mutually strongly connected (whose edge's weight is ≥ T). In other words, if one creates a new graph $\mathcal{G}'$ by removing all edges whose weight is < T from $\mathcal{G}$, then the search for the clusters is similar to finding the largest complete subgraphs (maximal cliques) in $\mathcal{G}'$. However $\mathcal{G}'$ is usually not connected (see for example the situation depicted in figure 4). Therefore, to speed up the search for clusters, one will work on the separate connected components of $\mathcal{G}'$.

Definition: following [5] we define Γ(n) as the neighborhood of a node n in $\mathcal{G}'$ **(**i.e. all the nodes adjacent to n in $\mathcal{G}'$**)**.

Since $\mathcal{G}'$ is not connected, the cluster finding algorithm can be applied separately to all connected components. A standard algorithm to find all the maximal cliques of a graph G of order n is presented in [5] :

```
Clique(C, i) {
if ( i > n )
   then { recordClique(C) }
   else {
      if ( C ∩ Γ(i) = C )
         then { Clique( C ∪ { i }, i+1) }
         else  {
            Clique( C , i+1)
            if( C ∩ Γ(i) ∪ { i } is maximal in G_i )
               then Clique(C ∩ Γ(i) ∪ { i } , i+1)
            }
      }
}
```

In this algorithm the nodes are numbered 1..n and G_i is the subgraph of G containing the nodes 1 to i. The function *recordClique(C)* records C as a new maximal clique of G. The algorithm is launched with: **Clique**({ 1 } , 1).

This recursive algorithm records all maximal cliques of G_i *at least once*. Its space complexity is polynomial according to the size of G_i [5].

4. RESULTS & ROBUSTNESS ANALYSIS

4.1 Example of clustering

To evaluate the quality of our clustering technique we need to define a benchmark. Then, we decided to apply our clustering technique to a recently written, well architected software system written in Java. This system holds more than 600 classes. By interacting with its developers, we knew these packages to represent well defined functional components. Therefore, if our technique was able to discover the functional components of the system, then there would be a strong match between the recovered component and the package structure. Ideally, the recovered clusters should then each be located in a single package. Therefore when faced with an unknown legacy system, we could apply the technique to recover its functional architecture: we would know that the computed clusters would be functional components. Figure 5 represents the concept of cluster span. The situation on the top present a cluster with 4 classes that spans 2 packages. The situation on the bottom represents two clusters located in single packages (i.e. they are called "single-package" clusters).

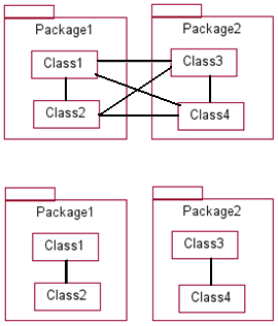

Figure 5. Cluster span

The packages of the chosen benchmark application represent entities in the domain. Structurally, they are all located at the same level, just below the root package of the application. Empirically, we determined that the number of segments of an execution trace should be set according to the number n of classes occurring in the trace [10]. In this example, we performed the clustering on an execution trace with 600'000 events, using Ns = 32*n and T = 90%. These are the parameters we empirically found to provide the best clustering results [10]. In other words, the best clustering is obtained with the number of segment equal 32 times the number of classes and with a coupling between classes of 90%. We then uncovered 35 independent clusters among which 31 were located in single packages and only 4 span two entity packages. The result is presented in figure 6 that contains all the packages representing entities in the software. The 15 darkened packages are the one containing "single package" clusters. Obviously some packages contain more than one cluster. In other words, in this experiment, 89% of the identified clusters matched the packages.

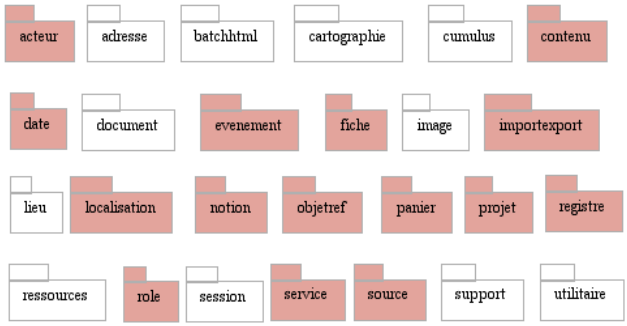

Figure 6. Packages containing single-package clusters

In figure 7 we present in different colors the packages that contain the clusters spanning 2 packages. It is worth noting that our technique will not cluster all the classes of the execution trace, since we are looking for strongly correlated classes. But we expect the clustered classes to represent functional components. In this experiment we were able to cluster 67% of the classes found in the execution trace. Among them, 89% matched functional components.

