Software Architecture Reconstruction: An Approach Based on Combining Graph Clustering and Partitioning

Ioana Şora, Gabriel Glodean, Mihai Gligor
Department of Computers
Politehnica University of Timisoara, Romania
Email:ioana.sora@cs.upt.ro

***Abstract*—This article proposes an approach of improving the accuracy of automatic software architecture reconstruction. Many research uses clustering for the purpose of architectural reconstruction. Our work improves the results of coupling/cohesion driven clustering by combining it with a partitioning preprocessing that establishes a layering of the classes of the system. Two simple and not really efficient algorithms for software clustering are improved by applying this approach, as it is shown in the validation section.**

***Keywords*-software architecture; partitioning; clustering ;**

I. Introduction

Software architecture is a model of the software system expressed at a high level of abstraction. The architectural view of a system raises the level of abstraction, hiding details of implementation, algorithms, and data representation and concentrating on the interaction of "black box" elements. As defined in [1],

> "The software architecture of a program is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Knowing and having an explicit representation of the system architecture is crucial in order to maintain, understand and evaluate a large software application. Architecture is not explicitly represented in the code, but it has to be documented [2], [1]. Often, the available documentation is incomplete, outdated or is completely missing, the designers having only the code available, sometimes only in compiled form. Reconstructing the architectural model from the available code remains the saving alternative in these cases.

Architecture reconstruction is an interactive and iterative process involving many activities. Architectural constructs are not represented explicitly in the source code, but the same architectural constructs can be realized by many diverse mechanisms in an implementation. When a system is initially developed, its high-level design/architectural elements are mapped to implementation elements. Therefore, when we reconstruct those elements, we need to apply the inverses of the mappings.

The reverse engineering community has developed a lot of techniques to help capture the structure of existing software systems, as they are surveyed and classified in [3]. Many reconstruction techniques, although supported by tools, have a reduced degree of automation. Our goal is to develop a quasi-automatic reconstruction technique, while obtaining a reconstructed architectural model of a good quality - similar to the one extracted by a human expert.

This article is organized as follows: Section II presents how we extract facts from code and build the initial model of the software system. Section III introduces our reconstruction approach that combines clustering and partitioning. Experimental results demonstrating the advantages of this approach are described in Section IV.

II. The dependency model

Our approach starts with constructing a lightweight model of the software system. We defined our model by having in mind following requirements:

- to operate only with information that can be extracted from compiled code (where no source code is available)
- to be a general object oriented model, not bound to a particular language

The model is a reification of the static dependencies that exist between the systems parts. A part A depends on a part B if there are explicit references in A to elements of B.

The elements of this dependency model are classes and the relationships between them. We emprically defined 11 different dependency types characterized by weights. The values of the weights have been empirically finetuned. Actually the absolute values of the weights are not important in our approach, but their relative ratios.

The dependency types taken into account in our approach and the weights are:

- Inheritance: A inherits B; weight=150
- Implements interface: A implements B; weight=100
- Member: A has at least a member of type B; weight = 75. The number of B's methods called on these members is taken into account to proportionally increase this weight.
- Method parameter: A has at least a method with a parameter of type B; weight= 50. The number of B's methods called on these members is taken into account to proportionally increase this weight.

- Local variable: A has a local variable of type B in a method; weight=50. The number of B's methods called on these members is taken into account to proportionally increase this weight.
- Returned type: A has a method that returns type B; weight=35.
- Field access: A accesses directly at least a field of B; weight=150.
- Static method call: A calls a static method of B; weight=50. The number of B's static methods called is taken into account to proportionally increase this weight.
- Object instantiation: inside A an object of type B is instantiated; weight 30.
- Class parameter; weight=15.
- Type cast : inside A a cast to B is done; weight=50.

Two classes A and B can be in the same time in several dependency relationships (for example, A can have members of type B and a can have a method with parameters of type B). The strenght of the dependency A depends on B is given by the sum of all dependency types occuring between them.

Our first version extracts this dependency model from Java bytecode that is processed with ObjectWeb ASM [4], while we are currently working on integrating also a dependency model extractor from compiled .NET (MSIL) code. This is possible since the dependency model is general and operates with general object-oriented concepts.

III. The reconstruction approach

Our reconstruction approach combines Partitioning and Clustering. This section describes the basic Partitioning and Clustering approaches implemented and how they are combined.

