Using Spectral Clustering to Automate Identification and Optimization of Component Structures

Constanze Deiters, Andreas Rausch, Mirco Schindler
Department of Informatics - Software Systems Engineering
Clausthal University of Technology
38678 Clausthal-Zellerfeld, Germany
{constanze.deiters, andreas.rausch, mirco.schindler}@tu-clausthal.de

Abstract—A well-structured, modular software architecture is known to support comprehensibility, maintainability and extensibility of a software system. To achieve this goal the software system is divided into components in such a way that its component structure is optimized regarding cohesion and coupling. But with increasing size and complexity identifying and evaluating a component structure can be rarely accomplished by humans manually.

To support this task, we developed an approach using Spectral Clustering from the field of neural computation. Based on the different dependencies between software elements, our approach automatically forms a component structure of the analyzed software system. In a case study we demonstrate this approach on a software system of manually manageable size and complexity. The results are compared to the component structure skilled software architects manually formed. In most cases both variants, manually as well as automated, provide similar component structures. For this reason, the presented approach seems to be suitable for systems which are not manageable by hand.

Index Terms—Software Architecture, Software Design, Architecture Erosion, Component Structure, Spectral Clustering

I. INTRODUCTION

Over time, the environment of a software system changes continuously. To cope with accordingly changing requirements, the software system itself has to be evolved. Essential for the adaptability of a software system is its architecture. Since the architecture defines the fundamental organization of a software system by providing a structure of components and their interrelationships [1], it significantly influences quality attributes like comprehensibility, maintainability and extensibility. Crucial to gain these attributes is modularity [2]. Various criteria can be used to consider the modularity of a component structure: A high cohesion between the elements within a component and a low coupling between components give evidence about the relation between components [3], [4].

To achieve modularity over the lifetime of a software system, it is not sufficient to initially design a well-structured architecture. Through adaptations the actual and the desired architecture can diverge. As a result to such a divergence, called architecture erosion, not only the quality decreases but also the ability for further adaptations can be lost [5]. If an architecture is eroded, a restructuring is necessary to obtain again a well-structured system. To avoid architecture erosion during evolution, strategies are required which identify how and where adaptations shall take place or which initiate a restructuring.

Even experienced software architects are not able to handle the architecture of complex software systems by hand. The identification of possible component structures and their evaluation against determined criteria have to be supported by tools. Furthermore, the responsible architect of a software system is changing during the system’s life cycle.

A software component usually implements a definite functionality. To realize the functionality the software elements the component consists of shall be strongly interconnected. The extend of interconnection between elements can be interpreted as a similarity measure. Hence, forming components can be simplified to grouping similar elements. Well-suited approaches to solve this issue, are clustering methods which are already widely used in data analysis [6]. Since software systems and their components have an hierarchical structure, the desired approach has to support hierarchical structuring, too.

In this paper, we present an approach which automatically derives a hierarchical component structure from existing code or design models and evaluates it regarding modularity. Based on the different relations between software elements, clusters of software elements are determined using Spectral Clustering and are mapped to components of the architecture.

The remaining paper is structured as follows: The different steps of our approach along with the Spectral Clustering algorithm [7] are explained in Section II. Section III presents and discuses the results of applying the approach on the CoCoME [8] system. An overview over the fields this problem and our approach are related to is given in Section IV. Finally, Section V concludes this paper and gives an outlook on further work.

II. APPROACH

Our approach for automated identification and optimization of component structures mainly based on a standard Spectral Clustering algorithm. After an overview of the overall approach this section describes the clustering algorithm itself and the different steps of our approach in detail.

A. Overview

As shown in Figure 1, the approach is divided into three main steps: the analysis and identification of relations (associ-
In the following the used normalized Spectral Clustering algorithm is described in summary. Inputs are an adjacency matrix $A$ and the number of cluster $k$.