This is a very encouraging result if one takes into account the simplicity of the technique.

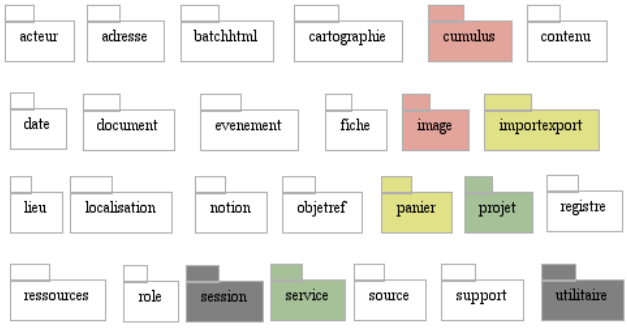

Figure 7. Packages containing 2-packages clusters

4.2 Segment boundary sensitivity

To rely on this clustering technique, we must evaluate the robustness of the match between the clusters and the packages with respect to the segmentation parameters. In fact, since the technique is to split the trace into contiguous segments and check the binary occurrence of the classes in each segment, we may think that the location of the boundary of the segments plays an important role in the result of the clustering, as shown conceptually in figure 8. In the left part of the figure, the boundaries of the segments are located in between blocks of calls to the same classes. In the right part, the boundaries are shifted one event to the left. This has an important impact on the binary occurrence of the classes in each segment hence the corresponding occurrence vectors. Therefore, the correlation between Class1 and Class2 would be 0 in the situation represented on the left but 66% on the right. Since we cannot guarantee to generate exactly the same sequence of events each time the execution trace of a given scenario is recorded, we may think that the result of the clustering would be very sensitive to the sequence. Therefore we must perform a sensitivity analysis of our technique.

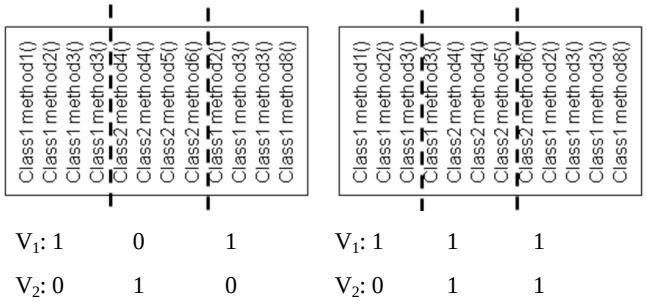

Figure 8. Sensitivity to the segment's boundaries

4.3 Sensitivity study workflow

Starting from the source code, it is first instrumented to be able to generate the execution trace. The result is compiled and executed on the target platform and the execution trace file is generated. This file is then analyzed to identify the clusters. The set of clusters is finally matched against the packages found in the source code and some matching metrics are computed (figure 9).

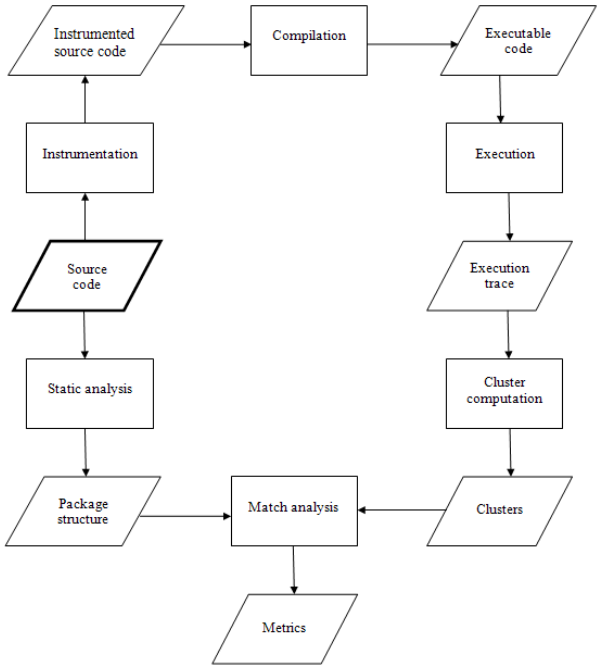

Figure 9. Workflow to assess the quality of the match

In order to determine the robustness of our dynamic clustering technique we performed several computations of the clusters by shifting the boundaries of the segments and observing the impact on the results of the clustering. As for the number of segments Ns and the correlation threshold T, we used again the settings that worked best according to our empirical assessment: Ns = 32*n and T = 90%, where n is the number of classes occurring in the execution trace. [10]. The robustness analysis has been performed on 2 execution trace corresponding to two scenarios from two different use-cases. The first trace, corresponding to the first use-case (UC1), contains 7 million of events and the second trace, corresponding to the second use-case (UC2), contains 600'000 events. For each execution trace, we first computed the clustering using the standard technique. Next we performed the same computation after having

shifted the start of the segmentation by 1, 5, 10, 20 and 50% of the segment size as represented conceptually in figure 10.