A. Partitioning

The term Partitioning [5] was defined in the context of product development processes and it means to resequence the design tasks in order to maximize the availability of information required at each stage of the process. In some situations, the information flow is as such that not all information can be made available when required - it is the case of circular paths of information flow.

The operation of partitioning can be defined also in the context of the structure of software systems. A well designed software system is one in which modules can be partitioned into layers, with each module having dependencies only on modules within the layer or belonging to the layer below. Partitioning finds layers and highlights dependency cycles.

A number of tools are available for extracting dependencies from code and analysing the layering of dependencies, such as Lattix [6] and [7]. These tools support only the manual architectural reconstruction, as they do not attempt to automatically extract subsystems and their relationships.

We have implemented a partitioning algorithm like [6], that orders all classes of the model in layers, determined by the directions of the dependencies.

B. Clustering

Quasi-automatic reconstruction techniques aim at finding the natural cluster structure of software systems, with as little user intervention as possible. These techniques divide a given system into subsystems that relate to each other from a logical design point of view. Software clustering refers to the decomposition of a software system into meaningful subsystems. To be meaningful, the automatic approach must produce clusterings that can help developers understand the system.

Clustering algorithms [8] have been largely used in data mining to identify groups of objects whose members are similar in some way. In reverse softare engineering, clustering is used to produce architectural views of applications, by grouping together in subsystems modules (classes, functions, etc) that relate to each other. The basic assumption driving this softwatre clustering approach is that software systems are organized into subsystems characterised by internal cohesion and loosely coupling with each other. There are several researches using different clustering algorithms for software decomposition [9], [10], [11], [12], [13] [14], [15], [16].

In our work, we have choosen to start from a class of simple graph theoretical clustering algorithms, based on minimum spanning trees [17], [18] and study the impact of combining these in an original way with partitioning, in order to improve the accuracy of the automatic reconstruction.

1) The ZMST clustering algorithm: The first clustering algorithm from this class was defined by Zahn [18], who demonstrated a set of theorems that proof this clustering method. This clustering algorithm was first used for software clustering by [14]. The ZMST clustering algorithm starts by finding the minimum spanning tree of a graph, and in a second phase proceeds to remove from the tree the edges that are too long (a treshold value has to be provided). The clusters are formed by the nodes that are still connected by edges from the minimum spanning tree. When applying ZMST for software clustering, in our approach, the initial weighted graph results from the dependency structure matrix. Since the dependency structure matrix describes a directed graph, we first transform it into an undirected graph by assigning to every edge the weight resulting as the average between the two directed dependencies. The treshold value used for edge ellimination can be set as a parameter of the algorithm - in our approach we specify the value in a percentual manner (an edge is considered too long if its weight is smaller than the certain percent of the average weight of all edges in the graph).

2) The MMST clustering algorithm: Another minimum spanning tree clustering algorithm is proposed in [14], named the Modified MST (MMST) Clustering. MMST also starts by finding the minimum spanning tree, but in the second phase it iteratively constructs the clusters: it starts having each node forming its own cluster, and in succesive iterations it unites clusters that are too close (a treshold value has to be provided). Finally, the nodes that are left isolated are assigned to the clusters they are the closest to (in a process called orphan adoption).

3) Additional preprocessing - library class ellimination: A first step performed before any clustering algorithm is the identification and ellimination of library classes [14]. These are classes used by many others, and if they are not elliminated from the clustering process they tend to group many classes in a single large cluster around them. We use a treshold value (classes used by more than a certain percent of the total number of classes) to elliminte them.

C. Clustering combined with partitioning

1) Motivation: As observed by many researchers, clustering software based on a similarity/dissimilarity metric derived from coupling/cohesion does not provide satisfactory results [19]. As a consequence, other categories of informations are taken into account [3]. Certain approaches like [15] consider even non-software informations, such as: symbolic textual information available in the comments or in the file, class or method names, historical data (time of last modification, author) held by version control system repositories, the physical organization of applications in terms of files and folders. Other researches use other architectural inputs such as identified design patterns, as a form of architectural clue to be used in the clustering process [14].

2) Original approach: In this work, we propose a new approach of enhancing the basic cohesion/coupling-based similarity/dissimilarity metric used for software clustering: besides the strenght of the dependencies between two classes A and B, we take into account also the abstraction layers of A and B. Abstraction layers are determined with a partitioning algorithm as described in subsection III-A.

