

Component-based Architecture Recovery from Object Oriented Systems via Relational Concept Analysis

Alae-Eddine El Hamdouni¹, A.Djamel Seriai¹, and Marianne Huchard¹

LIRMM/CNRS, Montpellier 2 University, France
LIRMM - UMR 5506 - CC 477, 161 rue Ada, 34095 Montpellier Cedex 5 - France
{huchard, seriai, elhamdouni}@lirmm.fr

Abstract. Software architecture modelling and representation has become an important phase of the development process of complex software systems. Using software architecture representation as a high level view provides many advantages during all phases of the software life cycle. Nevertheless, for many systems, such architecture representation is not available. To deal with this problem, we propose in this paper an approach of architecture recovery which aims to extract component-based architecture from an object-oriented (OO) system, by a semi-automatic exploration process. To this end, we use relational concept analysis in order to identify the architectural components. The *RCA*-based approach comes as a complementary method to relieve some limits of the existing implementation of *ROMANTIC* based on simulated annealing algorithm. In the *RCA* approach, architectural components are identified from concepts derived by exploiting all existing dependency relations between classes of the OO system. We evaluated the feasibility of our approach on a Java software.

Keywords: Software Architecture, Architecture Recovery, Relational Concept Analysis, Component-based, Object-oriented

1 Introduction

Given the explosive growth of the computer systems size and complexity, software architectures are emerging as a valuable ally for both the design and maintenance of these systems. During the last decade, this abstract view of systems has become a central field of software engineering [4]. Its main advantage is to make easier the program comprehension by allowing us to focus on architectural elements rather than implementation details [10]. Indeed, a component-based software architecture is a high level abstraction of a system using the architectural elements: components which describe functional computing, connectors which describe interactions and configuration which represents the topology of connections between components. In addition to program comprehension, this distinction between functionalities and interaction is crucial to safely maintain the system [14]. However, most existing systems do not have a reliable architecture representation. Indeed, these systems could have been designed without an architecture design phase, as it is the case for most legacy systems. In other systems, the available representation can diverge from the system implementation due to the lack of synchronization between software documentation and implementation. Taking into account the

previous considerations, we have proposed an approach called *ROMANTIC*¹, which focuses on recovering a component-based architecture from OO systems [6]. The main advantage of this approach is in its automation level, which decreases the need of human expertise which is expensive and it is not always available. In fact, starting from the source code, the *ROMANTIC* process aims at selecting among all the architectures which can be abstracted from a system, the best one according to an architectural quality model. Then we formulate this model as measurable constraints. We have experimented with a simulated annealing algorithm [7]. However, in spite of the many advantages that offers this implementation of *ROMANTIC*, it has some limits. First, the architecture obtained by simulated annealing algorithm is an architecture with only one abstraction level, which optimizes the objective-function. Second, composition relationships are not presented: In fact, the algorithm does not provide the execution trace that gives the composition hierarchy of the obtained components. Finally, based on the objective-function evaluation, the result of a simulated annealing algorithm is unique. Therefore, there are no alternative solutions to the final one, because intermediate solutions are considered as less good.

Thereby, we have explored Relational Concept Analysis (*RCA*) as a satisfactory solution to these issues. Thus, we have used *RCA* firstly to overcome the limitations of the search-based algorithm. On the other hand, we aim to combine the results of *RCA* and simulated annealing algorithms to deduce a single relevant architecture. Our objective in this paper is to present elements required for modelling the recovery of component-based architectures from OO systems as a *RCA* problem. In order to use *RCA*, the source code has first to be analysed in order to extract entities to be encoded into contexts. These contexts contain information on the relations linking the entities. Afterwards, the *RCA* algorithm is applied and builds a lattice containing a set of concepts. Finally, some of these concepts are decoded toward architectural components.

The remainder of this paper is structured as follows. Section 2 presents an overview of our approach core. The *ROMANTIC-RCA* approach is presented and illustrated with an example in section 3. Section 4 discusses related work. Conclusion and future work are given in section 5.

2 Background: *ROMANTIC* overview

ROMANTIC aims to recover a component-based software architecture from an OO system using a search-based approach [6,7], i.e. an exploration process of the solution space in order to identify the best solution according to a given fitness function. In order to model the recovery problem as a search-based one, the *ROMANTIC* approach defines the search space, i.e. a representation of all possible architectures, the fitness function driving the search and a meta-heuristic which is the algorithm of the search-space exploration.