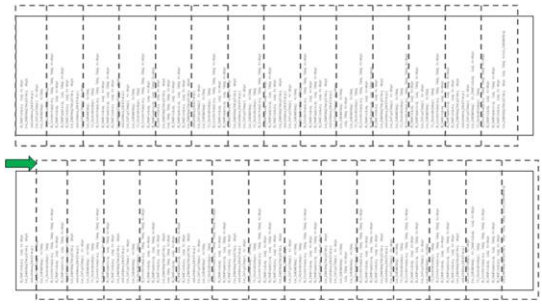

Figure 10. Shifting the segment boundaries

Then the result of the clustering was compared with the original one using different metrics. The first and foremost metric is the number of packages the cluster span. In fact, the goal is to have a maximum of clusters located in a single package since these represent functional components. Tables 1 and 2 show the number of clusters and the span for each execution trace. The row header represents the number of packages the clusters span.

Table 1. Clusters for the trace from the first use-case

span	Orig.	1%	5%	10%	20%	50%
1	17	16	16	15	15	16
2	9	6	6	5	4	4
3	2	5	4	4	3	3
>3	4	2	2	2	3	3

Table 2. Clusters for the trace from the second use-case

span	Orig.	1%	5%	10%	20%	50%
1	31	35	34	31	31	32
2	4	4	4	4	6	6

For example, the second row represents the number of cluster that span 2 packages. The column with header "Orig." represents the number of clusters for the original segmentation. The column with header "1%" represents the number of clusters in each category for the segmentation with a shift of 1% of the segment start and so on. However, since the classes are heavily interacting to implement the use-cases we expect some inter-packages coupling. This is why we cannot hope all the clusters to span only one package. The first observation we can make for both traces is that the shift of segmentation start barely impacts the result of the clustering, especially if we focus on the "single-package" clusters. These are the most interesting ones since they exactly correspond to functional components. Another important metric to assess the quality of the clustering is the coverage of the classes present in the trace by the clusters. We then compute the ratio of the classes that have been clusterized to the total number of classes in the trace. We then compare this coverage ratio among the experiments with the different segmentation starting points. These are shown in figure 11. (UC1 represents the execution trace from the first use-case and UC2 the execution trace from the second use-case). We observed that the class coverage is rather insensitive to the shift of segmentation start since the difference in results stayed within 3 percent. Normally the more the coverage the better, provided that the clusters hold a "significant" number of classes. In other words, we would not be happy with a large coverage by "atomic" clusters of minimal size. This is why we also checked the average and the standard deviation of the number of classes per cluster.

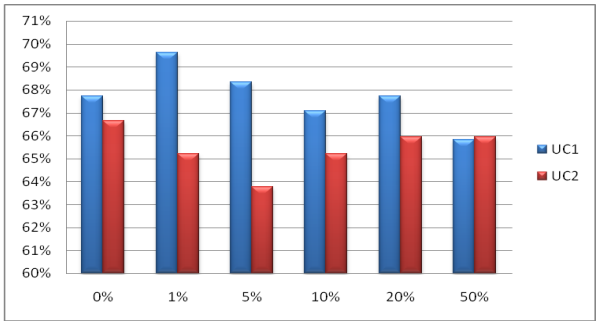

Figure 11. Coverage ratio

The comparison of these metrics is presented in figures 12 and 13 respectively. We clearly see that the impact of the start of the segmentation to the results of the metrics is even smaller than for the previous ones. The last metric we computed is the number of "functional components" identified using our segmentation technique.