We make the observation that two classes that are situated in layers of different abstraction levels are highly unlikely to be part of the same architectural subsystem, even if there is a strong dependency (given by many method calls or inheritance) between them. On the other hand, two classes that are situated on the same or on close layers have a higher chance to be part of the same architectural subsystem.

Starting from this observation, we have implemented our approach by defining a similarity metric between classes A and B that is composed by the dependency strength between A and B plus a distance adjustement proportional with the difference between the abstraction layers of A and B.

3) Implementation: We have studied different ways of applying the distance adjustement.

We define δ as the absolute value of the difference between the layers of A and B, divided to the total number of layers in order to normalize the value in the interval [0,1].

$$\delta(A,B) = \frac{|Layer(A) - Layer(B)|}{TotalLayers}$$

The similarity metric is given by the dependency strenght pondered with the distance adjustement.

$$sim(A,B) = dep(A,B) \cdot adjustement(\delta(A,B))$$

The distance adjustement is a decreasing function

$$adjustement : [0,1] \rightarrow [0,1]$$

We experimented with following adjustement functions: none, linear, exponential. The adjustement functions in these three cases are:

$$none(\delta) = 1, \forall \delta \in [0,1]$$

$$linear(\delta) = 1 - \delta, \forall \delta \in [0,1]$$

$$exponential(\delta) = e^{-4 \cdot \delta}, \forall \delta \in [0,1]$$

When applying any of the adjustement functions, classes that are mutually dependent and are situated on the same layer have $\delta = 0$, and the value of the linear or exponential adjustement function is 1, thus the similarity is given only by the dependency strenghth. For any other case, the bigger the distance is, the smaller will be the value of the adjustement function, reducing accordingly the dependency strength.

We have studied different possibilities of applying the distance adjustement functions:

- for MST clustering: adjustement is applied to the dependency structure matrix before starting any clustering algorithm. Following versions of MST clustering with adjustement result: *none*, *linear*, *exponential*.
- for MST clustering: adjustement is applied only in the second phase of the clustering algorithm: the minimum spanning tree is built without taking into account adjustements, and distance adjustements are taken into account only in the second phase of edge ellimination. There result following MST clusterings: *linear-after*, *expo-after*.
- for MMST clustering: adjustament can be applied separately for the phase of the cluster determination and the phase of orphan adoption. There result following MMST clusterings, by all possible combinations of different adjustements in the two phases of MMST: *none-none*, *none-linear*, *none-expo*, *linear-none*, *linear-linear*, *linear-expo*, *expo-none*, *expo-linear* and *expo-expo*.

IV. Validation

A. Approach used for validation

A good automatically produced clustering should approximate the clustering produced by a human expert - the architect. In order to validate our clustering technique, we compare how close the automatic obtained solution is to a reference solution. The reference solution is given by human software engineering experts who manually evaluate the systems.

The comparison of two clustering solutions is done automatically with help of the MoJo metric defined by Tzerpos and Holt [20], [21], [22]. This metric measures the distance between two clusterings of the same system. The MoJo metrics counts the minimum number of operations one needs to perform in order to transform one clustering solution to the other. The MoJo metric is based on two operations, both of which are consedered of the same importance: moving an element from one cluster to another and the other operation is that of joining two clusters. The direct MoJo metric is thus a dissimilarity measure, the bigger the value means the

two clusterings are less similar. In order to use a similarity measure, we use another quality measurement based on MoJo, introduced also by Tzerpos and Holt, which normalizes MoJo with respect to the number of elements in the system. Given two clusterings, A and B, of a system with N elements, the MoJo similarity measurement is defined as:

$$similarityMoJo(A,B) = [1 - \frac{MoJo(A,B)}{N}] \times 100\%$$

We have implemented an algorithm to automatically calculate the MoJo similarity metric and apply it to check the compliance between the clusterings produced automatically by our tool and a reference clustering solution given by the software architect expert.

Figures 1 and 2 compare the results of clustering an example system (the kernel of our clustering tool) using the MST and MMST algorithms with different distance adjustements vs the same algorithms without any adjustement.