¹ *ROMANTIC: Re-engineering of Object-oriented systems by Architecture extraction and migration to Component-based ones.*

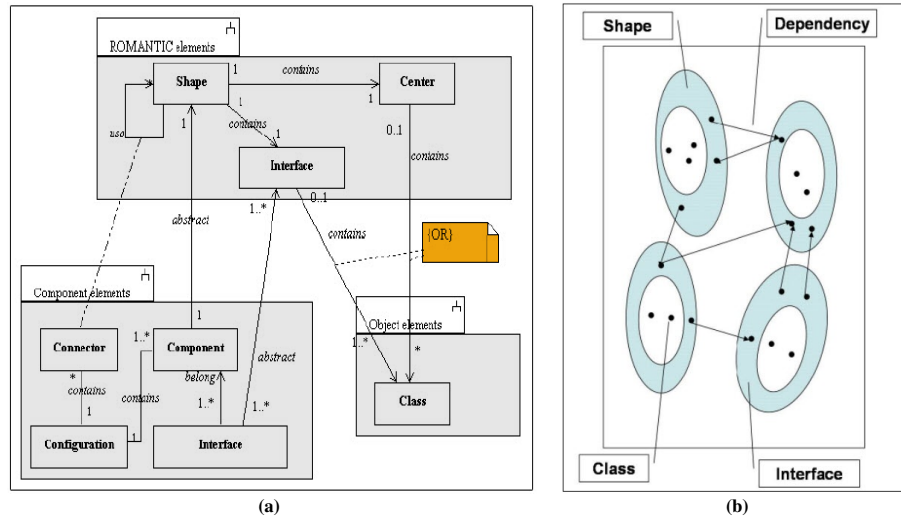


Fig. 1. Object-Component mapping model (a) and *ROMANTIC* elements representation (b)

2.1 Object-component mapping model: Definition of the search space

The search-space of *ROMANTIC* is composed of all architectures, which are partitions of the set of system classes, i.e. it contains all the instances of a mapping model (correspondence) between object concepts (i.e. classes, interfaces, packages, etc.) and architectural ones (i.e. components, connectors, interfaces, etc.).

This mapping model defines an architecture as a partition of the system classes. Each element of this partition represents a component (“Shape”) (cf. Figure 1(a)). Each shape is composed of two sets of classes: the “shape interface”, which contains classes linked with some classes outside the shape, e.g. via method call; and the “center”, which is the remainder of the shape (cf. Figure 1(b)). Concerning the component interface, it is assimilated to “shape interface” as shown on the figure 1(a). Finally, connectors are all dependencies existing between components. As a result of these considerations, in a system which contains n classes, the search-space contains $O(B_n)$ possible architectures, where B_n , or Bell number, is the number of partitions of a set with n members [19].

2.2 Romantic’s quality-model of Architecture : Definition of the fitness function

In addition to the search-space, *ROMANTIC* defines a quality model for software architecture which is used to drive the recovery process. This model is based on the ISO-9126 norm [21] and refines architecture quality characteristics to a set of sub-characteristics. These sub-characteristics are refined in properties on components and then on shapes, like coupling or cohesion. The first quality characteristic is the semantic correctness. This characteristic measures the relevance of a partition of classes according to the concept of architecture. Semantic correctness of architecture is based on semantic-correctness of its architectural elements (component, connector and configuration), and it is different from the semantic of the business logic of these elements. In

the current state of Romantic model, connectors are considered as simple links between components. Thus, semantic-correctness of an architecture is based on semantic-correctness of its components. The semantic correctness of a component is related to its sub-characteristics which are component autonomy, specificity and composability.

These sub-characteristics are refined throughout a refinement model in properties on component, like the component coupling. Finally, these properties are linked to shape properties, like cohesion of the shape classes.

Figure 2 presents the refinement model of the semantic correctness.