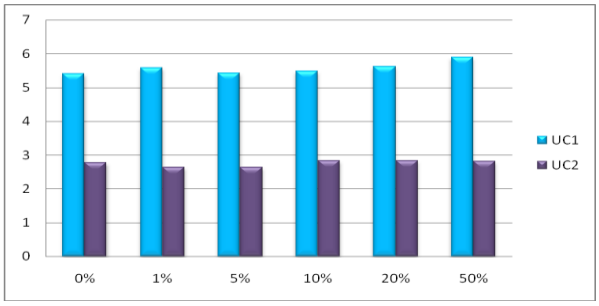

Figure 12. Average classes per cluster

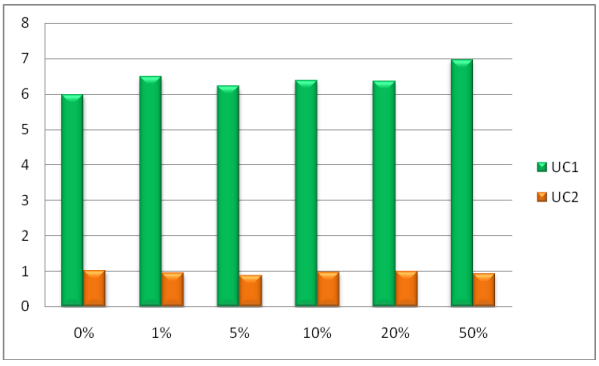

Figure 13. Standard deviation classes per cluster

These are the packages that contain "single-package" clusters (fig. 14). Out of the 26 packages contained in the application, we

identified 12 packages containing "single-package" clusters for the trace from UC1 and between 14 and 15 for the trace from UC2. In fact, whatever the segmentation starting point, the results were almost always the same. Furthermore, not only was the number of packages equal, but also the packages themselves were exactly the same. The only missing one for UC2 in the experiment with 1% to 20% shift was also the same.

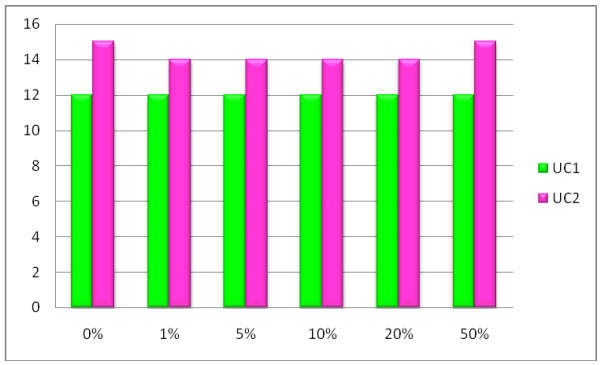

Figure 14. Number of packages in clusters of 1 package

4.4 Performance

Our clustering technique is efficient even with very large execution traces. For example, the preprocessing of the largest trace (7 millions of events) and its loading into an Oracle database table takes about 30 min on a standard PC (3Ghz, 2Gb). The clustering itself takes about 20 seconds. So far, we have not found in the literature other dynamic clustering techniques that can cope with traces as big as that. Usually, the research papers show results based on traces containing a few tens of thousands of events, rarely beyond 100'000. Our technique can easily cope with hundred times more. As far a trace generation is concerned, we use an instrumentation technique that can be applied to whatever programming language, provided that is has a well defined and unambiguous BNF grammar (i.e. can be parsed using the JavaCC (YACC) technology). The source codes of the programs to analyze are then modified to insert trace generation information. Depending on the programming language and the architecture of the legacy system, the performance penalty can be more or less large. In our experiment with Java, the instrumented code was on the order of 2 times slower than the original one. When we applied our technology to a large client-server application written in Visual Basic, the impact was on the order of 50 times! However, it is clear that the trace generation performance impact would apply to whatever dynamic clustering technique.

5. CLUSTER ANALYSIS TOOL

5.1 Introduction

To analyze an execution trace and compute dynamic clusters we developed a trace analysis tool in Java under Eclipse. As a first step, the trace file must be preprocessed and loaded into the database. In this step we rebuild the call tree and also correct the missing information. The trace file contains sequences of method call and ends of calls (represented by the keyword END). A rough example of the trace file together with the corresponding call tree is presented in figure 15. $F_i()$ represents the signature of the method called. END $F_i()$ represents the end of the execution of method $F_i()$. Due to the recording of the end of method execution, we can unambiguously reconstruct the call tree (in single-threaded application).

```
F1()
F2()
F3()
END F3()
F4()
F5()
END F5()
END F4()
END F2()
END F1()
```

