Evaluating the Impact of Software Evolution on Software Clustering

Fabian Beck and Stephan Diehl
University of Trier, Germany
Email: {beckf,diehl}@uni-trier.de

***Abstract*—The evolution of a software project is a rich data source for analyzing and improving the software development process. Recently, several research groups have tried to cluster source code artifacts based on information about how the code of a software system evolves. The results of these evolutionary approaches seem promising, but a direct comparison to traditional software clustering approaches based on structural code dependencies is still missing. To fill this gap, we conducted several clustering experiments with an approved software clustering tool comparing and combining the evolutionary and the structural approach. These experiments show that the evolutionary approach could produce meaningful clustering results. But still the traditional approach provides better results because of a more reliable data density of the structural data. Finally, the combination of both approaches is able to improve the overall clustering quality.**

I. Introduction

Software clustering is an important discipline in reverse engineering and software maintenance. It deals with the automatic unsupervised grouping of software artifacts like functions, classes, or files into high-level structures like packages, components, or sub-systems based on the similarity of the artifacts. Software clustering is applied, for instance, to understand complex software systems [12], to restructure software architectures [2], to identify reusable components [11], or to detect misplaced software artifacts [20].

Software clustering algorithms rely on characteristic information about the software artifacts to compute a pairwise similarity measure and to finally yield a reasonable clustering. Common clustering approaches retrieve this information directly from the static source code in form of structural dependencies based on, for example, method invocations and variable references among methods [1], [12], [14], or inheritance, aggregation, and method invocations among classes [17], [24]. Some approaches try to improve the clustering by taking dynamic code dependencies recorded during the program execution into consideration [6], [26]. Other approaches use the source code only indirectly by analyzing variable names and comments [9]. Although there exists such a variety of data sources, only few approaches integrate several of them into their clustering technique (e.g., [1], [24]).

In the last decade, software engineers have become aware of software evolution as an important and largely unused data source to enhance the software development and maintenance process. Information from the evolution of software projects, in particular, information on how developers change the source code of a software system, has been leveraged across many applications: It helps project managers to control the development process [10], software architects to detect design flaws [5], developers to find related files or hidden dependencies [27], and quality controllers to identify bugs [7]. But can it also be leveraged to cluster software artifacts?

A. Evolutionary Data in Software Clustering

Recently, some research groups have started to link ideas from both fields of research—software clustering and software evolution. Clustering-centered approaches enrich structural data with some evolutionary aspects. Andritsos and Tzerpos [1] integrated the file ownership information, a simple evolutionary data source, and improved a clustering approach based on structural data slightly. However, they also found that the integration of the file timestamps, another simple evolutionary data source, tends to decrease the clustering quality. Wierda et al. [24] used the assumption that the intended architecture of a software system is represented in a purer form in the initial version than in a later one. In their case study combining the current version with the first version by intersecting their structural dependencies improved the decomposition.

Other approaches are more evolution-centered and work on fine-grained co-change data (files that are often changed together): In one of the first works on software evolution, Ball et al. [3] used a specialized graph layout algorithm on an evolutionary co-change graph. Clusters emerge as visual groups in the graph visualization. Beyer and Noack [4] refine this approach by revising the graph data structure and the layout algorithm. In both cases the user, however, needs to finally mark the clusters manually. Voinea and Telea [21] integrate a clustering algorithm based on software evolution into their visualization tool *CVSgrab*. The evolutionary clustering is used to improve the sequential order of the files in the visualization. Vanya et al. [20] are able to identify design flaws in the software architecture by comparing the software-evolution-based clustering decomposition to the current architecture of the software. In a case study, experts rated most of the detected design flaws as valuable information.

B. Objectives

Methods that integrate software clustering and software evolution, such as those discussed above, seem promising. On the one hand, software clustering based on structural data

might be improved by integrating evolutionary data sources and, on the other hand, it appears to be possible to cluster software only by using evolutionary data.

But despite these positive results, there are some aspects that are not covered sufficiently yet. The clustering-centered approaches only employ very basic evolutionary data: file ownership, timestamps, or the first and latest program version. The timestamp information even gives an example that the integration of evolutionary information might change the clustering result for the worse. The evolution-centered approaches only show that these clustering techniques are working to some degree, they are not contrasted to approaches based on structural data.

Our goal is to overcome these shortcomings by comparing structural and evolutionary data sources directly to each other. We use an approved software clustering approach to recover the architecture of six software projects based on three different kinds of dependencies: static structural source code dependencies, evolutionary co-change dependencies, and combined structural and evolutionary dependencies. Different filtering setups and combination strategies lead to a total of 152 different concrete dependency graphs per project. For each of these graphs, we compute a clustering and assess its quality by measuring its similarity to a reference decomposition.

The main contributions of this paper are:

- The first systematic comparison of structural and evolutionary data for software clustering.
- One of the most extensive studies in software clustering with respect to the number of analyzed projects and project sizes.
- The first approach that consequently integrates evolutionary information into a traditional clustering technique.
- An assessment of the influence that filtering evolutionary data has on the clustering quality.

The rest of this paper is organized as follows. Section II introduces an appropriate experimental design that is able to cluster software systems and to measure the quality of the resulting software decompositions. Section III presents the study, consisting of three experiments. Finally, Section IV discusses the validity of the results and Section V concludes the findings.

II. Experimental Design

Our study concentrates on grouping classes into packages. Classes are the elementary units in the design process of object-oriented software systems. Their organization into packages reflects the architecture of a software system. Since interfaces are similar to classes, we handle them like classes in the experiments and therefore use the term *class* for classes as well as for interfaces.