1. Determine the unnormalized Laplacian $L$.
2. Calculate the normalized Laplacian $L_{NOM}$.
3. Determine the first $k$ eigenvectors $e^1, \ldots, e^k$ of $L_{NOM}$.
4. Determine the vectors $x^1, \ldots, x^n \in \mathbb{R}^k$ with $n = |V|$ and $x_j^k = e_j^k$.
5. Cluster the vectors $x_i$ with Neural Gas into $k$ clusters $C_1, \ldots, C_k$.

Starting point is the set of vertices $V = v_1, v_2, \ldots, v_n$ representing the software elements and an adjacency matrix $A^{n \times n}$ representing the pairwise similarities between the elements. This information can be easily considered as a similarity graph. A few restrictions have to be made on the following adjacency matrix:

$$A^G_{n \times n} = \begin{bmatrix}
    w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\
    w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\
    \vdots & \vdots & \ddots & \vdots \\
    w_{n,1} & w_{n,2} & \cdots & w_{n,n}
\end{bmatrix}$$

Spectral Clustering requires that the matrix is symmetric ($w_{i,j} = w_{j,i}$). Hence, the similarity graph corresponding to the considered software system is reduced to an undirected one; where in reality a directed relation determines that element $a$ depends on element $b$ the undirected case only determines that a relation between both elements exists but not its direction. Furthermore, all weights have to be non-negative ($w_{i,j} \geq 0$). If $w_{i,j} = 0$, this means that there exists no relation between node $v_i$ and $v_j$. For the used algorithm the value of $w_{i,i}$ is irrelevant, thus all diagonal elements are set to 0.

Spectral Clustering is based on graph Laplacian matrices [12]. The unnormalized graph Laplacian matrix is defined as

$$L = D - A. \quad (1)$$

$D$ is the degree matrix with the degrees $d_1, d_2, \ldots, d_n$ on the


<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implements A</td>
<td>implements B</td>
</tr>
<tr>
<td>Extends A</td>
<td>inherits B</td>
</tr>
<tr>
<td>A calls a method on B</td>
<td>A uses a method-parameter of type B</td>
</tr>
<tr>
<td>HasMethodParameter</td>
<td>A uses a return value of type B</td>
</tr>
<tr>
<td>HasConstructorParameter</td>
<td>A uses a constructor-parameter of type B</td>
</tr>
<tr>
<td>Declare</td>
<td>B is declared as member variable in A</td>
</tr>
<tr>
<td>Instantiate</td>
<td>B is instantiated in A</td>
</tr>
</tbody>
</table>

Table I: Relation Types

\[
d_i = \sum_j w_{i,j}. 
\]

(2)

The normalized Laplacian is calculated based on \( L \) and \( D \):

\[
L_{NOM} = D^{-1}L. 
\]

(3)

From the multiplicity of the eigenvalues of \( L_{NOM} \) with value equal to 0 the number of connected components of the graph can be derived [13]. If the graph has only one connected component, it is assumed that some eigenvalues are close to zero. An analysis of the pairwise differences of these neighboring eigenvalues gives an indication into how many parts the graph can be separated. This approximation is called Eigengap Heuristic [7].

C. Identifying and Weighting Relations

The focus of this approach is set to component-based and object-orientated systems. Fundamental is the question concerning the important relations between software elements that influence the component structure. By studying different example systems, we identified nine different types of relations, which are listed in Table I. Although these relations are directed, we only use the information whether two software elements are related by a relation or not. This is necessary because Spectral Clustering can only handle an undirected graph. Also the multiplicity of a relation, i.e., how often a relation of the same type exists between two software elements, and the kind of software element (e.g., interface or class) are considered. All of this data is aggregated and transformed into a weight.

The weighting also depends on the component model of the system, which defines the characteristics and a set of quality features for a well-designed component. In a distinct component model like CORBA [14], some relation types can be found within and also between components whereas other types only exist within a component.

