Using Version Information in Architectural Clustering – A Case Study

Andreas Wierda
Océ-Technologies BV,
P.O. Box 101,
NL-5900 MA Venlo,
The Netherlands
awi@oce.nl

Eric Dortmans
Océ-Technologies BV,
P.O. Box 101,
NL-5900 MA Venlo,
The Netherlands
hdo@oce.nl

Lou Somers
Eindhoven Univ. of Technology,
Dept. Math. & Comp.Sc.,
P.O. Box 513, NL-5600 MB
Eindhoven, The Netherlands
wsinlou@win.tue.nl

Abstract

This paper describes a case study that uses clustering to group classes of an existing object-oriented system of significant size into subsystems. The clustering process is based on the structural relations between the classes: associations, generalizations and dependencies. We experiment with different combinations of relationships and different ways to use this information in the clustering process. The results clearly show that dependency relations are vital to achieve good clusterings.

The clustering is performed with a third party tool called Bunch. Compared to other clustering methods the results come relatively close to the result of a manual reconstruction. Performance wise the clustering takes a significant amount of time, but not too much to make it unpractical.

In our case study, we base the clustering on information from multiple versions and compare the result to that obtained when basing the clustering on a single version. We experiment with several combinations of versions. If the clustering is based on relations that were present in both the reconstructed and the first version this leads to a significantly better clustering result compared to that obtained when using only information from the reconstructed version.

1. Introduction

The architecture of a software system represents a blueprint of the system. Having an up to date architecture description is an important prerequisite for software maintenance, which represents a large portion of a software project's total costs. In practice, however, such a description is often not available and source code is the most important information source for reverse engineering [1], [2].

Developers performing maintenance usually start by understanding the problem and the involved parts of the software. The gradual deterioration of the software's internal structure makes this increasingly difficult. Maintenance programmers performing adaptive or perfective maintenance spend half their time studying the program source code and the associated documentation, as already showed in [3]. When performing corrective maintenance this increases even further. This means that a reduction of the effort needed to understand the internal structure of software directly affects the total costs of the project.

Originally, the term legacy software was used for programs written in languages like assembler, Cobol, or Fortran. However, legacy problems are not constrained to specific types of languages. Changing environments and requirements also affect object-oriented software. Several projects where object-oriented legacy systems are reengineered are mentioned in [4]. It turns out that legacy software even exists in relatively young programming languages such as Java. The increasing rate of change causes object-oriented software to become legacy much sooner than non-object-oriented software [5].

In the context of architecture recovery, *pattern detection* and *clustering* are two complementary approaches. The first finds common abstractions embedded in the system, but in practice never covers all entities in the system [6]. The second classifies all entities in the system, but imposes a new ordering instead of some hidden ordering [7], [8], [9], [10], [11].

Other approaches for architecture recovery encompass *manual* architecture reconstruction, the usage of *Conway's law*, and *program slicing*. Manual reconstruction uses navigation and browsing tools to reconstruct an architecture. Conway's law [12], which states that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations", helps to choose suitable abstractions [5], [13]. Program slicing is often used to extract reusable components from an implementation or specification [14], [15].

1.1 Clustering

Clustering is a data analysis technique for dividing data elements into groups of similar elements that are called clusters [16]. This division is based on the similarity of data elements, which are usually represented as points in a multidimensional space or vectors of measurements [17]. Intuitively, in a valid clustering the data elements within a cluster are more similar to each other than to those in other clusters.

Various terms are used to refer to the data elements. Publications that describe the clustering process sec call them *objects* [16], [18] or *patterns* [17]. A unified framework for software subsystem classification techniques where the data elements are called *nodes* is presented in [19]. Approaches that use clustering for reverse engineering often use the terminology of [20], in which the clustered data elements are called *entities*.

Clustering is an unsupervised classification technique: it does not start with a collection of pre-classified entities. Clustering has many applications, including the classification of plants and animals, speech and character recognition, image segmentation, information retrieval and data mining [17].

The clustering algorithms usually start with an abstract graph representing the structure of the program, for example with the nodes representing classes and the edges inter-class relationships. Some similarity measure is then used to find groups of similar or closely related classes, which are grouped into subsystems. This is repeated until an optimal decomposition is found. The end result can be browsed top-down, helping to understand the complete program.

Several clustering-based architecture reconstruction approaches are reported in [20], [21], [22] and [23]. Among the clustering tools we find Arch [24], Rigi [25], which uses the method from [26], and Bunch [23], which implements three different partitional clustering algorithms, an exhaustive algorithm, an hill-climbing algorithm, and a genetic algorithm. Bunch has been used in various reverse engineering case studies [27], [23], [28], [29], [30]. Klocwork InSight [31] is a commercial architecture reconstruction and analysis tool.

1.2 Clustering result evaluation

In general, clusterings are evaluated with an external or internal assessment, or a *relative test* [32], [23]. The latter compares the produced clustering to an *expert decomposition* using some measure. It is considered the ideal assessment method to evaluate the quality of architectural clusterings [22]. However, it has the disadvantage that an expert's decomposition must be available. Some relative test methods are discussed below.

Precision and recall can be used to compare a clustering result to an a priori structure created by experts on the analyzed systems [30]. Though frequently used to evaluate clustering results, precision/recall has several limitations [29]. First of all, the calculation does not consider edges. Second, the measurement is sensitive to the number and size of the clusters. A few misplaced modules in a cluster with relatively few members have much more impact on precision/recall than when the cluster has many members. Finally, number and size of the clusters impact precision/recall.

Two decompositions can also be compared by means of the *MoJo metric* [28]. This is the minimal number of move and join operations required to transform one decomposition into the other. A move operation relocates a single entity from one cluster to another cluster. A join operation merges two clusters. Let K and D be two decompositions of a system of N entities and let $mno(K,D)$ be the minimum number of move and join operations to transform K into D. If $x{\downarrow}y$ denotes the minimum of x and y, then

$$MoJo(K,D) = mno(K,D) \downarrow mno(D,K) \ .$$

If K refers to a decomposition produced by a clustering algorithm and D to an expert decomposition, the quality of K relative to D is defined as

$$MoJoQuality(K,D) = \left(1 - \frac{MoJo(K,D)}{N}\right) \times 100\% \ .$$

An efficient algorithm to compute the MoJo distance between two decompositions is described in [33]. Two case studies where the MoJoQuality metric is used to evaluate the quality of decompositions produced by the ACDC algorithm are described in [34]. They achieve a MoJoQuality of 56% and 64%. This is claimed to be among "the higher ones an automatic clustering algorithm can hope to achieve". This matches with the results reported by [35], obtaining a MoJoQuality of about 50% to 65%.