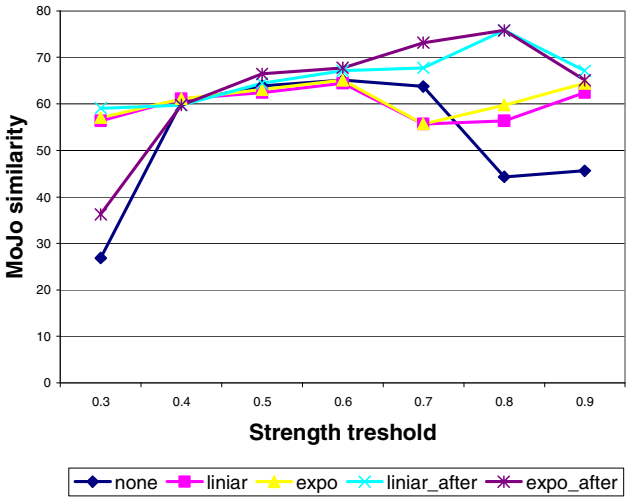

Fig. 1. MOJO similarity results for clustering of ARTkernel: MST algorithm, with different types of adjustements

Figure 1 compares the result produced by MST with no adjustement, and MST with different types of distance adjustements (linear, exponential, linear–after, exponential–after).

Figure 2 compares the result produced by MMST with no adjustement, and MMST with different types of distance adjustements (linear-none, exponential-none, none-linear, linear-linear, exponential–linear).

First, both algorithms (MST and MMST) required a tuning process in order to establish the values for the dependency strenghts assigned for different dependency types. The values used are these presented in section II.

Also, the results produced by both algorithms depend on the treshold value ST taken into account for edge ellimination. If the value for ST is too small, too many edges are elliminated and the algorithm produces many small clusters. If the value for ST is too big, the algorithm produces a single large cluster. For the algorithms with no adjustament, the optimal value for ST is around 0.6, as we can see in Figures 1 and 2 the MoJo

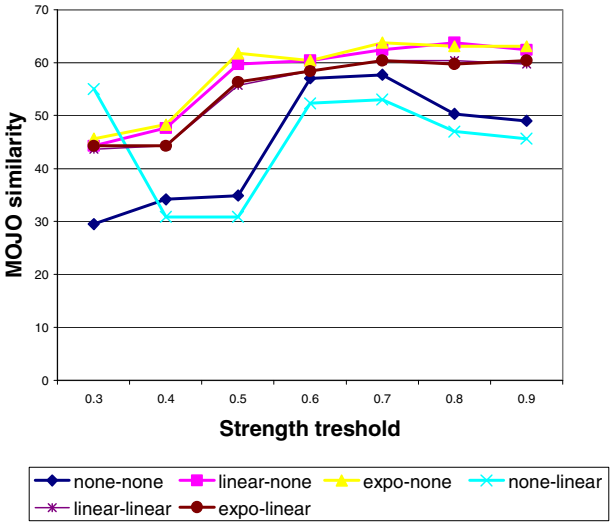

Fig. 2. MOJO similarity results for clusytering of ARTkernel: MMST algorithm with adoption, with different types of adjustements

similarity with the refereence solution has a maximal value around this value for ST.

B. Impact of distance adjustement on MST clustering

Analysing the impact of introducing distance adjustements in the MST algorithm, as we can see in Figure 1, all the 4 adjustement types have a positive influence on the MoJo similarity - the maximum MoJo similarity values of any of these curves are higher than the maximum MoJo simularity value obtained in case of no adjustement. Another general positive effect of applying any distance adjustement in the case of MST clustering is that the curves are flattened, that means that finding the optimal value for ST is not that important any more.

If we compare the effectiveness of the 4 adjustement types, we can notice that exponential adjustements give better MoJo similarity than their linear counterparts. In the case of the example analysed in figure 1, the differences between results obtained with linear vs exponential adjustements are not so big - this is due to the fact that the system has a relatively small (120) number of classes and thus the layer distances are limited.

What can be noticed is that the best result of increasing clustering quality (highest MoJo values) are always obtained for the after-type adjustements, both *linear–after* and *expo-after*, compared with their before-type adjustement counterparts *linear* and *expo*. Thus it results that for the MST clustering algorithm, best results are obtained when taking into account the distance adjustement only in the edge removal phase, but not in the initial phase of MST construction.