To represent the characteristics of different component models, global weights and a component model weighting function are utilized. Global weights \( \tau_n \) are assigned to every association type \( \tau_n \). The component model weighting function \( \Phi_{i,j}(M, \tau) \) of a component model \( M \) is defined as the set of context weighting functions \( \varphi_{M,\tau_n}(i,j) \):

\[
\Phi_{i,j}(M, \tau) = \{ \varphi_{M,\tau_1}(i,j), \ldots, \varphi_{M,\tau_n}(i,j) \}. 
\]

(4)

The context-weighting function \( \varphi_{M,\tau_n}(i,j) \) depends on the type of elements \( i \) and \( j \) (e.g., interface or class). Analyses have shown that the type of the involved elements can be relevant to decide whether a relation exists between or within a component. Thus, a context weighting function defines a weight in respect to the relation type and the type of the involved elements.

The weight of all relations of type \( \tau_n \) between two elements \( i \) and \( j \) is calculated by the weighting function

\[
f_{\tau_n}(i,j) = \begin{cases} 
0, & \text{if a relation of type } \tau_n \text{ does not exist} \\
\mu_{i,j}(\tau_n) \cdot (\lambda \cdot \gamma_{\tau_n} + \xi \cdot \varphi_{M,\tau_n}(i,j)), & \text{else} 
\end{cases} 
\]

(5)

Its value is either 0 if there does not exist a relation \( \tau_n \) between \( i \) and \( j \) or it is the product of the number of the relations of this type \( \mu_{i,j}(\tau_n) \) and a summed up weight value. This value is calculated by the global weight \( \gamma_{\tau_n} \), the context weighting function \( \varphi_{M,\tau_n}(i,j) \) and the parameters \( \lambda \) and \( \xi \) \( \lambda, \xi \in [0,1] \) are used to adjust the influence of \( \gamma_{\tau_n} \) and \( \varphi_{M,\tau_n}(i,j) \) on \( f_{\tau_n}(i,j) \). They depend on experience and can be changed to possibly achieve a different clustering result.

The total weight between two elements \( i \) and \( j \) is the sum over the weights of all relation types between \( i \) and \( j \). Finally, these values are divided by the number of all relations between \( i \) and \( j \) to gain the normalized values \( w_{i,j}^* \):

\[
w_{i,j}^* = \frac{\sum_{n=1}^{\tau} b_n f_{\tau_n}(i,j)}{\sum_{m=1}^{\tau} b_m \mu_{i,j}(\tau_m)}. 
\]

(6)

The binary variables \( b_n \) and \( b_m \) regulate whether an association of a distinct type \( \tau_n \) or \( \tau_m \) is considered or not. In a graph, whose nodes represent the elements of a software system, the values \( w_{i,j}^* \) are the edge weights. Finally, the matrix consisting of \( w_{i,j}^* \) is the adjacency matrix which serves as input for the Spectral Clustering algorithm.

D. Formation of Components

The application of Spectral Clustering results in a cluster separation of the graph created in Subsection II-C. Each cluster represents a software component. It includes all software elements to which the nodes within the relative cluster correspond.

A main characteristic for component-oriented design is composition of components. Hierarchical composition of components results in a hierarchical structure of the software system. In fact, it is also possible to use the described approach in a hierarchical way. Each cluster can be clustered again with this approach. This is illustrated in Figure 2. First, the initial graph is clustered into the three clusters \( C_1, C_2 \) and \( C_3 \). In the next iteration the clusters \( C_2 \) and \( C_3 \) are clustered again. It is also possible to vary the component model in each decomposition. To fully automate this approach, a termination condition is needed to get meaningful components with more than one element. Approved conditions are, for example, a specified number of elements per component, a fixed number of the composition level or different evaluation metrics.
Fig. 2. Hierarchical Spectral Clustering, cluster $C_2$ and $C_3$ are divided into subclusters

III. CASE STUDY

To evaluate the underlying concepts of the previous presented approach, we applied the implemented prototype to the small and well documented software system CoCoME [8]. The system analysis and its results are discussed in this section and compared to the component structure of CoCoME.