An extended version, *EdgeMoJo*, also takes the number and weight of edges into account [32]. An improved version of the MoJo metric, *MoJoFM* [36], solves some anomalies of the MoJo metric, such as the tendency of MoJo to consider clusterings with singleton clusters very good. However, MoJoFM does not take the edges into account. A method to compare *hierarchical decompositions* is informally described in [37].

2. Case study

The subject system for our reconstruction case study is a printer controller. Such a controller consists of general-purpose hardware on which proprietary and third party software is running. Its main task is to control physical devices such as a print- and scan-engine, and to act as an intermediate between them and the customer network. The software running on the controller has been written in multiple programming languages, but mostly in C++. An as-designed architecture is available, but it is not complete and parts of the architecture documentation are not consistent with the implementation.

Table 1 shows the characteristics of the first (1) and the last (8a) version of the controller and of two of its subsystems, called Grizzly and RIP Worker.

Table 1. Software characteristics

	Controller		**Grizzly**	**RIP Worker**
	(v. 1)	**(v. 8a)**		
Classes	1545	2661	234	108
Header and source files	4378	7549	268	334
Functions	21711	40449	2037	1857
Lines of source code (*1000)	453	932	35	37
Executable statements (*1000)	167	366	18	16

Although it is known that it is not possible to reconstruct architectures from source code fully automatically [38], it is expected that the architectural views reconstructed this way can serve as good starting points for manual refinement. Based on this we formulate the following hypothesis:

H1: Automatic clustering-based architecture reconstruction methods can reconstruct an architectural view of the controller from its source code that is a good starting-point for manual refinement.

During its lifetime the controller has been modified extensively. The internal structure of software systems that are continuously modified inevitably deteriorates, which obfuscates the architecture. This means that in the original version the architecture is present in a purer form than in later versions. Since architecture reconstruction is usually performed for software of which several versions have been released, it is likely to be applied to software of which the architecture has deteriorated significantly. We speculate that this reduces the effectiveness of clustering-based architecture reconstruction techniques. If this is the case, incorporating information from multiple versions in the clustering process could improve the quality of the result. This leads to the following hypothesis:

H2: Utilizing information obtained from source code of older versions can improve the quality of the output of architectural clustering algorithms for more recent versions of a system.

We have built a workbench that uses clustering techniques to reconstruct a static view of the software architecture from source code. This workbench can incorporate information obtained from the source code of multiple versions of a system into the clustering process. The workbench has been applied to the controller to confirm our two hypotheses.

3. Workbench set-up

Before we sketch the set-up of our architectural-clustering workbench, we first discuss the decisions that led to it, following the items laid out by [17].

3.1 Entity representation and feature selection

Most of the controller is written in an object-oriented programming language (C++), where classes are the most important building blocks. They provide an initial grouping of closely related data and functions [23]. Architectural clustering approaches for object-oriented source code often choose classes as the entities to be clustered. We therefore decided that the set of classes extracted from the source code would form the entity set. Clusters grouping a number of classes will be called subsystems, or simply clusters.

Based on [23], [34], and [35] we decided that the clustering will be based on structural relations between the classes. We distinguish the three most important types of relationships between classes in object-oriented systems:

- *Association*: a structural relationship between two classes that specifies one class is connected to another.
- *Generalization*: the object-oriented mechanism via which more specific classes incorporate the structure and behavior of more general classes.

- *Dependency*: a “using” relationship that specifies a change in one class may affect another class.

If two classes are related by any of these relations, it is possible that multiple instances of this relation exist. The clustering can take the actual number of relation instances into account, or just its presence. We define a Boolean parameter p_c that specifies if only the presence (p_c false), or the number of instances of a relation between two classes must be taken into account (p_c true).

By our definition of the relationship types, an association from class x_1 to class x_2 also implies a dependency from x_1 to x_2. A similar argument holds for generalizations: If x_1 inherits from x_2, x_1 is likely to use methods or attributes of x_2. Therefore generalization usually leads to a dependency from x_1 to x_2. We define a Boolean parameter p_i to indicate whether or not redundant dependencies should be removed: p_i false implies that all dependencies are included.

In various publications it is suggested to use different weights for the different relationship-types, making certain types more important than others [23], [35]. We introduce a parameter for each relationship-type that specifies the weight of instances of this relation in the similarity calculation. This leads to three parameters, p_{w_a}, p_{w_g} and p_{w_d}, respectively specifying the weight of association, generalization and dependency.

3.2 Similarity metric and algorithm choice

We need a clustering algorithm that can cluster an entity set with inter-entity features. Bunch [39] is a tool that implements several clustering algorithms that operate on this kind of data. It has been used in various architectural-clustering experiments and is known to produce clusterings within a bounded approximation of the optimal clustering [27]. Because Bunch has been implemented as a generic clustering tool, it can easily be integrated in an architecture-reconstruction workbench. Based on experiences with Bunch reported by [23], we decided to use the hill-climbing algorithm and the TurboMQ similarity metric.

We use Bunch differently than the applications reported in literature. The difference is threefold:

- We use Bunch to cluster object-oriented software, not procedural code. This affects the entity representation and feature selection, not the clustering algorithm itself.
- We distinguish multiple different relationship types with *different* weights. This is supported by Bunch.
- In some cases we also use information from *multiple* versions. This only affects the number of features, and not the clustering algorithm itself.