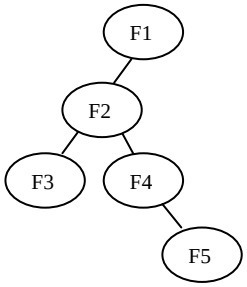

Figure 15. Trace file and corresponding call tree

In the instrumented code, the method call writing statement is the first statement executed when the method is entered. Unless there is an IO exception while writing to the file, the calls will always be recorded. However, there is one situation where a method call will not be recorded: when the call happened before we started to record the trace. Therefore there will be an end of execution without the corresponding call higher in the trace file. Here is an example:

```
F1()
F2()
END F2()
F3()
END F3()
END F1()
END F0()      // not matched by a call.
```

Moreover it might also be the case that the end of a method execution is missing. In fact, the end of execution writing statement must be the last executed in a method. But there are many ways to exit a method. In particular, in the case of un-catched Java exception, the control returns to the calling method directly. The remaining statements in the called method are not executed. Therefore, our trace reading mechanism must deal with missing calls and missing end of execution and be able to recover the proper call tree. Once the trace is loaded into the database we can proceed with the computation of clusters. Moreover, based on the call tree, we can perform many kinds of analysis including the dynamic Fan-in and Fan-out of classes in a similar way to what is done using static techniques [29].

5.2 Tool interface

When using the tool, one first selects the trace to analyze among those available in the database. Figure 16 presents the main screen of our trace analysis tool. In this example we selected a very small trace (26085 events). Most of the analysis techniques, but cluster identification, can be applied to the whole trace or to a specific subpart of it, called the analysis window. The analysis window is positioned using the two sliders displayed in the middle of the interface: one can select its width (in number of events) and its position. The top text pane presents the classes found in the trace file and the bottom text pane the result of the currently selected analysis. In figure 16 we present the dynamic Fan-out: the classes that are called by the selected class as well as the number of calls. Next we can display the occurrence vector of selected classes. Then, we must select the number of segments in the trace file. This is done using the upper slider (in this case the lower slider is disabled). If one

selects more than one class, the result pane displays the correlation factor between the classes as shown in figure 17.

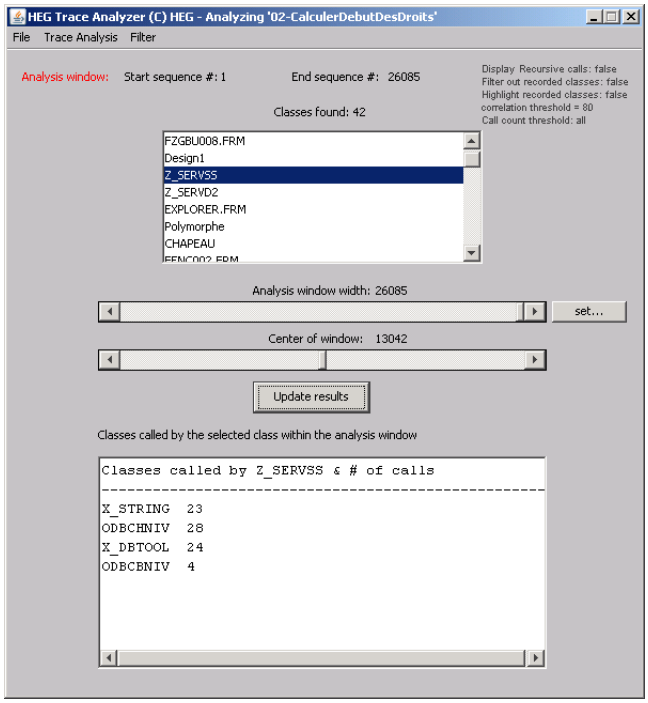

Figure 16. Trace analysis tool

The occurrence vector is represented by a sequence of vertical bars "|" meaning class presence (1) and dots "." meaning class absence (0) for the corresponding segment of the trace.