A. CoCoME - the Example System

CoCoME is located in the application domain of retailing as it can be observed in each supermarket. Customers pay their purchase at a cash desk where each article is identified with a bar code scanner and, thereupon, prices are displayed. In the back office, a store manager measures the stock, orders new products, or draws up the daily accounting.

Following the functional aspects, the system component TradingSystem is divided into two connected parts: an embedded system and an information system represented by the components CashDeskLine and Inventory, respectively. Central point of the embedded system CashDeskLine is a cash desk computer, which manages the communication between different hardware parts, like bar code scanner, cash box, or receipt printer. Also each hardware part within the embedded system is represented by a component. They communicate among each other and with the cash desk computer via events. According to the event channel pattern [15], they are all connected to an event channel component (see Fig. 3a), where each component can act as publisher as well as subscriber.

Each cash desk computer is connected to a store server to receive prices, to store sale transactions, or to perform credit card payment. As part of the information system Inventory, the store server is accessed by client applications supporting different store and company managers. The store manager is thus able to change prices for its whole store with only one mouse click or to generate reports regarding its store like an overview over the stock easily.

Following a common reference standard for information systems, a Three-Layer-Architecture [16], Inventory is decomposed into three hierarchically arranged components each playing the role of another layer: Gui, Application, and Data (see Fig. 3b). The component Gui provides functionality to interact with users of the system, for example, store managers, and to forward their requests to the component Application. Application contains the application logic of the system and has access to Data to read, update, create or delete data. Each layer component is furthermore decomposed into subcomponents managing different subtasks. Additionally, the layer components Application and Data contain each classes of a data model and data types for data exchange (transfer objects). These data types are also available at the interfaces between the containing and the upper next layer component.

B. Analysis of the Example System

Many component-orientated approaches to software development pursue a loose coupling and a high cohesion [4], [17], [18]. The normalized Spectral Clustering algorithm is a heuristic which optimizes these goals. To quantify the results of Spectral Clustering, a metric for coupling ($E_{Comp}$) and a metric for cohesion ($E_{Coh}$) are used. Both consider the number of relations $x_i$: $E_{Comp}$ from Cluster $C_i$ to other clusters and $E_{Coh}$ inside of $C_i$.

$$E_{Comp}(i) = 1 - \frac{x_i}{m_i n_i} \in [0, 1], \text{ with } n_i = |C_i|, \ m_i = |C \setminus C_i|$$

(7)

$$E_{Coh}(i) = \frac{2x_i}{n_i(n_i - 1)} \in [0, 1], \text{ with } n_i = |C_i|$$

(8)

We combined both metrics as

$$E_{Comp}(i) = E_{Comp}(i) + E_{Coh}(i) \in [0, 2].$$

(9)

A high value of $E_{Comp}$ indicates a good component structure. In the following, we apply the approach on the component Application and on the whole system. The aim of the application is to identify a functional separation using the described approach.

1) Component Application: As depicted in Figure 3c, the Application consists of three subcomponents encapsulating three different functional aspects. It contains 29 software elements, 24 of them are classes and five are interfaces, and 220 different associations between these elements.

As described in the previous section, an undirected graph representing this system part was derived. Additionally, the global weights, the component weighting function and parameters were determined and the edge weights were calculated. To compare the results more easily with the given structure, the number of clusters was set to three.