3.3 Data abstraction

Bunch automatically generates names for the created clusters. These names are based on an increasing sequence number and the level of the cluster in the decomposition. However, these names have little meaning to software maintainers. Ideally, the workbench would give meaningful names to the clusters. Because we consider the decomposition produced by the architectural clustering as a starting point that needs to be refined manually, using the names Bunch generated is no significant restriction. Therefore we leave the issue of automatically giving meaningful names to the clusters as future work and use the names Bunch generated.

3.4 Assessment of output

Architectural clustering methods usually assess their output by comparing it to some expert decomposition, or by manually checking it. Consensus is that the first method is to be preferred [23], [22], so we choose to implement this method in the workbench.

We choose to use the MoJoQuality metric [28] to compare the generated decompositions with an expert decomposition, because it is a normalized metric for comparing clustering results to expert decompositions for which reference values have been published. Based on clustering results reported in [28], [35], and [36], we consider a decomposition produced by architectural clustering good if it has a MoJoQuality of at least 60% relative to an expert decomposition.

The EdgeMoJo metric [32] is a non-normalized metric that also takes the relations between the classes into account. Incorrect class-placements that affect many relations are considered more important than those that affect a few relations. According to [40] and [32] it is important to take edge-information into account too. Therefore, we use the EdgeMoJo metric to validate hypothesis H2. Because it is not normalized, this metric cannot be used to compare the quality of architectural clusterings of different systems (so for hypothesis H1), but it can be used to determine if changes to the clustering process lead to an improvement of the result (assuming that the same classes are clustered).

Due to the size of the controller both the expert decomposition and the decomposition Bunch produces must be hierarchical. We use the approach of [37] to assess our hierarchical decompositions.

3.5 Combining version information

Hypothesis H2 states that the use of information from older versions of the controller during the clustering process can improve the quality of a decomposition of the last version.

The first question is which *classes* are selected from the models of the system versions. Assume that one of the models represents the version V of which the architecture is reconstructed. A decomposition that contains the *united* sets of classes of V and some older version is likely to contain classes that are no longer present in V. Because this shows unexpected classes to maintainers, we argue that this must be avoided. On the other hand, a decomposition of V that only contains the classes that were *also* present in the other version will not have much value either, because it is likely to leave some of the classes of V unclassified. We therefore decide that the produced decomposition must contain the classes in V and no more.

The second question is how the structural information of the two models (i.e. the *relations*) is combined. From the preceding discussion it is obvious that only the classes present in the last version must be clustered. This means that information from other versions must be incorporated through the relationships. Let C be the set of all classes of all versions, and T the set of relationship-types between the classes. As described earlier, multiple instances of a relationship may be present between two classes. Therefore each relation has a source, target, type and *count*. The count value represents the number of instances of the relation. The set of class-relations R contains each distinct triple of a source, target and type at most once. Hence, if N is the set of natural numbers, $R \subseteq C \times C \times T \times N$. The third component of R is called the *type-component*, and the fourth component the *count-component*. An element of R is called a *class-relation*.

Below we describe two operations to combine sets of class-relations using the definitions of C, T and R given above. They intersect, respectively unite, two sets of class-relations.

Class-relations-intersection, denoted by $\cap_r$, is an operation with type $R \times R \rightarrow R$ that gives the class-relations present in *both* sets of class-relations, ignoring differences in the count component. Informally, the class-relations-intersection of two sets of class-relations R_i and R_j starts by intersecting R_i and R_j with the count-component removed. Next, each tuple of the result is extended with a count-component that is the minimum of the count-components of the corresponding tuples in R_i and R_j.

If $n_i \downarrow n_j$ refers to the minimum of two values n_i and n_j, the class-relations-intersection of two sets of class-relations R_i, R_j is defined as:

$$R_i \cap_r R_j = \left\{ \left(x, y, t, n_i \downarrow n_j\right) \middle| \left(x, y, t, n_i\right) \in R_i \wedge \left(x, y, t, n_j\right) \in R_j \right\} .$$

Class-relations-union, denoted by $\cup_r$, is an operation with type $R \times R \rightarrow R$ that gives the class-relations present in *any* of the two sets. First, the class-relations union of two sets of class-relations R_i and R_j calculates the normal intersection of the two sets without the count-component, and adds a count-component to each tuple that is the *maximum* of the count-components of the corresponding tuples in R_i and R_j. Second, the obtained set is extended with the tuples in R_i for which no corresponding tuple in R_j exists and vice versa.

In order to define the class-relations-union operator more precisely, we need an operator to test the membership of an element in a subset of R without considering the count-component. This class-relations-membership operator $\in_r$ takes an element (x,y,t,n) of R and a subset $R_x \subseteq R$, and gives either true or false. If N is the set of natural numbers, it is defined as:

$$\left(x, y, t, n\right) \in_r R_x \Leftrightarrow \left(\exists m \in N : (x, y, t, m) \in R_x\right) .$$

The class-relations-membership operator gives true if the provided set contains an element that equals the provided element on the first three components. Otherwise, it gives false.

If $n_i \uparrow n_j$ refers to the maximum of two values n_i and n_j, the class-relations-union of two sets of class-relations R_i, R_j is now defined as:

$$R_i \cup_r R_j = \left\{ \left(x, y, t, n_i \uparrow n_j\right) \middle| \left(x, y, t, n_i\right) \in R_i \wedge \left(x, y, t, n_j\right) \in R_j \right\}$$
$$\cup \left\{ r \in R_i \middle| r \in_r R_j \right\} \cup \left\{ r \in R_j \middle| r \in_r R_i \right\}$$

3.6 Workbench architecture

Based on the decisions described above, the architecture of the architectural-clustering workbench has been defined. Figure 1 shows a conceptual view. The boxes indicate processing steps and the black arrows directed data flows.

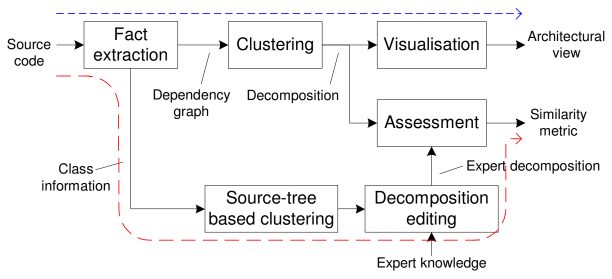

Figure 1. Conceptual view of the workbench

Figure 1 illustrates two typical usage scenarios of the workbench: automatic generation of a decomposition with clustering (fine dotted blue arrow) and assessment of the clustering result (coarse dotted red arrow).