A. Sample Software Projects

We employ a set of six open source *Java* projects as subjects of the study (see Table I). Although this set of projects cannot be considered statistically representative of the whole population of software projects, it covers a wide range of project types—from user clients and libraries to server applications. The numbers of classes (based on the latest version) give an idea of the project sizes: *JFtp* is the smallest project examined with only 78 classes while *JEdit* is the largest one with 840 classes. Note that in the cases of the *Azureus* and *Tomcat* project, we only considered one of their main packages because our experimental setup is not able to handle more than 1000 classes efficiently. We excluded all test case packages in *JUnit* because they are arranged in two large unstructured test packages—a structure that conflicts with the idea of grouping dependent files together.

B. Data Sources

In the following experiments the dependency information is the independent variable. A directed graph, where nodes represent classes and edges represent dependencies, models this asymmetric dependency information.

1) Structural Dependencies: Structural static source code dependencies (short: structural dependencies) are the most widely used data source for software clustering. As such, they represent the conventional approach and constitute the control group in our study. We incorporate all main types of structural class dependencies, namely inheritance, aggregation, and usage (e.g., method calls, method parameters, local variables).

Definition 1. *Let $C(S)$ be the set of classes of a software system S. We define four directed relations on classes:*

$$\begin{aligned} (c_1, c_2) \in E_{\mathrm{CIG}} &\Leftrightarrow c1 \textit{ extends } c2 \\ (c_1, c_2) \in E_{\mathrm{CAG}} &\Leftrightarrow c1 \textit{ aggregates } c2 \\ (c_1, c_2) \in E_{\mathrm{CUG}} &\Leftrightarrow c1 \textit{ uses } c2 \\ E_{\mathrm{SCDG}} &:= E_{\mathrm{CIG}} \cup E_{\mathrm{CAG}} \cup E_{\mathrm{CUG}} \end{aligned}$$

*The directed graphs $G_x := (C(S), E_x)$ are called Class Inheritance Graph (*CIG*), Class Aggregation Graph (*CAG*), Class Usage Graph (*CUG*), and Structural Class Dependency Graph (*SCDG*).*

To retrieve these graphs, we use *DependencyFinder*, a code analysis suite that works on *Java* bytecode and, among other things, is able to extract all relevant dependencies.

2) Evolutionary Dependencies: The evolution of a software project is documented by the changes applied to its source files in the course of development. In modern software engineering a revision control system (version archive), such as *CVS* or *SVN*, stores these changes. Transactions—changes simultaneously submitted to the version archive by the same developer—are the elementary units in these systems.

Class A depends on class B by evolution if class A has often been changed together with class B. In other words, both classes have often been part of the same transactions. This is the basic idea behind evolutionary dependencies (also referred to as evolutionary couplings or co-change couplings).

This approach is based on the assumption that transactions implicitly group dependent files together. But some transactions might relate files randomly, for example, if a developer

TABLE I
CHARACTERISTIC DATA OF THE SAMPLE SOFTWARE PROJECTS AND THEIR REPOSITORIES.

Project	Description	Archive	Time frame	# Classes		# Transactions	# Developers
Azureus	*BitTorrent* client	CVS	2003/07/10 – 2007/02/14	477	restricted to `org.gudy.azureus2.core3`	10665	27
JEdit	text editor	SVN	2001/09/02 – 2007/02/12	840		2190	20
JFreeChart (JFC)	*Java Swing* chart library	CVS	2001/10/18 – 2007/02/14	794		2413	5
JFtp	*FTP* client	CVS	2002/01/25 – 2003/03/23	78		210	5
JUnit	regression testing framework	CVS	2002/12/12 – 2007/02/08	103	test cases excluded	673	7
Tomcat	*Java Servlet* implementation	SVN	2006/03/27 – 2007/03/10	561	restricted to `org.apache.catalina`	661	13

fixed two totally unrelated bugs in a single transaction. Hence, a mechanism that allows filtering out such noise and considers only strong dependencies might improve the efficiency of the evolutionary data.

Zimmermann et al. [29] introduce the concept of support and confidence to measure the strength of evolutionary dependencies. The support value of a dependency counts how often the two software artifacts were changed together. Additionally, the confidence value of a dependency relates the support to the total number of changes applied to one of the artifacts.

Definition 2. *Let $c_1, c_2 \in C(S)$ be two classes and $\{T_i\}_{i=1}^{l}$ a sequence of transactions.*

$$\mathrm{Supp}(c_1, c_2) := |\{T_i : c_1 \in T_i, c_2 \in T_i\}|$$

is called ***support*** *of the evolutionary dependency of class c_1 to class c_2.*

If c_1 is element of at least one transaction T_i, the ***confidence*** *of the evolutionary dependency of class c_1 to class c_2 is defined as*

$$\mathrm{Conf}(c_1, c_2) := \frac{\mathrm{Supp}(c_1, c_2)}{\mathrm{Supp}(c_1, c_1)}$$

Otherwise, $\mathrm{Conf}(c_1, c_2) := 0$.

In the definition of Conf the value of $\mathrm{Supp}(c_1, c_1)$ represents the total number of transactions c_1 is part of. Thus, the value of Conf reaches its maximum 1 if c_2 is changed whenever c_1 is changed. Note that Supp is a symmetric function whereas Conf is not.

Analogously to the structural dependencies, an evolutionary dependency graph can be defined with the help of the Conf and the Supp functions. In contrast to the structural graphs, the evolutionary graph depends on two parameters that filter out weak dependencies: a confidence threshold α and a support threshold k. As part of our experiments, we have to find a reasonable setting for these parameters.

Definition 3. *Let $C(S)$ be the set of classes of a software system S. We define a directed relation of classes*

$$(c_1, c_2) \in E_{\mathrm{ECDG}_\alpha^k} \Leftrightarrow \mathrm{Supp}(c_1, c_2) > k \wedge \mathrm{Conf}(c_1, c_2) > \alpha$$

The directed graph $G_{\mathrm{ECDG}_\alpha^k} := (C(S), E_{\mathrm{ECDG}_\alpha^k})$ is called ***Evolutionary Class Dependency Graph*** *(ECDG_α^k) with parameters $\alpha \in [0, 1)$ (confidence threshold) and $k \in \mathbb{N}$ (support threshold).*

We use the approach by Weißgerber and Zimmermann [22], [28] to extract the evolutionary class dependency graphs from the version archives.