C. Impact of distance adjustement on MMST clustering

Analysing the impact of introducing distance adjustements in the MMST algorithm with adoption, as we can see in Figure 2, all the 4 of the 5 adjustement types have a positive influence

TABLE I
SYSTEMS USED FOR CLUSTERING EXPERIMENTS

System	Nb classes	Nb layers	Depend density	Avg layer distance	Avg maximum distance
art kernel	140	125	2.06 %	20 (16%)	28 (22.7%)
junit-4.8.1	230	162	1.39 %	32 (20 %)	51 (31 %)
xercesImpl	649	478	0.65 %	105 (22 %)	144 (29 %)
deploy	531	233	0.54 %	18 (8 %)	34 (14 %)
jEdit	1132	381	0.4 %	37 (9.8 %)	77 (24 %)
Ant	870	514	0.5 %	111 (21 %)	147 (28 %)

TABLE II
EXPERIMENTAL RESULTS - INFLUENCE OF ADJUSTEMENT ON THE CLUSTERING RESULTS OBTAINED WITH THE MST ALGORITHM

System	MOJO - none	MOJO - expo after	diference
art kernel	65.1	75.84	10.73
junit-4.8.1	48.26	56.08	7.82
xercesImpl	36.67	54.69	18.02
deploy	47.08	57.81	10.73
jEdit	28.44	46.73	11.95
Ant	28.62	40.57	11.95

TABLE III
EXPERIMENTAL RESULTS - INFLUENCE OF ADJUSTEMENT ON THE CLUSTERING RESULTS OBTAINED WITH THE MMST ALGORITHM

System	MOJO - none-none	MOJO - expo-expo	diference 1	MOJO - expo-none	diference 2
art kernel	57.5	60.4	2.89	63.4	5.89
junit-4.8.1	54.2	66.95	12.17	65.2	10.43
xercesImpl	49.6	50.5	0.9	62.55	12.94
deploy	67.22	73.63	6.4	74.19	6.96
jEdit	56.07	68.02	11.95	67.4	11.33
Ant	55.28	63.44	8.16	62.98	7.7

on the MoJo similarity. It can be observed that taking into account the distance adjustement only in the clustering phase but not in the orphan adoption phase gives for the analysed system slightly better results (*expo-none* is better than *expo-liniar*, *liniar-none* is better than *liniar-liniar*).

D. Quantitative evaluation of the improvement brought by the layer distance adjustement

We repeated the experiments on several different systems, in order to compare the improvement obtained with exponential adjustement vs no adjustement.

We have applied the evaluation procedure on several case studies, and we present here a summary for following systems: the kernel of our own prototype tool, junit-4.8.1, xercesImpl, deploy, jEdit, Ant. The characteristics of these test cases are presented in Table I. Each system is characterized by a set of parameters representing:

- the size, given by the number N of classes composing the system;
- the number of distinct layers resulted after partitioning. The difference between the number of layers and number of classes is given by the classes with cyclic dependencies that are placed on the same layer.
- the density of the dependency matrix - the number of existing dependency relationships divided to the number of all possible dependency relationships between N classes
- the average layer distance of all dependencies, in absolute value and as a percent of the total number of layers
- the average of the maximum distances where a class presents dependencies, in absolute value and as a percent of the total number of layers

Table II contains the results for the MST clustering algorithm, while Table III contains the results for the MMST clustering algorithm. The tables present the absolute maximum values of the MoJo similarity metric, obtained with and without distance adjustement on a set of systems.

In table III we analysed also the differences between the *expo-expo* and *expo-none* versions, diference 1 presents the diference between *expo-expo* and *none-none*, while diference 2 presents the difference between *expo-none* and *none-none*.

As the tables show, including a distance adjustement in both clustering algorithms always produces decompositions that are closer to the reference solution.