The clustering resulted in three clusters, where one cluster contains only one element. This is due to the fact that this element has no associations to any other element within Application. Excluding this separated element, the clustering resulted in three clusters, which have nearly the same structure as the application layer of the CoCoME system. Differences are only in the mapping of transfer objects to components. However, this is not crucial because these objects are only used for data exchange among components. Herold et. al. [4] tackle this issue by assigning transfer objects to that
TABLE II
METER VALUES FOR THE SUBCOMPONENTS OF APPLICATION

<table>
<thead>
<tr>
<th></th>
<th>CoCoMe</th>
<th>Spectral Clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporting</td>
<td>E_Coup</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>E_Coh</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>E_Comp</td>
<td>1.60</td>
</tr>
<tr>
<td>Store</td>
<td>E_Coup</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>E_Coh</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>E_Comp</td>
<td>1.21</td>
</tr>
<tr>
<td>ProductDispatcher</td>
<td>E_Coup</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>E_Coh</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>E_Comp</td>
<td>1.51</td>
</tr>
</tbody>
</table>

The values of the metrics evaluated on these three components are listed in Table II. With our approach, the coupling could be reduced and simultaneously the cohesion of two components could be increased but to the disadvantage of the third component.

2) Whole System: When applying the approach to the whole system all software elements with no relations to other elements were excluded; 93 classes and 21 interfaces remained. Between these 114 elements 1074 different relations were identified. Global weights and all parameters were chosen identically as above.

To determine the number of clusters, the eigengap heuristic is used. Figure 4 visualizes the eigenvalues of the graph Laplacian matrix and their pairwise differences. The relative large gap between the second and third eigenvalue indicates two clusters. One cluster of the result corresponds to the component CashDeskLine while the other corresponds to the component Inventory. Both clusters are structurally identical to the CoCoME components.

Since the whole system should be clustered hierarchical, each of both clusters was examined in another clustering step. Clustering the set of elements contained in CashDeskLine and Inventory resulted in nine and fifteen subclusters, respectively.

The eigenvalues and pairwise differences of the cluster Inventory are plotted in Figure 5, where the gap between the 15th and 16th eigenvalues can be seen clearly.

Compared with the CoCoME structure each of the nine generated subcluster of cluster CashDeskLine can be mapped to one of the nine subcomponents of CashDeskLine. In
contrast, the fifteen subclusters of Inventory cannot be mapped one-to-one to the subcomponents of Inventory. Since the information system represented by Inventory has a classical Three-Layer-Architecture, it could be expected that the clustering results in three clusters which are divided into subclusters in a next iteration. The reason for this is that the approach identifies functional relationships and the analyzed system has no strong dependencies among components within the same layer. For example, all components of the application layer realize different functionalities with low interaction to other components.

Except for the assignment of transfer objects or data types, the generated clusters are structurally identical to the CoCoME components designed by a software architect. We can also note that no data type of the presentation layer and no transfer object of the application layer were assigned to the data layer.

IV. RELATED WORK

Forming the component structure, a software architect can make use of guidelines documented by various researchers and practitioners. Taylor et al. [19] discuss possibilities to cut components and connectors to reach different goals like complexity, scalability, adaptability or dependability. Similar general support is given by Siedersleben in [20], who distinguishes between component categories like application logic or technical. Such considerations are fundamental for a well-cut initial component structure but lack a more precise and measurable criterion which will also enable automation of component identification.

Criteria to evaluate the quality of a software architecture are required. This can be done from different viewpoints: On the one hand, approaches like ATAM [21] have to be mentioned, which primarily examine whether required non-functional properties are met. Since such methods have only a qualitative character, they are not well-suited to provide a basis for a constructive approach. On the other hand, structural properties like modularity are relevant quality criteria. To measure structural properties, many metrics exist or can be adapted to software architecture. For example, a metric similar to the lack of cohesion in methods (LCOM) [3] evaluates the relationships between the different elements contained in a component. In addition to cohesion, the degree of coupling between components can be considered as criteria of modularity in software architecture.

Approaches which group elements using similarity measures are suitable to establish a component structure. In this way, Herold et. al. [4] assign service methods to interfaces based on the similarity of the input and output parameter types. Instead of the similarity between elements, our approach considers dependencies like inheritance between software elements. We also include the number and coupling impact of the different dependencies into the calculations. The approach of Abdeen et. al. [22] consider only systems with existing component structure. They did not touch the structure itself but move software elements like classes from one to another component to optimize the dependencies between components.