As common when mining software repositories, we omit large transactions (here, transactions with more than 50 participating classes) to reduce noise in the evolutionary dependency data. The converter also ignores classes from branched versions to avoid conflicts caused by multiple copies of the same class.

Although it is possible to relate non-source files with the concept of evolutionary dependencies, in the context of this study we restrict it to source files to guarantee the comparability to structural dependencies. In real world applications, the possibility to cluster non-source files might be a crucial advantage of evolutionary dependencies over structural dependencies.

This approach is different to the two clustering approaches based on co-change information [20], [21], which we already discussed in the introduction. We chose a graph data model to match the structural dependency graphs and apply a specialized software clustering algorithm that will be introduced in the following section. In contrast, the two other approaches use similarity matrices and a non-specialized agglomerative clustering algorithm.

C. Clustering Algorithm

In general, a clustering algorithm divides a set of entities—here, classes—into clusters. The result of this division is called clustering decomposition. Depending on the algorithm, the decomposition is either flat or hierarchical. Various software clustering approaches have been proposed and studied (e.g., Maqbool and Babri [14] provide an overview). Since this work does not aim at improving a particular clustering algorithm directly but at assessing the quality of different data sources, any approved software clustering algorithm would serve the purpose.

Bunch [13], [16] is a graph-based clustering tool that follows the concept of low coupling and high cohesion [18]. Its clustering algorithm optimizes a clustering quality metric with a heuristic search technique and produces hierarchical clustering decompositions. Several evaluations showed that *Bunch* is among the best currently available software clustering tools [1], [14], [25]. Since *Bunch* is customizable, we tried to find a good parameter setting. The resulting setup is similar to the ones used in the evaluations cited above.

Bunch works on a graph structure that is called *Module Dependency Graph*. The graph represents modules as nodes and module dependencies as directed edges. Since the terms *module* and *dependency* are not bound to a strict definition, we are allowed to consider the weighted dependency graphs defined in Section II-B as *Bunch Module Dependency Graphs*.

Bunch provides three optimization strategies: an exhaustive search algorithm, a hill climbing algorithm, and a genetic algorithm. We prefer the hill climbing algorithm because it produces stable high quality results efficiently in predictable runtime. When using the hill climbing algorithm, further parameters need to be set. Based on some performance tests with the *JFtp* project and a study by Mitchell and Mancoridis [17], we set the initial population size to 1, chose the nearest ascent hill climbing option, and deactivated further algorithm extensions.

Since *Bunch* uses a heuristic search approach that has a random element, the clustering process can be considered a random experiment. Mitchell and Mancoridis showed in their study about *Bunch* [17] that in most cases the resulting decompositions differ only slightly. Nevertheless, we perform several repetitions of our experiments to increase the reliability of the results. Due to these repetitions and some performance problems of *Bunch*, we had to restrict our experiments to less than 1000 classes per project as mentioned earlier.

D. Evaluation Method

The quality of a software decomposition can be evaluated either by considering internal quality criteria (internal assessment), or by comparing it to a reference decomposition (external assessment). Internal metrics do not measure the quality independently or are only applicable to different algorithms working on the same input graph. In contrast, an external assessment allows an independent quality measure and does not rely on a constant input graph.

1) Reference Decompositions: The approach of creating a reference decomposition as a benchmark for an external assessment assumes that a perfect clustering exists, which is closely approximated by the reference decomposition. As the quality of a decomposition depends on the developers' opinions, an objective and reliable reference, however, does not exist—the approach is only a heuristic to estimate the quality of a software clustering method.

A good reference decomposition can be created by using the current factual architecture of the system (e.g., the package structure of an object-oriented system) or by employing domain experts, which perform the task manually. Since we cannot count on domain experts for all six examined projects, we chose the factual decompositions as the most efficient method. It is reasonable to assume that these decompositions have a good quality because they have also been created by domain experts—the developers themselves.

2) Similarity of Decompositions: The clustering decomposition that is most similar to the reference is considered the best result. Though, there does not exist a natural metric that measures the similarity between two decompositions, but some heuristics are proposed in literature.

Tzerpos and Holt [19] developed a metric, called *MoJo*, that is estimating the distance between two decompositions with the minimal number of *Move* and *Join* operations needed to transform one decomposition into the other. Furthermore, Wen and Tzerpos [23] introduced *MoJoFM*, a revision of *MoJo* normalized by the decomposition most distant to the reference decomposition. *MoJoFM* ranges from 0, representing the most distant decomposition, to 100, representing a clustering that is completely identical to the reference. It simulates the operations a user would perform to transform one decomposition into the other. The metric is clear and simple to understand. For these reasons we preferred *MoJo* over comparable metrics like the *Precision/Recall* metric [2] or the *Koschke-Eisenbarth* metric [8].

Definition 4. *Let* $\mathrm{mno}(X, Y)$ *be the minimum number of Move and Join operations that is needed to transform a flat decomposition* X *into a flat decomposition* Y*. For two flat decompositions* A, B

$$\mathrm{MoJoFM}(A, B) := 100 \cdot \left(1 - \frac{\mathrm{mno}(A, B)}{\max_x(\mathrm{mno}(x, B))}\right)$$

is called the ***MoJoFM similarity*** *from* A *to* B*.*

As B is the criterion for the normalization, the reference decomposition has to be represented by B. The polarity of the scale is reversed to get a measure of similarity instead of distance. Despite the normalization of *MoJoFM*, one cannot compare the *MoJoFM* values of different sample projects directly because the normalization depends on the structure of the reference decomposition. Only comparisons on the same reference—i.e., on the same project—are valid.