The values of the differences between the MoJo similarities for clustering with and without distance adjustemens vary. As expected, the biggest difference (biggest improvement) is noticed for systems that do not present a strict layering (that

TABLE IV
EXPERIMENTAL RESULTS - EXECUTION TIMES (SEC)

System	Size (classes)	Read input	Create system model	Partitioning	MST Clustering	MMST Clustering
ARTkernel	120	0.028	0.002	0.116	0.125	0.118
ASM	159	0.261	0.006	0.118	0.137	0.237
Lobo	235	0.138	0.003	0.116	0.134	0.145
inCode	663	0.417	0.026	0.126	0.15	0.226
jython	1021	0.446	0.017	0.206	0.219	0.495
clojure	1215	0.494	0.03	0.202	0.326	0.625
classes.jar	20242	6.42	0.616	5.228	31.3	120

have many dependencies spanning big layer distances). It is the case of the systems xercesImpl, jEdit, Ant and junit, that have the biggest layer distances between classes in dependency relationships, and that give the biggest improvements in the quality of the reconstructed architecture after applying the distance adjustement.

We can conclude that the distance adjustement is always benefic for the quality of the clustering result, and the exponential adjustement works best, and from a quantitative point of view improvements are biggest for systems with many classes that that have many dependencies spanning big layer distances.

E. Scalability and performance of reconstruction tool

In order to validate the scalability and performance of our automatic reconstruction tool, we used it for systems of different dimensions, starting from a small classroom application of 20 classes and up to the java runtime classes.jar having over 16000 classes. The execution times in seconds for each of the processing phases are shown in table IV.

V. CONCLUSION AND FUTURE WORK

Our approach for automatic software architecture reconstruction combines traditional clustering approaches with partitioning. In this way, we develop an automatic reconstruction technique able to produce a reconstructed architectural model of a good quality - closer to the one extracted by a human expert. As a future work we intend to apply distance adjustements issued from partitioning to more other software clustering methods in order to prove the generality of the proposed approach.

REFERENCES

[1] L. Bass, *Software Architecture in Practice*, 2nd ed. Addison Wesley, 2002.

[2] *ISO/IEC Standard for Systems and Software Engineering - Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std. ISO/IEC 42 010 IEEE Std 1471-2000, 2007.

[3] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Campan, and H. Verjus, "Towards a process-oriented software architecture reconstruction taxonomy," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 137–148, 2007.

[4] ObjectWeb consortium: the ASM homepage. [Online]. Available: http://asm.ow2.org/

[5] D. Gebala and S. Eppinger, "Methods for analyzing design procedures," 1991.

[6] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 167–176.

[7] NDepend tool. [Online]. Available: http://www.ndepend.com/

[8] S. E. Schaeffer, "Survey: Graph clustering," *Computer Science Review*, vol. 1, 2007.

[9] F. B. e Abreu, G. Pereira, and P. Sousa, "A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems," in *Proceedings of the Conference on Software Maintenance and Reengineering,CSMR*, 2000, pp. 13–22.

[10] T. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Proceedings of the Fourth Working Conference on Reverse Eengineering*, 1997, pp. 33–43.

[11] Y. Chiricota, F. Jourdan, and G. Melancon, "Software components capture using graph clustering," in *Proceedings IWPC*, 2003, pp. 217–226.

[12] A. Christl, R. Koschke, and M.-A. D. Storey, "Automated clustering to support the reflexion method," *Information & Software Technology*, vol. 49, no. 3, pp. 255–274, 2007.

[13] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 193–208, 2006.

[14] M. Bauer and M. Trifu, "Architecture-aware adaptive clustering of OO systems," *Software Maintenance and Reengineering, European Conference on*, vol. 0, p. 3, 2004.

[15] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Trans. Software Eng.*, vol. 31, no. 2, pp. 150–165, 2005.

[16] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of component-based architecture from object-oriented systems," in *Software Architecture, Working IEEE/IFIP Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 285–288.

[17] O. Grygorash, Y. Zhou, and Z. Jorgensen, "Minimum spanning tree based clustering algorithms," in *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, 2006.

[18] C. Zahn, "Graph-theoretical methods for detecting and describing gestalt clusters," *IEEE Transactions on Computers*, vol. C, no. 20, pp. 68–86, 1971.

[19] F. B. e Abreu and M. Goulão, "Coupling and cohesion as modularization drivers: Are we being over-persuaded?" in *Proceedings of the Fifth Conference on Software Maintenance and Reengineering, CSMR*, 2001, pp. 47–57.

[20] V. Tzerpos and R. C. Holt, "Mojo: A distance metric for software clustering," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999, p. 187193.

[21] Z. Wen and V. Tzerpos, "Evaluating similarity measures for software decompositions," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2004, pp. 368–377.

[22] B. S. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *Proceedings of ICSM*, 2001.