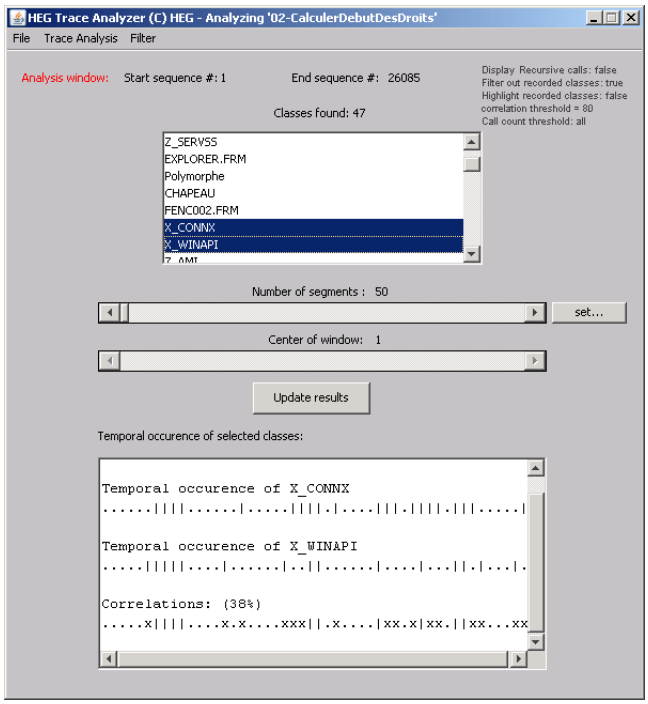

Figure 17. Occurrence vectors and dynamic correlation

The correlation is displayed as a sequence of vertical bars (meaning simultaneous presence), dots (meaning simultaneous absence) and crosses (meaning mismatch). Finally, we can compute the clusters among the selected classes. Again, the number of segment must be selected using the upper slider. The result is presented in figure 18. We can see that there are 5 clusters among all the classes identified in the trace file, when using 94 segments.

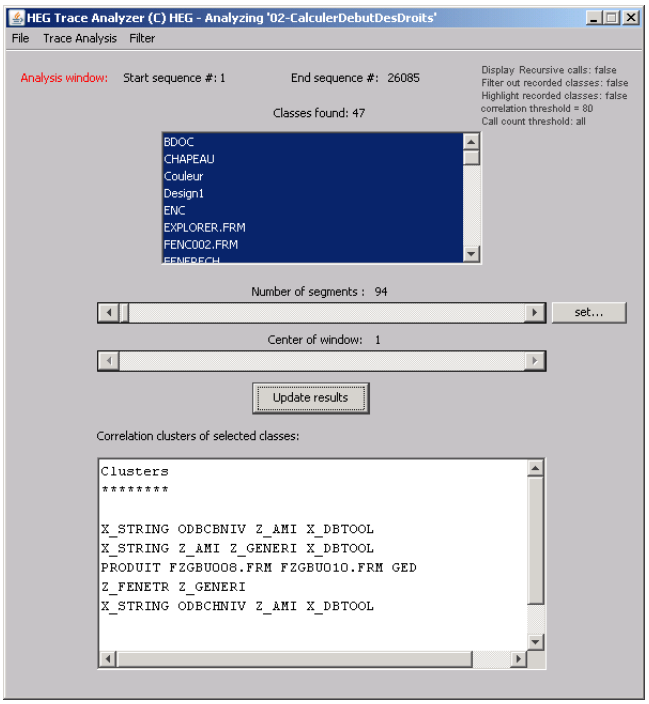

Figure 18. Cluster computing

The result of whatever analysis can be displayed graphically, thanks to the *dot* tool from the open source graph visualization software Graphviz [30]. Figure 19 presents the result of the computation of clusters as graphs.

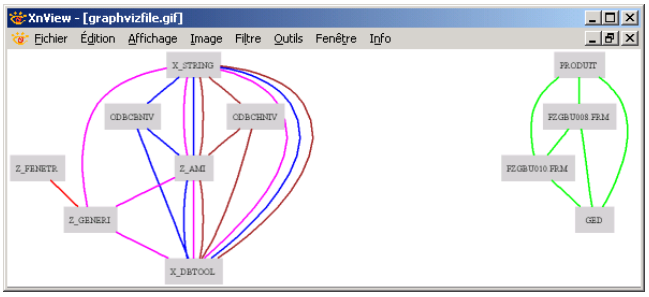

Figure 19. Clusters presented graphically

6. CONCLUSION

In this paper we presented the dynamic clustering technique that we use to recover the architecture of a legacy system from the analysis of the execution traces. First we presented the way we compute the clusters of classes that are likely to represent functional components, together with associated algorithm. In this context, a functional component is a maximal clique in the correlation graph representing

the dynamic correlation between classes. We then use a standard maximal clique computation algorithm from graph theory. To show that our clustering technique is robust we performed a sensitivity analysis to study the influence of the shift of the segmentation start to the result of the clustering. We presented the workflow of this analysis together with the metrics used. This analysis has shown that the resulting "single-package" clusters (functional components) are largely independent from the start of the segmentation; even with a shift of 50% with respect to the segment size. This seems to suggest that the statistical properties of the class occurrences and the class correlation are stable throughout the trace file. Although we expected a good stability of the technique with respect to small changes (<10%) in the position of the segments in the trace, we have been surprised to see that even a large shift would still produce comparable results. This finding is very encouraging. A second and important result we can bring out is that our clustering technique is efficient even with very large execution traces. Finally, it is worth mentioning that our approach has been successfully applied to the reverse engineering of industrial software [27].