Since *MoJoFM* works only on flat decompositions, we have to transform the hierarchical decompositions retrieved from the package structure as well as from *Bunch* into flat decompositions. To transform the reference decomposition, we use the low-level package partition ignoring the package hierarchy. To transform the hierarchical clustering result, we cut the hierarchy on the level where the resulting flat decomposition is most similar to the reference (based on the *MoJoFM* value). This solution avoids noise or a bias caused by too fine- or coarse-grained clustering decompositions.

III. Study

One of the main use cases of software clustering is architecture recovery. When the architecture of a software system is totally undocumented or the documentation is just outdated, software clustering helps to retrieve the current architectural information. An automatically recovered architecture also supports the developers to redesign a badly structured system.

The following study addresses this problem of architecture recovery. It evaluates the quality of a clustering decomposition in terms of its similarity to the package structure as described in Section II-D. This approach of retrieving the already documented package structure by a clustering algorithm may sound

strange, but is an approved assessment method for software clustering algorithms [12], [2], [25]. The goal is not to use this procedure in practical application, but to get a measure for the quality of a clustering approach. To this end we need examples where we already know the *correct* answer. Outside our experimental environment, we would of course only apply the clustering algorithm to projects or subsystems with no documented structure or an assumed bad structure.

The study consists of three experiments. Experiment 1 contrasts structural and evolutionary dependency graphs in the application of software clustering. Experiment 2 looks at the dependency quality in these graphs to better understand the previously gained results. Finally, Experiment 3 focuses on combining both data sources. Each of the experiments tries different setups of dependency graphs to find the best possible solution.

TABLE II
DEPENDENCY GRAPH SIZES.

		Azur.	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*
# Nodes		477	840	794	78	103	561
# Edges	CIG	279	318	484	40	55	312
	CAG	434	691	223	66	49	455
	CUG	2269	4020	3714	38	294	2207
	SCDG	2362	4117	3937	127	306	2348
	$ECDG^0_{0.0}$	6218	29372	13922	1184	642	696
	$ECDG^0_{0.2}$	1354	8438	9512	738	487	502
	$ECDG^0_{0.4}$	727	4011	5427	414	310	293
	$ECDG^0_{0.6}$	345	2117	2478	261	146	177
	$ECDG^0_{0.8}$	277	1655	2063	166	105	169
	$ECDG^1_{0.0}$	2330	9430	2376	530	138	84
	$ECDG^1_{0.2}$	650	3289	1991	365	116	63
	$ECDG^1_{0.4}$	302	1542	1334	251	92	45
	$ECDG^1_{0.6}$	172	888	930	171	71	25
	$ECDG^1_{0.8}$	104	426	515	76	30	17
	$ECDG^2_{0.0}$	1258	4538	696	322	42	24
	$ECDG^2_{0.2}$	419	1713	588	231	33	23
	$ECDG^2_{0.4}$	193	814	338	164	14	18
	$ECDG^2_{0.6}$	99	390	172	122	9	13
	$ECDG^2_{0.8}$	51	247	56	50	5	11

Table II characterizes the dependency graphs of the studied software projects in terms of their numbers of nodes and edges. The selected sample projects are significantly varying in the number of nodes: the smallest project, *JFtp*, consists of 78 nodes (i.e., classes), while the largest project, *JEdit*, has 840 nodes.

Comparing the simple structural dependency graphs with respect to their edge density, the *Class Usage Graph* (CUG) and the *Structural Class Dependency Graph* (SCDG) are far denser than the *Class Inheritance Graph* (CIG) and the *Class Aggregation Graph* (CAG). Thus, clustering might be harder only using inheritance or aggregation dependencies.

The *Evolutionary Class Dependency Graph* (ECDG) depends on two parameters, the support threshold and the confidence threshold. These two parameters can be considered as filters that are getting stronger (i.e., reducing the number of dependencies) with increasing values. Table II confirms that the number decreases block by block for increasing support values and line by line for increasing confidence values. Only the slightly filtered graphs contain extensive dependency information. Since it is hard to define reasonable threshold values in advance, the first experiments will vary support and confidence systematically. Finding a good filtering setup is a trade-off between dependency efficiency and dependency density.

A. Experiment 1: Simple Data Sources

The first experiment addresses the question whether it is possible to get meaningful clustering decompositions using only structural or evolutionary data sources: The experiment compares the clustering results for the CIG, CAG, CUG, and SCDG to the ECDG in different filtering setups. Table III presents the results of the experiment regarding the *MoJoFM* metric values for each clustering setup. As mentioned in Section II-C repeated runs were performed ($n = 50$) and averaged to increase the precision of the quality information.

A precision measure of average values is the standard error $\hat{\sigma}_{\overline{x}}$ (the standard deviation of the mean values). In Table III and all following tables containing clustering results, less precise *MoJoFM* values with a standard error of $0.5 \leq \hat{\sigma}_{\overline{x}} < 1.0$ are marked with ' and those with a standard error of $1.0 \leq \hat{\sigma}_{\overline{x}}$ with *. The upper bound of the standard error completes each table. Moreover, we highlight the best overall quality values in light gray and the best project specific ones in gray.

TABLE III
MOJOFM CLUSTERING QUALITY.