Both scenarios start with the extraction of facts from the source code. The first scenario represents the normal process when using clustering to reconstruct an architecture from source code. In this scenario the extracted dependency graph is clustered and the result is visualized.

The second scenario is used to validate the approach: the clustering result is compared to an expert decomposition. This decomposition is obtained in two steps. First, the classes found during the fact extraction step are organized according to their location in the source tree. Although our reconstruction approach does not need this information, in the case of the controller it is available and not using it would make the manual construction of the expert decomposition much more labor intensive. Second, an editor is used to refine the "draft" decomposition. The resulting expert decomposition is then compared to the clustering result.

Figure 2 shows a process view of the workbench architecture. The rectangles represent processes and the black arrows directed communication channels. The dotted lines represent the data flows of the two scenarios discussed above.

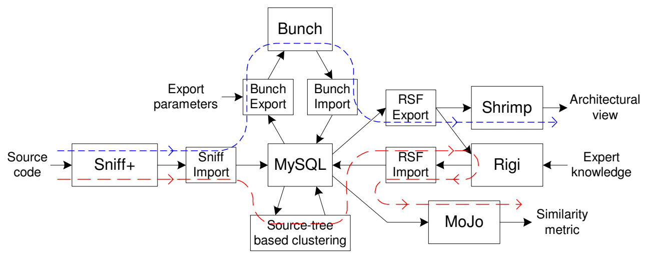

Figure 2. Process view of the workbench

Because Columbus/CAN is not able to extract facts from the complete controller, we use the Sniff+ module to extract the facts from the source code. Due to the large size of the controller this is a computation-intensive step. Therefore we store the results in a MySQL database. The Bunch module implements the clustering process. During the conversion of the facts into a format Bunch accepts, the Bunch Export module takes our user-specified parameters into account. The import and export modules contain "glue-logic" that connects the third party applications to the database.

The Shrimp module allows users to browse a decomposition, but without editing possibilities. The MoJo module compares two decompositions to assess the quality of the clustering result.

The "source-tree based clustering" module creates a decomposition for a single version based on the structure of the source tree. This decomposition is used as the starting point for the expert decomposition. The algorithm is based on the assumption that classes that are defined in source-files in the same directory belong together. The resulting clusters are hierarchically related by their location in the source tree. Based on [41], [34], and [5] we expect this to be a good starting point for refinement by an expert. The expert uses Rigi to refine the decompositions from the "source-tree based clustering" module. After the refinement, the RSF Import module writes the results back to the database.

3.7 Meta-model

The meta-model of the workbench is an abstraction of the source code from which the input for the clustering process is derived. It needs to accommodate the classes and the structural relations.

The meta-model for the workbench has been based on the meta-models of FAMIX [42], MeMoJ [43], Columbus [44], and HisMo [45]. Since the clustering uses information from multiple versions of the controller, the model must accommodate multiple versions. Figure 3 shows an ER model [46]. The rectangles represent entity-sets, diamonds relationships between entity-sets and ellipses attributes. Lines represent one-to-one and one-to-many relationships, the numbers denote the cardinality.

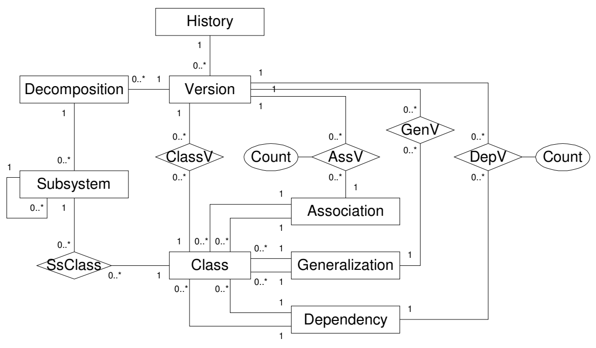

Figure 3. Clustering workbench meta-model

The entities that are clustered (the classes) are the central component. A class is associated with certain versions of the system, indicating its presence in these versions. A set of versions forms a history of a system. Because multiple association and dependency relations can exist between two classes, a "count" value is associated with these relations. A decomposition classifies classes of a certain version of the system. It contains a set of subsystem-trees, modeled by the recursive relation of the subsystem entity. Each subsystem groups a set of classes.

4. Parameter tuning

Before the workbench could be used to reconstruct the architecture of the controller, the proper values of the user-specified parameters had to be determined. We first determined a set of parameter values that produced the best clusterings for two subsystems of the controller. These values were then used to cluster the complete controller.

As explained before, during the transformation of the meta-model into the module dependency graph, a parameter-tuple $(p_{w_a}, p_{w_g}, p_{w_p}, p_c, p_i)$ of our five parameters is used:

- p_{w_a}, p_{w_g} and p_{w_d}: numeric parameters giving the weight of association, generalization, and dependency relations respectively.
- p_c and p_i: Boolean parameters that reduce the amount of information that is written by the Bunch export module. p_c specifies if the instance-count or just the presence of class-relations must be taken into account; p_i specifies whether or not redundant dependencies must be omitted.

Because the controller is relatively large, clustering its module dependency graph is time consuming: on the platform described in Table 9 clustering the most recent version once takes about eighteen minutes (including Bunch Export and Import). The following MoJo calculation takes about five minutes. Therefore, the number of tested parameter-tuples must be limited significantly. Besides this, the numeric parameters (that have no upper bound) make it impossible to test all different parameter-tuples anyway. For each of the numeric parameters the search space is initially set to {0,1,2,3,4,5,6}. Because the four combinations with all numeric parameters equal to zero are not relevant, this gives a total of 1368 combinations to investigate, each requiring ten executions of the clustering algorithm.

Because the clustering algorithm of Bunch is non-deterministic, the average EdgeMoJo value of ten different clusterings is calculated for each parameter-tuple. We refer to the cycle of exporting the module dependency graph (MDG) once and the tenfold execution of the clustering and MoJoQuality and EdgeMoJo calculations as a *ten-clustering cycle*.