As further work, we will extend our statistical processing of the class occurrence in the trace. In fact the data analysis today is very limited: it amounts to observing the presence or absence of a class in each segment. Therefore, a class occurrence of 1000 or 1 in a given segment is considered the same. We will then expand our technique to compute class occurrences statistics within the segments. We expect this to bring us new insights to the architecture of the legacy system and that even more precise functional component identification could be performed.

7. RELATED WORK

In the literature, many techniques have been proposed to recover the structure of a system by splitting it into components. They range from document indexing techniques [11], slicing [12] to the more recent "concept analysis" technique [13] or even mixed techniques [14][14]. All these techniques are static i.e. they try to partition the set of source code statements and program elements into subsets that will hopefully help to rebuild the architecture of the system. But the key problem is to choose the relevant set of criteria (or similarity metrics) [16] with which the "natural" boundaries of components can be found. In the reverse-engineering literature, the similarity metrics range from the interconnection strength of Rigi [17] to the sophisticated information-theory based measurement of Andritsos and Tzerpos [18][19], the information retrieval technique such as Latent Semantic Indexing [11] or the kind of variables accessed in formal concept analysis [13][20]. Then, based on such a similarity metric, an algorithm decides what element should be part of the same cluster [21]. In their work, Xiao and Tzerpos compared several clustering algorithms based on dynamic dependencies. In particular they focused on the clustering based on the global frequency of calls between classes [23]. This approach does not discriminate between situations where the calls happen in different locations in the trace. This is to be contrasted with our approach that analyzes where the calls happen in the trace. Very few authors have worked on sampling or segmentation techniques for trace analysis. One pioneering work is the one of Chan et al. [24] to visualize long sequence of low-level Java execution traces in the AVID system (including memory event and call stack events). But their approach is quite different from ours. It selectively picks information from the source (the call stack for example) to limit the quantity of information to process.

The problem to process very large execution traces is now beginning to be dealt with in the literature. For example, Zaidman and Demeyer proposed to manage the volume of the trace by searching for common global frequency patterns [22]. In fact, they analyzed consecutive samples of the trace to identify recurring patterns of events having the same global frequencies. In other words they search locally for events with similar global frequency. It is then quite different from our approach that analyzes class distribution throughout the trace. Another technique is to restrict the set of classes to "trace" like in the work of Meyer and Wendehals [6]. In fact, their trace generator takes as input a list of classes, interfaces and methods that have to be monitored during the execution of the program under analysis. Similarly, the tool developed by Vasconcelos, Cepêda and Werner [8] allows the selection of the packages and classes to be monitored for trace collection. In this work, the trace is sliced by use-case scenarios and message depth level and it is then possible to study the trace per slice and depth level. Another technique developed by Hamou-Lhadj [7] uses text summarization algorithms, which takes an execution trace as input and returns a summary of its main contents as output. Sartipi and Safyallah [9] use a patterns search and discovery tool to separate, in the trace, the patterns that correspond to common features from the ones that correspond to specific features. Although the literature is abundant in clustering and architecture recovery techniques, we have had a hard time finding any research work whose results would actually be benchmarked against some reference architecture, but the notable exception of Mitchell [21] who uses static techniques. Our approach seems original also to this respect.

8. ACKNOWLEDGEMENTS

This work has been done with the support of HESSO Grant N° 15989 from the Swiss Confederation, which is gratefully acknowledged. The authors would also like to thank the computing center (CTI) of the State of Geneva for their support.

9. REFERENCES

[1] Bergey J., Smith D., Weiderman N., Woods S. 1999. Options Analysis for Reengineering (OAR): Issues and Conceptual Approach. Software Engineering Institute, Tech. Note CMU/SEI-99-TN-014, 1999.

[2] Kazman R., O'Brien L., Verhoef C. 2003. Architecture Reconstruction Guidelines, 3rd edition. Software Engineering Institute, Tech. Report CMU/SEI-2002-TR-034, 2003.

[3] Tilley S.R., Santanu P., Smith D.B. 1996. Toward a Framework for Program Understanding. Proc. IEEE Int. Workshop on Program Comprehension, 1996.