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 1.7$	*Azur.*	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*	**avg**
CIG	30.2	42.0	35.4	53.4	28.5	32.8	**37.0**
CAG	28.6	47.1'	14.3	45.7	21.6	35.8	**32.2**
CUG	52.4	63.5*	40.6'	49.4	33.5	54.5'	**49.0**
SCDG	49.9'	65.5*	44.2	54.0	35.0	54.0'	**50.4**
$ECDG^0_{0.0}$	31.7	40.4	40.7	41.5	18.5	11.1	**30.7**
$ECDG^0_{0.2}$	34.7	46.2	41.3*	40.0	19.4	11.1	**32.1**
$ECDG^0_{0.4}$	35.6	47.0	43.2	42.3	20.7	11.3	**33.3**
$ECDG^0_{0.6}$	33.6	47.2	40.9	42.4	21.1	11.4	**32.8**
$ECDG^0_{0.8}$	32.6	45.9	39.8	42.7	20.6	11.4	**32.2**
$ECDG^1_{0.0}$	29.0	36.5	29.1	35.5	18.2	8.0	**26.0**
$ECDG^1_{0.2}$	30.7	40.0	29.3	35.2	17.5	8.0	**26.8**
$ECDG^1_{0.4}$	31.3	39.6	28.2	35.2	17.5	7.5	**26.6**
$ECDG^1_{0.6}$	29.6	37.5	26.5	35.9	18.3	6.9	**25.8**
$ECDG^1_{0.8}$	27.0	36.9	20.9	35.4	13.4	6.8	**23.4**
$ECDG^2_{0.0}$	27.3	32.9	13.8	35.1	11.0	6.4	**21.1**
$ECDG^2_{0.2}$	27.3	34.6	14.2	35.3	9.9	6.4	**21.3**
$ECDG^2_{0.4}$	28.0	32.8	12.9	35.3	11.0	5.8	**21.0**
$ECDG^2_{0.6}$	27.1	30.9	11.7	36.0	9.9	5.8	**20.2**
$ECDG^2_{0.8}$	23.3	30.0	6.8	36.1	8.8	6.0	**18.5**
Default_1	16.3	15.7	1.8	36.6	3.3	4.1	**13.0**
Default_n	11.3	0.4	2.8	0.0	4.4	2.1	**3.5**

1) Results for Structural Graphs: Starting with the simple structural dependencies (CIG, CAG, and CUG), the highest *MoJoFM* values, indicating a high agreement with the reference decomposition, are mostly reached with the CUG. This fits with the observation that the CIG and CAG usually do not contain as much information as the CUG. There exist, however, some exceptions: For *JUnit* the quality of the

CIG, CUG, and CAG results are nearly equal although the CUG has much more dependencies than the CIG and CAG (Table II). For *JFtp* the CIG (53.4) even outperforms the CUG (49.4).

The SCDG, as an aggregation of the CIG, CAG, and CUG, incorporates the information from the three simple structural graphs and increases or at least steadies the clustering quality of the simple data sources: The average clustering quality of the SCDG (50.4) is clearly higher than the CIG and CAG average values (37.0 and 32.2) and at least slightly higher than the CUG value (49.0).

2) Results for Evolutionary Graphs: The $\mathrm{ECDG}^0_{0.4}$, which has a support threshold of 0 and a confidence threshold of 0.4, produces the best average quality for evolutionary dependencies (33.3), closely followed by the other evolutionary graphs with a support threshold of 0 (please recall that a threshold of 0 means that the support must be at least 1). This clustering quality is in the range of the CIG (37.0) and the CAG (32.2) but clearly lower than the CUG (49.0) and SCDG (50.4) values.

These characteristics of the average value need not be valid for each of the individual projects. For instance, the evolutionary clustering quality for the *Tomcat* project is very low (5.8 to 11.4) and far from being competitive to any structural dependency information. This is probably caused by its sparse evolutionary class dependency graph (Table II). In contrast, for *JFreeChart* the relation between structural and evolutionary data sources is nearly balanced (best structural: 44.2; best evolutionary: 43.2).

3) Default Clustering Decomposition: Additional to the clustering results, Table III lists two default quality metric values in the last rows: $\mathrm{Default}_1$ represents a decomposition that consists of only one huge cluster; $\mathrm{Default}_n$ represents a decomposition that consists of n singleton clusters. These values provide a reference for the *MoJoFM* values of a project.

The default decomposition qualities cover a wide range of *MoJoFM* values, from 0.0 (*JFtp*, $\mathrm{Default}_n$) up to 36.6 (*JFtp*, $\mathrm{Default}_1$). This observation underpins clearly that comparisons of clustering results based on *MoJoFM* are only valid for the same reference decomposition (i.e., the same sample software project). The *MoJoFM* difference between a clustering and the best default clustering gives a hint at the overall clustering quality. In contrast, taking the absolute value into account might be misleading. For instance, the structural clustering seems to work better for *JEdit* (SCDG: 65.5; $\mathrm{Default}_1$: 15.7) and *Tomcat* (CUG: 54.5; $\mathrm{Default}_1$: 4.1) than for *JFtp* (SCDG: 54.0; $\mathrm{Default}_1$: 36.6) although their absolute *MoJoFM* values are similar. Regarding the evolutionary dependencies, only *JFtp* and *Tomcat* do not exceed their default values clearly, but at least slightly. Finally, we have to state that *MoJoFM* is a relative, project specific quality metric and not an absolute measure where a value of x indicates a good clustering result.

Result 1. *The usage graph as well as the aggregated structural graph significantly outperform all other data sources. The slightly filtered evolutionary dependencies produce results similar to the inheritance and aggregation dependencies. At least in four of six cases these evolutionary decompositions are meaningful as they exceed the default decompositions clearly.*

B. Experiment 2: Dependency Quality

Support and confidence are established metrics to measure the strength of evolutionary dependencies. This second experiment investigates whether stronger evolutionary dependencies are more efficient for software clustering. Furthermore, we want to examine the interplay between efficiency and density of evolutionary dependencies.

We call the edges of a dependency graph that connect classes of the same package *intra-edges*. They enable the clustering algorithm to retrieve the package structure. In contrast, edges that connect classes from different packages influence the clustering result negatively. Hence, the percentage of intra-edges among all edges of the graph provides a measure of the dependency efficiency for software clustering. This measure is independent of the total amount of available dependencies. Table IV lists the values of this intra-edge measure (first value) for the previously used set of graphs. Additionally, the data density is expressed as the percentage of nodes with at least one in- or outgoing edge (second value), which we denote as *node coverage*.

TABLE IV
PERCENTAGE OF PACKAGE INTRA-EDGES (FIRST VALUE) AND NODE COVERAGE (SECOND VALUE).