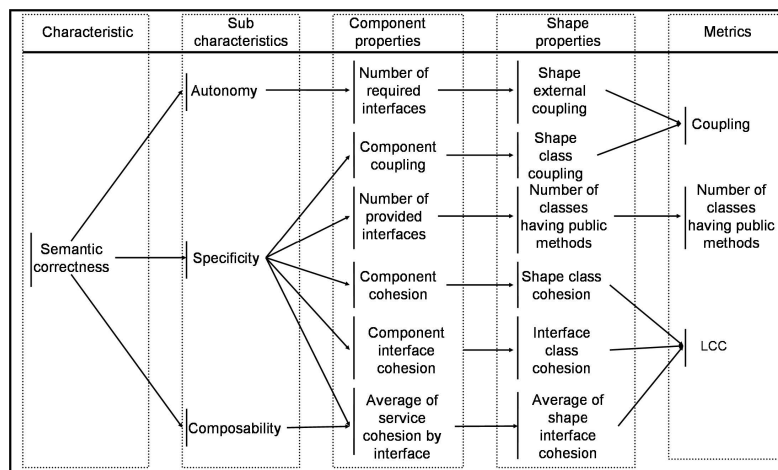


Fig. 2. ROMANTIC's refinement model of a component's semantic properties

The second quality characteristic is the architectural quality. It describes the quality characteristics of the architectural elements, e.g. maintainability and reliability. As result, ROMANTIC's fitness function is based on the quality model and uses a set of metrics to measure the identified shape properties and to evaluate each characteristic. The quality model, the metrics and the fitness function are presented exhaustively in [7].

3 Architecture recovery via Relational Concept Analysis

Throughout this section we use the example of a small pictures collage tool called "PhotoCollage"² to illustrate the different stages of the architecture recovery process using RCA.

PhotoCollage is a small tool to collate pictures over each others on a board. It contains 13 classes: CollageBoard(CB), PhotoItem(PhI), ImageViewer(IV), PhotoItem-Panel(PhIP), HeaderPanel(HP), Java2DHelper(J2Dhlp), GhostGlassPane(GGP), CollageItem(CI), CollageItemTransferHandler(CITH), DragAndDropLock(DnDL), CollageDemo(CD), CollageApplet(CA), ShadowFactory(ShF). Figure 3 presents an overview of the application class diagram.

² Source code available on: <http://www.lirmm.fr/~seriai/PhotoCollage>

metrics definition in the Romantic refinement model (*cf.* Section3.3). Thus, entities and relations have to be extracted by source-code analysis. In order to apply *RCA*, we have defined a process with four steps: The first one focuses on the extraction of a graph of dependencies of source-code classes. The second aims to create an *RCA* model using dependency-graph data. Next, the third step generates a lattice of concepts representing clusters of object classes and the last step aims to identify candidates components from the resulting lattice (Figure 4).

The objective of the Dependency graph (DG) extraction is to generate an accurate representation of the source-code as a graph in which both graph loops and multiple edges are allowed. Graph vertices represent classes of OO source-code, and edges represent different dependency types. The weight of these edges is the value of the strength of the dependency it represents. The Dependency graph is obtained from the extracted dependencies with static analysis of the source-code. The graph is then encoded into RCF (Relational Context Family) (*cf.* Section3.4). The RCF is composed of several contexts describing entities of different categories, and relations between entities. These contexts are processed to obtain a final lattice. The lattice's concepts define the set of all meaningful clusters of system classes. From these concepts, some are selected to represent the candidate architectural components. Here, components are considered as sets of OO classes called *Shape* (*cf.* Section 2). The provided and required interfaces of one of these components are deduced from method calls between classes belonging to different components. Method calls represent the set of connectors between the identified components.

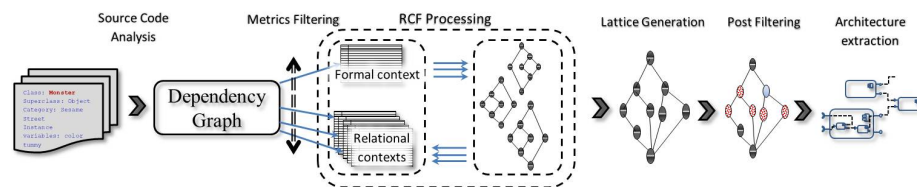


Fig. 4. Process overview

3.3 From *ROMANTIC* metrics to *RCA* relations

As outlined in the previous section, OO classes dependency-links used as relations in *RCA* model must translate the meaning of metrics used in the *ROMANTIC* refinement model. Thus, the study of this refinement model whose constituents have been presented in Figure 2, shows that the characteristics of semantic correctness and architectural quality are refined into metrics like coupling and LCC cohesion.