[4] Biggerstaff T. J., Mitbander B.G., Webster D.E. 1994. Program Understanding and the Concept Assignment Problem. Communicaitons of the ACM, CACM 37(5), 1994.

[5] Gély, A. 2005. Algorithmique Combinatoire: Cliques, Bicycles et Systèmes Implicatifs. PhD thesis. Univ. de Clermont-Ferrand II, 2005.

[6] Meyer M., Wendehals L. 2005. Selective Tracing for Dynamic Analyses. Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05).

[7] Hamou-Lhadj A. 2005. The Concept of Trace Summarization. Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05).

[8] Vasconcelos A., Cepêda R., Werner C. 2005. An Approach to Program Comprehension through Reverse Engineering of Complementary Software Views. Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05).

[9] Sartipi K., Safyallah H. 2006. An Environment for Pattern based Dynamic Analysis of Software Systems. Proceedings of the 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06).

[10] Dugerdil Ph., Jossi S. 2008. Empirical Assessment of Execution Trace Segmentation in Reverse-Engineering. ICSOFT 2008.

[11] Marcus A. 2004. Semantic Driven Program Analysis. Proc IEEE Int. Conference on Software Maintenance (ICSM'04).

[12] Verbaere M. 2003. Program Slicing for Refactoring. MS Thesis, Oxford University. 2003

[13] Siff M., Reps T. 1999. Identifying Modules via Concept Analysis. IEEE Trans. On Software Engineering 25(6). 1999

[14] Harman M., Gold N., Hierons R., Binkeley D. 2002. Code Extraction Algorithms which Unify Slicing and Concept Assignment. Proc IEEE Working Conference on Reverse Engineering (WCRE'02).

[15] Tonella P. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. IEEE Trans. On Software Engineering. 29(6), 2003

[16] Wiggerts T.A. 1997. Using Clustering Algorithms in Legacy Systems Remodularization. Proc IEEE Working Conference on Reverse Engineering (WCRE '97),

[17] Müller H.A., Orgun M.A., Tilley S., Uhl J.S. 1993. A Reverse Engineering Approach To Subsystem Structure Identification. Software Maintenance: Research and Practice 5(4), John Wiley & Sons. 1993

[18] Andritsos P., Tzerpos V. 2003. Software Clustering based on Information Loss Minimization. Proc. IEEE Working Conference on Reverse engineering. 2003

[19] Andritsos P., Tzerpos V. 2005. Information Theoretic Software Clustering. IEEE Trans. on Software Engineering 31(2). 2005

[20] Tonella P. 2001. Concept Analysis for Module Restructuring. IEEE Trans. On Software Engineering, 27(4), 2001

[21] Mitchell B.S. 2003. A Heuristic Search Approach to Solving the Software Clustering Problem. Proc IEEE Conf on Software Maintenance. 2003

[22] Zaidman A., Demeyer S. 2004. Managing trace data volume through a heuristical clustering process based on event execution frequency. Proc. of the IEEE European Conference on Software Maintenance and Reengineering (CSMR'2004).

[23] Xiao C., Tzerpos, V. 2005. Software Clustering basd on Dynamic Dependencies. Proc. of the IEEE European Conference on Software Maintenance and Reengineering (CSMR'2005).

[24] Chan A., Holmes R., Murphy G.C., Ying A.T.T. 2003. Scaling an Object-oriented System Execution Visualizer through Sampling. Proc. of the 11th IEEE International Workshop on Program Comprehension (ICPC'03).

[25] Hamou-Lhadj A., Lethbridge T.C 2002. Compression Techniques to Simplify the Analysis of Large Execution Traces. Proc. of the IEEE Workshop on Program Comprehension (IWPC), 2002.

[26] Dugerdil Ph. 2007. Using trace sampling techniques to identify dynamic clusters of classes. Proc. of the IBM CAS Software and Systems Engineering Symposium (CASCON), 2007

[27] Dugerdil Ph., Jossi S. 2007. Role based clustering of software modules: an industrial experiment. Proc. ICSOFT 2007.

[28] Jacobson I., Booch G., Rumbaugh J. 1999. The Unified Software Development Process. Addison-Wesley Professional 1999.

[29] Hamou-Lhadj A. Lethbridge T. 2006. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behavior of a Software System. Proc. of the IEEE Int. Conference on Program Comprehension (ICPC'06), 2006.

[30] http://www.graphviz.org/