% values	*Azur.*	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*	**avg**
CIG	25;60	70;44	69;50	40;56	31;54	47;51	**47;53**
CAG	42;52	81;64	41;29	26;59	51;37	46;61	**48;50**
CUG	32;98	59;99	27;97	74;42	33;84	35;94	**43;86**
SCDG	31;99	59;100	29;98	39;81	33;84	35;94	**38;93**
$\mathrm{ECDG}^0_{0.0}$	16;54	23;65	40;66	37;68	14;48	30;15	**27;53**
$\mathrm{ECDG}^0_{0.2}$	32;54	36;64	47;66	39;68	16;48	31;15	**34;52**
$\mathrm{ECDG}^0_{0.4}$	39;51	43;64	54;64	41;65	16;48	35;14	**38;51**
$\mathrm{ECDG}^0_{0.6}$	45;45	51;62	67;57	43;64	19;43	37;14	**44;47**
$\mathrm{ECDG}^0_{0.8}$	47;39	53;60	63;56	44;64	16;40	37;14	**43;46**
$\mathrm{ECDG}^1_{0.0}$	19;40	29;49	78;36	41;49	26;31	48;06	**40;35**
$\mathrm{ECDG}^1_{0.2}$	36;39	43;49	83;36	45;49	27;30	46;06	**47;35**
$\mathrm{ECDG}^1_{0.4}$	45;34	54;48	89;34	40;49	27;29	49;06	**51;33**
$\mathrm{ECDG}^1_{0.6}$	45;30	65;45	94;32	43;49	24;28	56;04	**55;31**
$\mathrm{ECDG}^1_{0.8}$	51;21	83;41	99;22	46;42	20;22	71;03	**62;25**
$\mathrm{ECDG}^2_{0.0}$	20;32	33;39	85;16	37;42	24;13	67;03	**44;24**
$\mathrm{ECDG}^2_{0.2}$	36;31	47;39	87;15	40;42	24;13	65;03	**50;24**
$\mathrm{ECDG}^2_{0.4}$	44;27	59;38	87;14	37;42	29;13	61;02	**53;23**
$\mathrm{ECDG}^2_{0.6}$	45;21	76;34	98;12	39;42	33;12	69;02	**60;21**
$\mathrm{ECDG}^2_{0.8}$	49;13	90;31	100;05	40;35	20;09	82;02	**63;16**

1) Dependency Efficiency: The average values in the last column of Table IV show similar intra-edge ratios for the structural graphs, ranging from 38% to 48%, and more varying ratios for evolutionary graphs, ranging from 27% to 63%. The lower efficiency of totally unfiltered evolutionary dependencies explains their lower clustering quality in the previous experiment. With a stronger filter setup, the evolutionary dependencies, however, provide better dependency efficiencies. But this does not automatically imply better clustering results.

2) Dependency Density: Although we covered considerable development time spans of one year up to six years, the average node coverage rate of the evolutionary dependencies is 53% at most. In other words, on average there is no evolutionary information available for about half of the system. We had to ignored the initial check-in because it does not provide any reliable co-change information and the developers just did not change the files in the considered time span (1 to 6.5 years, see Table I). In contrast, the CUG and SCDG nearly cover the whole system; their average node coverage rate is 86% (CUG) and 93% (SCDG).

The low coverage rates of the evolutionary dependencies are clearly the main problem of a clustering approach exclusively based on this kind of data: Many parts of the system just do not get changed over years. Since the frequently changed parts of the system might be much more relevant in many software clustering applications, the evolutionary dependencies at least cover the potentially critical parts of the system. Nevertheless, structural dependencies obviously remain the first choice when complete coverage is important.

3) Filtering: The intra-edge percentages show that the dependency efficiency increases significantly with higher confidence values while the coverage rate only decreases slightly. In contrast, varying the support, just the first step (incrementing the support threshold from 0 to 1) enhances the efficiency. Further increments show no relevant positive effect, but decrease the coverage rate heavily. All in all, filtering by confidence seems to be more successful for the application of software clustering than filtering by support.

Result 2. *The dependency density, not the dependency efficiency, is the main problem of the evolutionary dependencies and explains their lower clustering quality. Nevertheless, it is important to filter the evolutionary data to exclude inefficient dependencies. Filtering by confidence works much better than filtering by support.*

C. Experiment 3: Combined Data Sources

Clustering exclusively based on evolutionary data was only partly successful because the data density is often too low for a complete clustering. Since the data quality of the filtered evolutionary dependencies is good, yet sparse, in this experiment we integrated the evolutionary dependencies into the dense structural data to improve the overall clustering results.

For this experiment we only considered a subset of the previously used graphs because the expected extra gain would not justify the extra effort of combining each of the four structural graphs with each of the 15 evolutionary graphs. SCDG will be the representative of the structural graphs while the following selection of evolutionary graphs represents the evolutionary data: the raw data ($\text{ECDG}^0_{0.0}$), the best setup in the first experiment ($\text{ECDG}^0_{0.4}$), two trade-offs between efficiency and coverage ($\text{ECDG}^0_{0.8}$ and $\text{ECDG}^1_{0.4}$), and a setup focusing on efficiency ($\text{ECDG}^1_{0.8}$).

1) Simple Union: A straightforward method to integrate structural and evolutionary dependencies is a union operation on graphs.

Definition 5. *Given two unweighted directed graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the* ***graph union operation*** $\cup$ *applied to G_1 and G_2 creates an unweighted directed graph*

$$G_1 \cup G_2 := G_3 = (V_3, E_3)$$

with $V_3 := V_1 \cup V_2$ and $E_3 := E_1 \cup E_2$ (here, $\cup$ denotes the normal set union operation).

Table V presents the new clustering results for the combined data sources and contrasts them to the results of the first experiment. Please recall the usage of the symbols ' and * as precision indicators (introduced in context of Table III).