Also in the field of architecture reconstruction and reverse engineering several approaches exist which are using clustering algorithms or machine learning techniques. Most of them are graph-based [23]. The various approaches mainly differ in the kind of similarity or dependency represented by the edges and their weighting of the derived graph. Chiricota et. al. [24] determine edges with low density. Density values they calculate based on the number of links within the neighborhood of the nodes linked by the edge. Also Mitchell et. al. [25] did not include the kind of linking and the kind of participating software elements. Bauer and Trifu [26], on the other hand, consider not only different types of dependencies like our approach but also a so called enrichment of semantic information. They detect architectural patterns and calculate the weighting influenced by these results. But all these information can be taken into account by our and all other approaches which are using weighted edges.

Similar to our approach cohesion and coupling is often used as optimization criteria. The clustering algorithm used in [26] based on a modified two-pass minimal spanning-tree algorithm. Other approaches are executing methods from the field of machine-learning like hill-climbing [27], genetic algorithms [28] or simulated annealing [22]. A challenge within these techniques is to guarantee the reproducibility of clustering results which Spectral Clustering does. We also choose Spectral Clustering because of the aim of optimizing cohesion and coupling. This goal fits the NCut (normalized cut) problem [29]. It was proved, that NCut is NP-hard [29], but Spectral Clustering can be regarded as a heuristic for NCut [7]. Certainly, there is no guarantee of an optimal approximation quality.

Another aspect is using these clustering algorithms in a hierarchical way. It is possible to construct a dividing clustering with all techniques. But for an automated approach a separation criterion, like the number of clusters or concurrent the number of created components, has to be determined. In [30], for example, the separation criterion has to be set manually. To solve this issue, we used the eigengap heuristic, which calculates the number of components for each hierarchical level based on the similarities represented by the edge weights.

V. CONCLUSION

We have proposed an approach for automated identification of component structures within a software system to support an architect with designing, maintaining, extending or refactoring. The case study has shown that the approach generates a modular software architecture, which is nearly identical to manually designed components. Through the introduction of the component model function the approach is open for extensions and adaptations to other questions in this domain. With the possibility of using this approach in a hierarchical way and a heuristic to determine the number of clusters, the approach is able to create a modular architecture automatically. Thereby, it is possible to get an overview of large software systems without knowing any detail of it.
One point to discuss, is the issue that the approach does not detect components resulting from the usage of architectural patterns. As observed in the case study, components representing single layers of a Three-Layer-Architecture are not identified. This is due to the fact that in the sample system the elements within a layer are not strongly connected. It is uncertain if the selected types of association are sufficient for identifying such clusters. However, this might be possible by introducing further types of associations or properties depending on the system representation or meta-information. Maybe, methods from other fields like reverse engineering or text mining could be integrated. Deriving information from the directions of the associations combining with bipartite graph matching would be another approach.

The Spectral Clustering algorithm used in this approach deals only with undirected graphs. Since it is recommended that the coupling between components should be a directed and acyclic graph [17], the direction of associations should be taken into account in an improved version of this approach. Certainly, the aim of the approach is to support a software architect - no refactoring or restructuring is performed automatically. In this case, cycles are not obstructive, but the architect should be advised to it.

To investigate the impact of the former discussed open issues, further evaluations of different systems have to be done. Also, since the sample system was small and manageable, the systems to evaluate shall differ in size and complexity.

As future work, we plan to establish a learning procedure for the weights. To setup the weights of a component model function, it is essential to identify meaningful clusters. As observed in the case study, the clustering is performed and checked if the generated component structure matches the training set. For adapting the weights a Hebbian Learning is conducted [31]. The learning of weights leads to aspects like an actual-theoretical comparison between design and implementation of a component-model. Also user-specific weights are imaginable.

ACKNOWLEDGEMENTS

We would like to thank Barbara Hammer for her support regarding neural computation and being advisor of the diploma thesis [31] led to the ideas presented in this paper.

REFERENCES