We analyse these metrics in order to identify the dependency relations that participate in their calculation. In fact, the definition of metrics on OO system elements is obtained by the identification of the different types of relationships between system classes and the computation of their strengths. Thus, we need to determine precisely the dependency relations which must be taken into account and how to measure their strengths. The list of possible links between OO system entities includes inheritance, composition, aggregation, invocation relationships, etc.

Study of coupling metrics to identify and assess dependency relations. Coupling [22] has been defined as "the measure of the strength of association established by a connection of one module to another". Following the considered coupling relations between classes, we distinguish five coupling metrics. The first is the *invocation coupling* (eq.1): two classes are coupled by invocation if at least one method of the former invokes a method of the latter. The strength of the interaction will be given by the number of such invocations, relative to the other invocations made by the class.

$$W_{invokes}(Cl_i, Cl_j) = \frac{|Calls(Cl_i, Cl_j)|}{|Mtds(Cl_i)|} \cdot \frac{|Callers(Cl_i, Cl_j)| + |Callees(Cl_j, Cl_i)|}{|Mtds(Cl_i)| + |Mtds(Cl_j)|} \quad (1)$$

where:

- $Calls(Cl_i, Cl_j)$ is the set of method invocations of the class Cl_j by methods of the class Cl_i ;
- $Callers(Cl_i, Cl_j) \subseteq Mtds(Cl_i)$ is the set of methods of Cl_i that contain invocations to the methods of the class Cl_j ;
- $Callees(Cl_j, Cl_i) \subseteq Mtds(Cl_j)$ is the set of methods of Cl_j invoked by elements of $Callers(Cl_i, Cl_j)$.

The second is the *access coupling* (eq.2) that evaluates the accesses made by a class on attributes of another class.

$$W_{access}(Cl_i, Cl_j) = \frac{|Access(Cl_i, Cl_j)|}{|Mtds(Cl_i)|} \cdot \frac{|Accessors(Cl_i)| + |Accessed(Cl_j)|}{|Mtds(Cl_i)| + |Attrs(Cl_j)|} \quad (2)$$

where:

- $Access(Cl_i, Cl_j)$ is the set of accesses to attributes of class Cl_j by methods of class Cl_i ;
- $Accessors(Cl_i, Cl_j) \subseteq Mtds(Cl_i)$ is the set of methods of class Cl_i that make accesses to attributes of class Cl_j ;
- $Accessed(Cl_j, Cl_i) \subseteq Attrs(Cl_j)$ is the set of attributes of class Cl_j accessed by elements of $Accessors(Cl_i, Cl_j)$.

The third metric is the *type-dependency coupling* (eq.3). For two classes Cl_i and Cl_j it evaluates the use of the class Cl_j as a type by the methods of class Cl_i (parameter or return type) and adds the number of attributes of class Cl_i whose type is class Cl_j .

$$W_{typeDep}(Cl_i, Cl_j) = |Attrs_{Cl_j}(Cl_i)| + \frac{|RtnType_{Cl_j}(Cl_i)| + |Params_{Cl_j}(Cl_i)| + |LocalVars_{Cl_j}(Cl_i)|}{|Mtds(Cl_i)|} \quad (3)$$

where:

- $RtnType_{Cl_j}(Cl_i)$ is the set of methods of class Cl_i whose return type is Cl_j ;
- $Params_{Cl_j}(Cl_i)$ is the set of parameter of class Cl_i whose type is Cl_j ;
- $LocalVars_{Cl_j}(Cl_i)$ is the set of local variables of class Cl_i whose type is Cl_j .

The above mentioned three equations were defined taking into account specially the notion of the density of interaction relations between the system classes. In fact, this way of calculation allows to relativize the coupling measure value compared to the total number of methods of the two classes in interaction.

The fourth coupling metric is the *inheritance coupling* ($W_{extends}$) that evaluates to 1 ($W_{extends}(Cl_i, Cl_j) = 1$) if there exists an inheritance relation between the two classes (Cl_i, Cl_j). And the fifth coupling metric is the *composition coupling* ($W_{compose}$) that evaluates to 1 ($W_{compose}(Cl_i, Cl_j) = 1$) if there exists a composition relation between the two classes (Cl_i, Cl_j).