TABLE V
MOJOFM CLUSTERING QUALITY BASED ON COMBINED STRUCTURAL AND EVOLUTIONARY DEPENDENCY GRAPHS USING THE UNION OPERATION.

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 2.2$	*Azur.*	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*	**avg**
SCDG	49.9'	65.5*	44.2	54.0	35.0	54.0'	**50.4**
$\text{ECDG}^0_{0.0}$	31.7	40.4	40.7	41.5	18.5	11.1	**30.7**
$\text{ECDG}^0_{0.4}$	35.6	47.0	43.2	42.3	20.7	11.3	**33.3**
$\text{ECDG}^0_{0.8}$	32.6	45.9	39.8	42.7	20.6	11.4	**32.2**
$\text{ECDG}^1_{0.4}$	31.3	39.6	28.2	35.2	17.5	7.5	**26.6**
$\text{ECDG}^1_{0.8}$	27.0	36.9	20.9	35.4	13.4	6.8	**23.4**
SCDG$\cup\text{ECDG}^0_{0.0}$	48.4'	54.8*	53.1	47.7	28.2	51.7'	**47.3**
SCDG$\cup\text{ECDG}^0_{0.4}$	50.6'	63.7*	55.9	52.4	31.9	56.1	**51.8**
SCDG$\cup\text{ECDG}^0_{0.8}$	52.7	68.7'	53.3*	55.1	34.4	55.9	**53.3**
SCDG$\cup\text{ECDG}^1_{0.4}$	52.3	64.6*	56.0	54.0	32.7	57.0	**52.8**
SCDG$\cup\text{ECDG}^1_{0.8}$	52.6	64.1*	50.5'	61.0	35.1	56.6	**53.3**

Combining the graphs with the simple union operation improves the average clustering quality from 50.4 (SCDG) to 53.3 (SCDG$\cup\text{ECDG}^0_{0.8}$ or SCDG$\cup\text{ECDG}^1_{0.8}$). Since we compare only six projects, a Friedman Test that compares the results of the SCDG to the four combined graphs integrating filtered evolutionary data does not rate the distributions as statistically significant.

In contrast to Experiment 1, a stronger filtering of evolutionary dependencies provides the best results (Experiment 1: $\text{ECDG}^0_{0.4}$; current experiment: SCDG$\cup\text{ECDG}^0_{0.8}$ or SCDG$\cup\text{ECDG}^1_{0.8}$). Thus, when combining data sources, efficiency seems to be more important than coverage. Integrating unfiltered evolutionary dependencies even decreases the clustering quality from 50.4 to 47.3.

Result 3. *It is possible to increase the quality of a clustering based on structural dependencies by integrating filtered evolutionary dependencies. A stronger filtering is beneficial; unfiltered dependencies might even decrease the overall quality.*

2) Weighted Union: Both structural as well as evolutionary dependency information may be flawed: A particular structural dependency may exist because a developer has misplaced a method. Similarly, a particular evolutionary dependency may exist because two classes were changed coincidentally at the same time. But if both dependencies link the same two classes, it is unlikely that this happens just by chance. An analysis of the dependency quality in terms of intra-edge ratios (compare to Experiment 2) shows the importance of this effect.

Table VI presents the intra-edge and node coverage results for duplicate dependencies—i.e., dependencies included in the intersection of both original graphs, which we define analogously to the union operation: $(V_1, E_1) \cap (V_2, E_2) := (V_1 \cap V_2, E_1 \cap E_2)$. The intra-edge ratio of the duplicate dependencies (48% to 66%) is clearly higher than the intra-edge ratio of the original dependencies (structural: 38%; evolutionary: 27% to 62%). Moreover, the intersection still includes a considerable number of dependencies (node coverage: 9% to 42%). Thus, we try to use this increased efficiency to improve the clustering results.

TABLE VI
PERCENTAGE OF PACKAGE INTRA-EDGES (FIRST VALUE) AND NODE COVERAGE (SECOND VALUE) FOR DUPLICATE DEPENDENCIES.

% values	*Azur.*	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*	**avg**
SCDG	31;99	59;100	29;98	39;81	33;84	35;94	**38;93**
$ECDG^0_{0.0}$	16;54	23;65	40;66	37;68	14;48	30;15	**27;53**
$ECDG^0_{0.4}$	39;51	43;64	54;64	41;65	16;48	35;14	**38;51**
$ECDG^0_{0.8}$	47;39	53;60	63;56	44;64	16;40	37;14	**43;46**
$ECDG^1_{0.4}$	45;34	54;48	89;34	40;49	27;29	49;06	**51;33**
$ECDG^1_{0.8}$	51;21	83;41	99;22	46;42	20;22	71;03	**62;25**
SCDG∩$ECDG^0_{0.0}$	37;50	55;63	59;45	49;36	42;44	45;11	**48;42**
SCDG∩$ECDG^0_{0.4}$	60;29	72;49	58;27	75;13	38;33	42;08	**57;27**
SCDG∩$ECDG^0_{0.8}$	63;15	87;40	56;11	50;08	46;17	40;05	**57;16**
SCDG∩$ECDG^1_{0.4}$	61;21	74;37	70;13	75;13	37;19	58;03	**63;18**
SCDG∩$ECDG^1_{0.8}$	58;09	93;28	75;03	50;08	20;06	100;01	**66;09**

After integrating both data sources with the union operation, the relevance of all dependencies is identical. To be able to exploit the more efficient duplicate dependencies, we introduce a dependency importance implemented as an edge weight. An extended union operation on graphs allows to assign a higher edge weight to duplicate dependencies.