For the Grizzly subsystem, the ten-clustering cycle took about four minutes, resulting in a total execution time of 5488 minutes (91 hours) to test the 1372 combinations. For the RIP Worker it was about 3.3 minutes, resulting in a total execution time of 4573 minutes (76 hours).

Table 2 shows the five best and five worst parameter-tuples for Grizzly and the RIP Worker, and the resulting EdgeMoJo and MoJoQuality. Observe that the two sets of best parameter-tuples are disjoint. For Grizzly the EdgeMoJo metric varies between 101.7 for the best and 169.7 for the worst decomposition. The MoJoQuality varies between 69.0% for the best and 57.3% for the worst decomposition. For the RIP Worker these figures are 42.6 and 67.9, and 66.3% and 55.0% respectively. These figures indicate that the choice of the clustering parameters affects the quality of the clustering result significantly.

Table 2. Best five parameter-tuples for Grizzly and the RIP Worker

Grizzly						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	**Edge MoJo**	**MoJo Quality**
2	5	5	false	false	101.7	69.0%
1	3	2	false	false	102.0	69.0%
0	0	6	true	false	102.2	69.8%
2	3	5	false	true	102.3	68.2%
1	4	3	false	false	102.7	68.9%
1358 other measurements						
4	0	0	false	false	169.1	61.2%
1	0	0	true	true	169.4	61.2%
3	0	0	false	true	169.4	61.2%
6	0	0	false	true	169.6	61.2%
0	3	0	false	false	169.7	57.3%

RIP Worker						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
0	5	2	false	false	42.6	66.3%
2	4	3	false	true	42.6	66.4%
1	6	4	true	true	42.7	66.7%
1	5	2	false	true	42.7	67.1%
2	4	1	true	false	42.7	66.0%
1358 other measurements						
1	0	0	true	false	67.4	55.0%
5	0	0	false	true	67.6	54.8%
3	0	0	true	false	67.8	55.1%
3	0	0	true	true	67.8	54.8%
4	0	0	true	false	67.9	55.0%

For the tuples that lead to a *good clustering* it is difficult to distinguish trends. The presence of parameter-tuples with zero for the numeric parameters indicates which types of relationships are important for the clustering result and which are not. When considering the best fifty parameter-tuples for Grizzly, tuples that have $p_{w_a}=0$ also have $p_{w_g}=0$. For the RIP Worker several tuples with $p_{w_a}=0$, but none with $p_{w_g}=0$, are present in the top fifty. So, in both cases no tuples with $p_{w_g}=0$ and $p_{w_a}\neq0$ are present in the top fifty. This indicates that ignoring the generalizations while taking the associations into account does not lead to a good clustering result. In other words, if the associations are used, the generalizations must be used too.

With respect to the two Boolean parameters p_c and p_i no trends can be distinguished. In the best fifty parameter-tuples all four possible combinations are represented equally.

For the parameter-tuples that lead to a *poor clustering* a clear trend is visible: for both Grizzly and the RIP Worker, the parameter-tuples with $p_{w_d}=0$ give the worst clustering result. Any parameter-tuple with $p_w\neq0$ gives a better clustering result than the same parameter-tuple with $p_{w_d}=0$. This means that ignoring the *dependencies* leads to a poor quality clustering.

However, architects we consulted considered dependencies to be the least important indicator. Instead, most of them base their decomposition on functional criteria. The unexpected importance of dependencies for the result can be explained in two ways:

1. Ignoring the dependencies leaves many classes without any connection to other classes (*unconnected classes*). These classes are then placed in the "unconnected classes" subsystem, which is probably not the right choice.
2. The presence and number of dependencies reflects the functional relations between the classes better than the associations and generalizations do.

If the first explanation holds, ignoring the dependencies must increase the number of unconnected classes much more than ignoring the other relationship-types. Table 3 shows the number of unconnected classes and the number of classes that are only connected with relations of the three types. Observe that the number of unconnected classes in both subsystems is about the same. In the RIP Worker the number of classes that are only connected with dependency relations is relatively high compared to the other relationship types. But for Grizzly this is not the case; much more classes are only connected with a generalization than with a dependency. This means that the first explanation for the importance of the dependencies for the clustering result does not hold.

We therefore assume that dependencies are so important for the clustering result because they reflect the functional relations between the classes better than the other relationship-types.

Table 3. Connectivity of classes in Grizzly and the RIP Worker

Type of classes	Grizzly	RIP Worker
Unconnected	6	7
Only connected with dependency	4	19
Only connected with association	4	1
Only connected with generalization	9	3

Because no single best parameter-tuple could be identified, we decided to use a set of tuples instead of a single one. The overlap between the set of best tuples for Grizzly and for the RIP Worker is very small. In fact, the set of twenty tuples that lead to the best clustering result for each of them are disjoint. We therefore decided to use the union of these two sets, leading to forty tuples to test.

5. Results

Now that a sub-optimal set of parameter-tuples has been identified, the complete controller can be clustered. The same procedure has been followed, where for every parameter-tuple ten clusterings are generated and the average quality is determined. To avoid basing conclusions on a single case, the architectures of the last two versions of the controller, 7e and 8a, are reconstructed.

The structure of software usually does not decrease monotonically. When a system is refactored, its internal structure improves again. Therefore, it is not necessarily the first version of the system in which the architecture is present in its purest form. In order to prevent basing our conclusions on a single case, we decided to combine the two reconstructed versions with *two* other versions, namely the first version, and the version released before the one of which the architecture is reconstructed. This leads to four different version combinations.

5.1 Clustering one version

Table 4 shows the five parameter-tuples that, according to the EdgeMoJo metric, produce the best clusterings for version 7e (left) and 8a (right) of the controller. Recall that we use the EdgeMoJo metric to compare the clustering result to the result of a manual architecture reconstruction. So the parameter-tuples in Table 4 are those that produce decompositions that come closest to a manually reconstructed architecture.