Study of cohesion metric to identify and assess dependency relations. Cohesion is defined in the literature as the degree of collaboration between different elements of a group. In the OO paradigm, it is based on the method collaboration. The LCC (*Loose Class Cohesion*) metric [5], for example, defines the cohesion as the fraction of the methods that use a same attribute or invoke a same method by the number of method pairs. We will consider that two classes Cl_i et Cl_j are cohesive if their methods are. That is, if there is at least one method pair $(m_{ik}, m_{il}), m_{ik} \in Mtds(Cl_i)$ and $m_{il} \in Mtds(Cl_j)$ that invoke the same method or access the same attribute of another class. This is thus encoded by invocation and access relations (eq.1 and eq.2). The above identified relations are represented in the dependency graph. As example, Figure 5 represents a simplified view of the dependency graph of the studied "PhotoCollage" application.

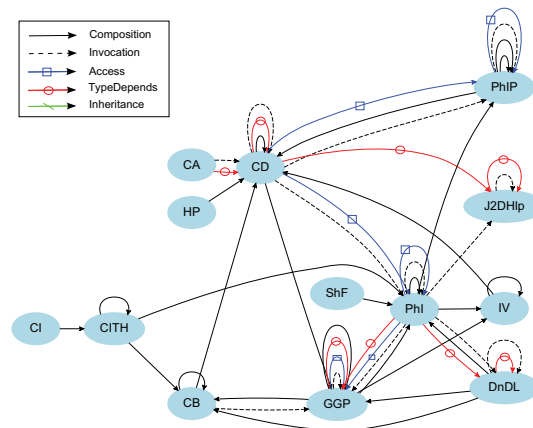


Fig. 5. A simplified dependency-graph of the "PhotoCollage" software.

3.4 Relational Concept Analysis for architecture recovery

RCA analyses data represented through a relational context family (RCF). The RCF is composed of several contexts describing entities of different categories, and relations between entities. Here we consider only one category of entities, which is composed of all the classes of the system under study. They are described by their name

in the unique non-relational context of the RCF. Then, a relational context is added for each type of dependency, so we build five relational contexts: $r_Invokes$, $r_Accesses$, $r_TypeDepends$, $r_Compose$, $r_Extends$ (for inheritance). These relational contexts are built on the basis of the dependency graph. Let d be the considered dependency, d is *invokes*, *access*, *typeDep*, *compose* or *extends*.

In the r_d relational context, a pair (Cl_i, Cl_j) is established if the dependency is greater than a threshold θ , that is $W_d(Cl_i, Cl_j) \geq \theta$. The threshold θ is given by the architect according to its own understanding of the system, or it can be preset to a fixed value, for example, the average of the $W_d(Cl_m, Cl_n)$ values, for all classes Cl_m, Cl_n of the system. The RCA process then takes all these tables and iterates on two steps: (1) building a lattice on the main (non-relational) context concatenated with the five relational contexts, (2) transforming the five relational contexts to integrate the concepts found at that current step (to use this knowledge in the next iteration) [12]. During this transformation, we use an existential scaling operator. The classes that form the columns of a r_d context are replaced by concepts that group the classes. If $(Cl_i, Cl_j) \in r_d$ initially, and Cl_j is in the extent of a concept C at the current step, thus in the current version of r_d we add (Cl_i, C) . The obtained concepts can be interpreted with patterns such as "group of classes that invoke methods of this group of classes and inherit from this other group of classes, etc." The process stops when no new concept emerges during the FCA analysis.

As example, Table 1 shows one of the relational contexts of the system RCF. This RCF is constructed from the dependency graph of the studied "PhotoCollage" application.³

	CollageBoard	PhotoItem	ImageViewer	PhotoItemPanel	HeaderPanel	Java2DHelper	GostGlassPane	CollageItem	CollageItemTransferHandler	DragAndDropLock	CollageDemo	CollageApplet	ShadowFactory
CollageBoard							×						
PhotoItem	×					×	×			×			
ImageViewer													
PhotoItemPanel			×										
HeaderPanel													
Java2DHelper						×							
GostGlassPane							×						
CollageItem													
CollageItemTransferHandler													
DragAndDropLock										×			
CollageDemo		×	×								×		
CollageApplet												×	
ShadowFactory													

Table 1. The relational context $r_Invokes$ of the "PhotoCollage" software.