Table 4. Best five clusterings for version 7e and 8a of the controller

Version 7e						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
4	6	1	true	true	1639.4	60.5%
1	4	1	true	true	1644.9	60.5%
1	5	5	true	true	1646.1	60.2%
1	1	5	true	false	1646.8	60.3%
2	5	5	false	false	1648.4	60.4%

Version 8a						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
4	6	1	true	true	1477.1	62.5%
2	1	2	true	false	1481.1	62.3%
6	3	4	true	false	1481.3	62.5%
1	4	3	false	false	1483.8	62.4%
2	3	5	false	true	1484.2	62.3%

As shown in Table 4 a MoJoQuality of 60.5% is achieved for version 7e of the controller. The best clustering for version 8a has a MoJoQuality of 62.5%. Because in both cases the MoJoQuality for the best parameter-tuples exceeds 60%, we consider these decompositions good starting points for manual refinement. Thus, these results *confirm hypothesis H1*. For both versions the parameter tuple (4,6,1,true,true) achieves the best clustering. Although it is tempting to conclude that this is the optimal parameter-tuple, this is probably a coincidence. Recall from Table 2 that for Grizzly and the RIP Worker different parameter-tuples led to the best clustering result.

To confirm hypothesis H2 the EdgeMoJo metric is used. This means that the addition of information from older versions must produce decompositions with an EdgeMoJo value that is lower than 1639.4 for version 7e and lower than 1477.1 for version 8a.

5.2 Clustering multiple versions: class-relations-intersection with first version

Table 5 shows the five parameter-tuples that produce the best clusterings for the class-relations-intersection of the two versions, 7e and 8a, with version 1.

Table 5. Best five clusterings for the class-relations-intersection with version 1

Version 7e with 1						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
0	0	2	false	false	1223.3	73.7%
0	0	1	true	false	1229.5	73.7%
0	0	6	true	false	1266.1	73.3%
1	5	6	false	true	1286.6	72.3%
1	5	5	true	true	1293.5	72.0%

Version 8a with 1						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
0	0	6	true	false	950.5	78.1%
0	0	1	true	false	958.0	78.1%
0	0	2	false	false	980.9	77.8%
0	1	1	true	false	1006.3	76.8%
1	6	4	true	true	1006.7	76.7%

The class-relations-intersection of version 7e and 1 produces decompositions with an EdgeMoJo between 1223.3 and 1398.0. The MoJoQuality varies between 73.7% and 70.9%. For version 8a the EdgeMoJo varies between 950.5 and 1126.0, and the MoJoQuality between 78.1% and 75.5%.

Compared to the clustering based on the versions alone, this is a significant quality improvement. The best tuple with the class-relations-intersection of version 7e and 1 has an EdgeMoJo value that is 25% lower than the best tuple when clustering 7e alone (from 1639.4 to 1223.3). For version 8a the EdgeMoJo improves with 36%.

From this we conclude that basing the clustering on the class-relations-intersection with version 1 leads to a *significantly better clustering*. This confirms hypothesis H2.

5.3 Clustering multiple versions: class-relations-intersection with previous version

Table 6 shows the five parameter-tuples that produce the best clusterings for the class-relations-intersection of the two versions, 7e and 8a, with the version released before them (7d and 7e respectively).

Table 6. Best five clusterings for the class-relations-intersection with the previous version

Version 7e with 7d						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	**Edge MoJo**	**MoJo Quality**
6	2	1	true	true	1642.6	60.2%
1	5	5	true	true	1644.8	60.3%
3	3	4	true	true	1646.9	60.4%
1	5	2	false	true	1651.3	60.3%
4	4	3	false	false	1652.0	60.1%

Version 8a with 7e						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	**Edge MoJo**	**MoJo Quality**
6	1	6	false	true	1642.6	59.1%
4	6	2	false	true	1644.7	59.0%
2	4	3	false	true	1649.1	59.1%
6	5	4	false	true	1651.2	59.0%
3	3	1	false	false	1652.3	58.9%

The class-relations-intersection of version 7e and 7d produces decompositions with an EdgeMoJo between 1642.6 and 1804.1. The MoJoQuality varies between 60.2% and 57.4%. For version 8a the EdgeMoJo varies between 1642.6 and 1811.8, and the MoJoQuality between 59.1% and 56.0%.

These results are similar to the results achieved when using only information from the version of which the architecture is reconstructed: For version 7e the single-version EdgeMoJo is similar to the multi-version result achieved here, for version 8a the single-version EdgeMoJo is even slightly better than the result achieved here. The MoJoQuality metric shows the same pattern. This means that basing the clustering on the class-relations-intersection with the previous version does not lead to a better clustering result.

5.4 Clustering multiple versions: class-relations-union with first version

Table 7 shows the clustering results for the five best clusterings of the class-relations-union of version 8a and 1.

Table 7. Best five clusterings for the class-relations-union of version 8a and 1

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	**Edge MoJo**	**MoJo Quality**
3	3	4	true	true	1458.9	62.9%
4	4	3	false	false	1459.5	62.9%
3	6	1	false	false	1461.2	62.7%
1	4	3	false	false	1461.6	62.8%
6	5	4	false	true	1462.3	62.7%

The clustering of the class-relations-union of version 8a and 1 achieves an EdgeMoJo value between 1458.9 and 1619.8. In the case where version 8a is clustered alone, the EdgeMoJo is between 1468.3 and 1661.4. This means that the class-relations-union does not lead to an improvement of the quality of the clustering. The MoJoQuality for the class-relations-union lies between 62.9% and 59.9%, which is also comparable to the result achieved when clustering with version 8a alone.

5.5 Clustering multiple versions: class-relations-union with previous version

Table 8 shows the clustering results for the five best clusterings of the class-relations-union of version 8a and 7e.

Table 8. Best five clusterings for the class-relations-union of version 8a and 7e

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	**Edge MoJo**	**MoJo Quality**
4	4	3	false	false	1458.5	62.7%
6	3	4	true	false	1466.8	62.6%
4	6	1	true	true	1468.5	62.7%
6	1	6	false	false	1468.6	62.5%
3	6	1	false	false	1469.4	62.6%

The clustering of the class-relations-union of version 8a and 7e achieves an EdgeMoJo value between 1458.5 and 1622.2. Similar to the class-relations-union with version 1, this is comparable to the results when clustering version 8a alone. The MoJoQuality metric confirms this. It now has a value between 62.7% and 59.8%, which is similar to the value achieved when clustering with version 8a alone.

This leads to the conclusion that combining two versions with the class-relations-union operator does *not* lead to an improvement of the clustering result. We therefore decided not to test other combinations of versions.

5.6 Execution times

All performance figures have been measured on the test platform shown in Table 9.

Table 9. Test system characteristics

Processor	Pentium 4; 2,0 GHz
Memory	2 GB
Operating system	Windows 2000 SP4
Java	1.4.2_06
Sniff+	4.2 CP2
MySQL	4.1.8-nt
Bunch	3.3.6
Shrimp	2.0 build 2
Rigi	6.0, version 2-Oct-2003

Table 10 shows some representative examples of the time needed to execute the essential steps of the architectural clustering process for the controller. The fact extraction and subsequent Sniff Import take a lot of time. About half of this time is spent creating Sniff's internal meta-data repository and parsing the source code. The other half is spent importing this information in the database. Both need to be done only once for every analyzed version. Note that building the complete controller from source code takes about one to two hours on our test platform, which is about one order less.

Table 10. Execution times for the controller (wall-clock time)

Task	**Time (hh:mm)**
Fact extraction of version 8a (Sniff+ parsing and import in database)	21:19
Clustering version 8a (Bunch Export, Clustering and Bunch Import)	0:18
Clustering class-relations-intersection of version 8a and 1 (Bunch Export, Clustering and Bunch Import)	0:11
Clustering class-relations-union of version 8a and 1 (Bunch Export, Clustering and Bunch Import)	0:58
MoJo calculations for version 8a	0:05
Visualization version 8a (RSF Export and loading in Shrimp)	0:05

The ten-clustering cycle combines several of these steps. Table 11 shows the execution times of the executed ten-clustering cycles. The class-relations-union of version 7e with the first and the last version have not been measured.

Table 11. Execution times of the ten-clustering cycles (wall-clock time)

Task	**Time (hh:mm)**	
	7e	**8a**
One version	2:58	3:11
Class-relations-intersection with first version	1:01	1:02
Class-relations-intersection with previous version	3:03	2:42
Class-relations-union with first version	n.m.	4:41
Class-relations-union with previous version	n.m.	3:34

5.7 Problems encountered

The size of the controller caused several problems. First of all, MySQL could only execute the queries after tuning it for large databases. Second, a complete cycle of fact extraction, clustering and result assessment or visualization took a significant amount of time. For one clustering cycle this is no problem, but it is when a large number of clustering cycles are performed. Effectively, this limits the number of different clusterings that can be created, and hence the number of different parameter-tuples that can be tested. Note that in practical clustering-based architecture reconstruction cases only one, or a limited number of clusterings are generated. Therefore this limitation applies mainly to projects experimenting with different clustering approaches or parameters, and not to practical architecture reconstruction cases.

Although Sniff+ proved to be a reliable and stable fact extractor, the Sniff API caused some problems with associations based on C++ templates. Because of the low number of these associations in the controller this had little impact on the clustering result.

6. Conclusions

From the results of the previous chapter we conclude architectural clustering based on structural relations between the classes can reconstruct architectural views of object-oriented software that are useful for software maintenance, which confirms our first hypothesis.

The quality of the decompositions our architectural clustering method produced is relatively good in the sense that they approach the result of a manual architecture reconstruction relatively well. In our experiments where the architecture of two versions of the controller was reconstructed the produced decompositions had a MoJoQuality of 60.5% and 62.5% respectively. This exceeds our goal of 60%.

Furthermore, the execution time is such that clustering systems of the size of the complete controller is feasible in practice.

The weight of the relationship-types significantly affects the quality of the clustering result. However, in our experiments there was no single combination of weights that produced the best clusterings for all analyzed pieces of software.

Dependency relations are very important for the quality of the clustering result. In all experiments ignoring the dependencies led to a reduction of the clustering result's quality, regardless of the weight assigned to the other relationship-types.

Incorporating information from other versions in the clustering process can improve the clustering result. The quality improves if the clustering is based on those class-relations that are *also* present in the *first* version of the software (class-relations-intersection with version one). We see an improvement of about 20% to 35%, confirming our second hypothesis. If instead of the first, the previous version is used, no improvement is achieved. This might be due to the fact that in the previous version the architecture is deteriorated much further than in the first version.

Clustering based on the class-relations present in the clustered version *or* the first one (class-relations-union) does not lead to better clustering results. The same holds for the combination with the previous version instead of the first.

References

[1] A. Trevors, M.W. Godfrey. Architectural Reconstruction in the Dark. Position paper, *Workshop on Software Architecture Reconstruction*, collocated with WCRE '02, Oct. 2002.

[2] J. Buckley. Some standards for software maintenance. Standards, *IEEE Computer*, Nov. 1989.

[3] R.K. Fjeldstadt, W.T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. *Proc. GUIDE 48*, IEEE Computer Society Press, Apr. 1984.

[4] S. Ducasse. *Reengineering Object-Oriented Applications*. Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, Sept. 2003, IAM-03-008.

[5] S. Demeyer, S. Ducasse, O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers, San Francisco, CA, USA. 2004.

[6] A. Quilici. Reverse Engineering of Legacy Systems: A Path Towards Success. *Proc. 17th International Conference on Software Engineering (ICSE'95)*, Apr. 1995, pp. 333-336.

[7] L. O'Brien, C. Stoermer, C. Verhoef. Software Architecture Reconstruction: Practice Needs and Current Approaches. SEI Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, Aug. 2002.

[8] A. van Deursen. Software Architecture Recovery and Modelling [WCRE 2001 Discussion Forum Report]. *ACM SIGAPP Applied Computing Review*, 10(1), 2002.

[9] A.E. Hassan, R. Holt. The Small World of Software Reverse Engineering. *Proc. 2004 Working Conference on Reverse Engineering (WCRE'04).* Nov. 2004. pp. 278-283.