³ This lattice is built using a framework called eRCA available on <http://code.google.com/p/erca/>

3.5 Identification of architectural components

The concept lattice is explored from the top, by considering sets of concepts forming a candidate architecture, whose extents cover the whole class set.

The choice, done by the expert, of the concepts to be considered as candidate components depends on several factors, including the concepts extent size. A large concept may be considered as a composite component whose sub-components are the sub-concepts. Otherwise, one class can belong to the extent of several concepts of the candidate architecture. In this case, the expert has to evaluate the strength of the relation between the class and the other classes of each extent. The class is discarded when this relation is under a threshold. If there is more than one class shared by many concepts, this set of shared classes belongs necessarily to a common sub-concept of the concerned concepts. Here we choose this sub-concept instead of its parents. The remaining classes belong to the other sub-concepts which are chosen too. The process stops when no sub-concept with more than a class in its extent is shared by more than one parent concept. Each concept in the candidate architecture is then considered as an architectural component. The sub-concepts of each architectural component can be used as an opportunity to identify composite components.

In the *PhotoCollage* example, the top-part, of the generated lattice, groups concepts which are candidate components⁴. From these concepts we were able to identify 3 components. The first contains 6 classes (CB, PhI, PhIP, GGP, DnDL and CD) which are responsible of the collage board management (where pictures are glued). The second contains 5 classes (IV, HP, CI, CITH and ShF) which manage the item that handle a chosen picture. The last component contains the remaining classes (J2Dhlp and CA) which represent the system execution interface. The identified candidate components constitute the architectural components of the system architecture of the "PhotoCollage" application.

4 Related Work

Several works are proposed in literature to extract architecture from an OO system [18]. We distinguish these works according to their automation level. First, some approaches are almost manual. For example, Focus [16] proposes a guideline to a hybrid process which regroups classes and maps the extracted entities to a conceptual architecture obtained from an architectural style according to the human expertise. Second, most approaches propose semi-automatic techniques. They automate repetitive aspects of the recovery process but the reverse engineer steers the iterative refinement or abstraction, leading to the identification of architectural elements. Thus ManSART [11] tries to match source code elements on the architectural styles and patterns defined by reverse engineers. Third, some techniques are quasi-automatic. For example, we can cite the clustering algorithms which are used to produce cohesive clusters that are loosely interconnected [1]. *ROMANTIC* approach is quasi-automatic too. The main difference with other quasi-automatic approaches is that it refines the commonly used definitions of

⁴ Both complete lattice and contexts tables are available on <http://www.lirmm.fr/~seriai/CLA10>

components into characteristics and refinement models, whereas other works use the expertise of the authors in order to define rules driving the process.

Similarly, different approaches have been proposed using formal concept analysis either in software re-engineering or maintenance. For example, in [17] they propose an approach of refactorings and design defects correction on procedural OO systems using *RCA*. In fact, these systems often include Blob or God Class that reveals a procedural design. Thus, correcting a Blob amounts to splitting the Blob class into smaller cohesive sets by grouping class members that collaborate to realize a specific responsibility of the Blob class. Cohesive sets are identified using formal concepts whose intents involve both proper characteristics and inter-member links, such as calls between methods. G. Arévalo [2,3] has developed different *FCA*-based software engineering tools that help to generate high level view at different levels of abstraction. In re-engineering context, many work has used *FCA* to identify modules and components in legacy systems [8,15,20]. For example, V. Deursen [8] uses both *FCA* and clustering algorithm [13] to build OO classes from procedural source code. Elements from source code are gathered according to the features they share. Then, the resulting concepts are candidate classes and sub-concept relationships represent relations between these classes. In a similar way, M. Stiff [20] has presented a method for identifying modules in legacy systems based on concept analysis. A formal context is built from the system elements, and both negative and positive attributes are used in order to extend the context to be well formed. Then, an algorithm of concept partition is used to discover possible partitions in the set of the generated concepts. The chosen partition represents the set of candidate classes.

5 Conclusion and Future Work

We have proposed in this paper an approach to recover component-based architecture from an object-oriented system. We have presented elements required to model this issue with *RCA*. We rely in part on the results obtained in a previous work based on the use of a simulated annealing algorithm. We mainly use a model refining the quality characteristics of components in a set of metrics. Thus, we studied how metrics are calculated in order to identify relationships within classes of the object-oriented system. These relationships are used in an *RCA* model to generate a lattice of concepts. Some of these concepts are considered as components of the resulting architecture. The obtained lattice allows us to: identify architectures with several abstraction levels, identify composite components, select components according to some grouping criteria and navigating in the lattice. In our future work, we want to develop some query patterns in order to select component (concepts) whose classes satisfy some given criteria. Also, we aim to compare architectures obtained by the simulated annealing and *RCA* algorithms respectively. We expect to exploit the combined results of these algorithms to extract a more relevant architecture.

References

1. Anquetil, N., Fourier, C., Lethbridge, T.C.: Experiments with clustering as a software re-modularization method. In: Proc. of the Sixth WCRE. p. 235. IEEE (1999)

2. Arévalo, G., Ducasse, S., Nierstrasz, O.: Lessons learned in applying formal concept analysis to reverse engineering. In: Ganter, B., Godin, R. (eds.) ICFCA. Lecture Notes in Computer Science, vol. 3403, pp. 95–112. Springer (2005)
3. Arévalo, G., Mens, T.: Analysing object-oriented application frameworks using concept analysis. In: Bruel, J.M., Bellahsene, Z. (eds.) OOIS Workshops. Lecture Notes in Computer Science, vol. 2426, pp. 53–63. Springer (2002)
4. Bertolino, A., Bucchiarone, A., Gnesi, S., Muccini, H.: An architecture-centric approach for producing quality systems. In: QoSA/SOQUA. pp. 21–37 (2005)
5. Bieman, J.M., Kang, B.K.: Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes 20(SI), 259–262 (1995)
6. Chardigny, S., Seriai, A., Oussalah, M., Tamzalit, D.: Extraction of component-based architecture from object-oriented systems. In: WICSA. pp. 285–288. IEEE Computer Society (2008)
7. Chardigny, S., Seriai, A., Tamzalit, D., Oussalah, M.: Quality-driven extraction of a component-based architecture from an object-oriented system. In: CSMR. pp. 269–273. IEEE (2008)
8. van Deursen, A., Kuipers, T.: Identifying objects using cluster and concept analysis. In: ICSE. pp. 246–255 (1999)
9. Ganter, B., Wille, R.: Formal Concept Analysis - Mathematical Foundations. Springer (1999)
10. Garlan, D., Perry, D.: Introduction to the special issue on software architecture. IEEE Transactions on Software Engineering 21(4), 269–274 (1995)
11. Harris, D.R., Reubenstein, H.B., Yeh, A.S.: Reverse engineering to the architectural level. In: Proc. of ICSE. pp. 186–195. ACM, Inc. (1995)
12. Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Annals of Mathematics and Artificial Intelligence 49(1-4), 39–76 (2007)
13. Johnson, S.: Hierarchical clustering schemes. Psychometrika 32, 241–245 (1967)
14. Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. Ph.D. thesis, University of Stuttgart (2000)
15. Lundberg, J., Löwe, W.: Architecture recovery by semi-automatic component identification. Electr. Notes Theor. Comput. Sci. 82(5) (2003)
16. Medvidovic, N., Jakobac, V.: Using software evolution to focus architectural recovery. Automated Software Eng. 13(2), 225–256 (2006)
17. Moha, N., Hacene, A.R., Valtchev, P., Guéhéneuc, Y.G.: Refactorings of design defects using relational concept analysis. In: Medina, R., Obiedkov, S.A. (eds.) ICFCA. Lecture Notes in Computer Science, vol. 4933, pp. 289–304. Springer (2008)
18. Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S., Verjus, H.: Towards a process-oriented software architecture reconstruction taxonomy. In: CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. pp. 137–148. IEEE Computer Society, Washington, DC, USA (2007)
19. Rota, G.C.: The number of partitions of a set. The American Mathematical Monthly 71, No 5, 498–504 (1964)
20. Siff, M., Reps, T.W.: Identifying modules via concept analysis. IEEE Trans. Software Eng. 25(6), 749–768 (1999)
21. for Standardization, I.O.: ISO 9126-1 Software Engineering - Product Quality - Part 1: Quality Model. International Organization for Standardization (2001)
22. Stevens, W., Myers, G., Constantine, L.: Structured design pp. 205–232 (1979)