[10] S.E. Sim, R. Koschke. WoSEF: Workshop on Standard Exchange Format. *ACM SIGSOFT Software Engineering Notes*, 26, Jan. 2001, pp. 44-49.

[11] S. Bassil, R.K. Keller. Software Visualization Tools: Survey and Analysis. *Proc. 9th International Workshop on Program Comprehension (IWPC'01)*, 2001, p. 7.

[12] M.E. Conway. How Do Committees Invent? *Datamation Magazine*, 14(4), Apr. 1968, pp. 28-31.

[13] J.D. Herbsleb, R.E. Grinter. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, 16(5), Sept./Oct. 1999, pp. 63-70.

[14] J. Beck, D. Eichmann. Program and Interface Slicing for Reverse Engineering. *Proc. 15th International Conference on Software Engineering (ICSE'93)*, 1993, pp. 509-518.

[15] J. Zhao. A Slicing-Based Approach to Extracting Reusable Software Architectures. *Proc. 4th European Conference on Software Maintenance and Reengineering*, Feb. 2000, pp. 215-223.

[16] P. Berkhin. *Survey of Clustering Data Mining Techniques*. Technical report, Accrue Software, San Jose, California, 2002.

[17] A.K. Jain, M.N. Murty, P.J. Flynn. Data Clustering: A Review. *ACM Computing Surveys*, 31(3), Sept. 1999, pp. 264-323.

[18] S.K. Pal, P. Mitra. *Patterns Recognition Algorithms for Data Mining*. Chapman & Hall/CRC, 2004.

[19] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36, Mar. 1997, pp. 211-231.

[20] T.A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. *Proc. 4th Working Conference on Reverse Engineering (WCRE '97)*, 1997, p. 33.

[21] V. Tzerpos, R. C. Holt. Software Botryology: Automatic Clustering of Software Systems. *Proc. International Workshop on Large-Scale Software Composition*, Aug. 1998.

[22] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Institut für Informatik, Universität Stuttgart, 2000.

[23] B.S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD thesis, Drexel University, Mar. 2002.

[24] R.W. Schwanke. An Intelligent Tool for Re-engineering Software Modularity. *Proc. 13th International Conference on Software Engineering*, 1991, pp. 83-92.

[25] Rigi. http://www.rigi.csc.uvic.ca/.

[26] H.A. Müller, J.S. Uhl. Composing Subsystem Structures using (k,2)-partite Graphs. *Proc. Conference on Software Maintenance*, Nov 1990, pp. 12-19.

[27] A. Shokoufandeh, S. Mancoridis, T. Denton, M. Maycock. Spectral and meta-heuristic algorithms for software clustering. *Journal of Systems and Software*, accepted Mar. 2004, published online.

[28] V. Tzerpos, R.C. Holt. MoJo: A distance metric for software clusterings. *Proc. 6th Working Conference on Reverse Engineering (WCRE'99)*, Oct. 1999, pp. 187-193.

[29] B.S. Mitchell, S. Mancoridis. Using Heuristic Search Techniques to Extract Design Abstractions from Source Code. *Proc. Genetic and Evolutionary Computation Conference (GECCO'02)*, Jul. 2002.

[30] N. Anquetil, T.C. Lethbridge. Experiments with Clustering as a Software Remodularization Method. *Proc. Sixth Working Conference on Reverse Engineering (WCRE'99)*, 1999, p. 235.

[31] InSight. http://www.klocwork.com/products/insight.asp.

[32] Z. Wen, V. Tzerpos. Evaluating similarity measures for software decompositions. *Proc. International Conference on Software Maintenance (ICSM'04)*, Sept. 2004, pp. 368-377.

[33] Z. Wen. V. Tzerpos. An optimal algorithm for MoJo distance. *Proc. of the 11th International Workshop on Program Comprehension (IWPC'03)*, May 2003, pp. 227-235.

[34] V. Tzerpos, R.C. Holt. ACDC: An Algorithm for Comprehension-Driver Clustering. *Proc. seventh Working Conference On Reverse Engineering (WCRE'00)*, 2000, pp. 258-267.

[35] M. Trifu. Architecture-Aware, *Adaptive Clustering of Object-Oriented Systems*. Diploma thesis, Forschungszentrum Informatik, Karlsruhe, Germany, Sept. 2003.

[36] Z. Wen, V. Tzerpos. An effectiveness measure for software clustering algorithms. *Proc. 12th International Conference on Program Comprehension (IWPC'04)*, Jun. 2004, p. 194.

[37] M. Shtern, V. Tzerpos. A Framework for Comparison of Nested Software Decompositions. *Proc. 11th Working Conference on Reverse Engineering (WCRE'04)*, Nov. 2004.

[38] H. Müller, M. Orgun, S. Tilley, J. Uhl, A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5, 1993, pp. 181-204.

[39] Bunch. http://serg.cs.drexel.edu/projects/bunch/.

[40] B.S. Mitchell, S. Mancoridis. Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements. *Proc. International Conference on Software Maintenance (ICSM'01)*, Nov. 2001.

[41] S.C. Choi, W. Scacchi. Extracting and Restructuring the Design of Large Systems. *IEEE Software*, 7(1), Jan. 1990, pp. 66-71.

[42] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A.M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, J. Weisbrod. *The FAMOOS Object-Oriented Reengineering Handbook*. Oct., 1999.

[43] M. Bauer, M. Trifu. Architecture-Aware Adaptive Clustering of OO Systems. *Proc. eight European Conference on Software Maintenance and Reengineering (CSMR'04)*, 2004.

[44] *Setup and User's Guide to Columbus/CAN*, Academic Version 3.5. FrontEndART Ltd, Jan. 2003.

[45] S. Ducasse, T. Girba, J.-M. Favre. Modeling Software evolution by Treating History as a First Class Entity. *Proc. Workshop on Software Evolution through Transformations* (SETra 2004).

[46] A. Silberschatz, H. Korth, S. Sudarshan. *Database System Concepts*, 4th edition. McGraw-Hill Higher Education, 2002.