Definition 6. *Given two unweighted directed graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the **weighted graph union operation** $\cup_{[\omega_a,\omega_b,\omega_c]}$ with $\omega_a, \omega_b, \omega_c \in \mathbb{R}$ applied to G_1 and G_2 creates a weighted directed graph*

$$G_1 \cup_{[\omega_a,\omega_b,\omega_c]} G_2 := (V_3, E_3, \mu)$$

where $(V_3, E_3) = G_1 \cup G_2$ and $\mu : E_3 \to \mathbb{R}$ is a weight function defined as

$$\mu(e) := \begin{cases} \omega_a & \text{if } e \in E_1 \wedge e \notin E_2 \\ \omega_b & \text{if } e \in E_1 \wedge e \in E_2 \\ \omega_c & \text{if } e \notin E_1 \wedge e \in E_2 \end{cases}$$

Note that this definition allows weights of 0, which will result in ignoring the according dependencies in the clustering process. A weighted union with weights of 1 for all three groups is equivalent to the simple union.

These weights influence the *Bunch* clustering tool, or more exactly, its internal quality metric. To emphasize the importance of the duplicate dependencies, their weight is set to 4 while the other edge weights stay 1. Table VII compares the clustering results based on the weighted union operation $\cup_{[1,4,1]}$ to the previous results.

TABLE VII
MOJOFM CLUSTERING QUALITY BASED ON COMBINED STRUCTURAL AND EVOLUTIONARY DEPENDENCY GRAPHS USING THE WEIGHTED UNION OPERATION $\cup_{[1,4,1]}$.

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 1.6$	*Azur.*	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*	**avg**
SCDG	49.9'	65.5*	44.2	54.0	35.0	54.0'	**50.4**
SCDG∪$ECDG^0_{0.0}$	48.4'	54.8*	53.1	47.7	28.2	51.7'	**47.3**
SCDG∪$ECDG^0_{0.4}$	50.6'	63.7*	55.9	52.4	31.9	56.1	**51.8**
SCDG∪$ECDG^0_{0.8}$	52.7	68.7'	53.3*	55.1	34.4	55.9	**53.3**
SCDG∪$ECDG^1_{0.4}$	52.3	64.6*	56.0	54.0	32.7	57.0	**52.8**
SCDG∪$ECDG^1_{0.8}$	52.6	64.1*	50.5'	61.0	35.1	56.6	**53.3**
SCDG∪$_{[1,4,1]}ECDG^0_{0.0}$	53.6'	64.7*	52.1*	48.8	35.1	52.6'	**51.2**
SCDG∪$_{[1,4,1]}ECDG^0_{0.4}$	52.8'	67.9*	54.8'	55.4	33.2	55.1'	**53.2**
SCDG∪$_{[1,4,1]}ECDG^0_{0.8}$	52.7'	71.1'	54.4'	57.8'	37.1	55.4	**54.8**
SCDG∪$_{[1,4,1]}ECDG^1_{0.4}$	53.1'	69.0*	54.8'	62.1	34.8	55.3'	**54.8**
SCDG∪$_{[1,4,1]}ECDG^1_{0.8}$	52.8	70.4'	50.0'	57.4'	34.9	56.2	**53.6**

The clustering qualities of the weighted combinations enhance slightly in comparison to the non-weighted combinations. This effect is highly significant ($p = 0.006$, Wilcoxon Test that compares all non-weighted union clustering results to the equivalent weighted union results). The best clustering result increases from 53.3 to 54.8. This extends the difference between the best combined strategy and the exclusively structural clustering (SCDG: 50.4; SCDG∪$ECDG^0_{0.8}$ or SCDG∪$ECDG^1_{0.8}$: 54.8). In contrast to the previous sub-experiment with simple graph union, in this case a Friedman Test between the clustering results of the SCDG and of the four graphs integrating filtered evolutionary dependency graphs rates the improvement as statistically significant ($p = 0.048$).

Result 4. *The efficiency of duplicate dependencies is better than the efficiency of general structural or evolutionary dependencies. The clustering results show that emphasizing these duplicate dependencies in the integration of structural and evolutionary data is able to improve the clustering further.*

3) Parameter Optimization: The weights in the previous experiment were derived from theoretical considerations. Nevertheless, better weighting setups may exist, which should be found by systematically varying the weights in a reasonable range. The following sub-experiment implements such a weight optimization by comparing the clustering qualities of differently weighted combined graphs.

To get results in due time, we decided to vary the weights in five steps, resulting in $5^3 = 125$ different weight setups. While the SCDG stands for the structural data, the $ECDG^0_{0.8}$, which has produced the best results in combination with the SCDG up to now, represents the evolutionary data. The search space is covered by the set of weights $\{0, 1, 2, 4, 8\}$ for each weight parameter.

Table VIII documents the parameter optimization experiment by a selection of the 14 best clustering results with respect to the average *MoJoFM* similarity.

The first and most important conclusion from the results is that the clustering quality cannot be improved in comparison to the weighted union $\cup_{[1,4,1]}$ ($\cup_{[2,8,2]}$ is equivalent and pro-

TABLE VIII
BEST MOJOFM CLUSTERING QUALITIES BASED ON THE COMBINED SCDG AND $\text{ECDG}^1_{0.4}$ USING THE WEIGHTED UNION OPERATION IN DIFFERENT SETUPS.

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 1.8$	*Azur.*	*JEdit*	*JFC*	*JFtp*	*JUnit*	*Tomc.*	**avg**
$\text{SCDG}\cup_{[2,8,2]}\text{ECDG}^0_{0.8}$	53.2	71.4'	54.0'	57.4'	37.3	55.9	**54.9**
$\text{SCDG}\cup_{[1,4,1]}\text{ECDG}^0_{0.8}$	52.7'	71.1'	54.4'	57.8'	37.1	55.4	**54.8**
$\text{SCDG}\cup_{[2,8,1]}\text{ECDG}^0_{0.8}$	53.4	69.2*	51.1'	60.3	35.9	55.4	**54.2**
$\text{SCDG}\cup_{[2,4,2]}\text{ECDG}^0_{0.8}$	52.3'	70.0'	54.9	56.3'	35.2	55.6	**54.1**
$\text{SCDG}\cup_{[2,2,1]}\text{ECDG}^0_{0.8}$	52.6	69.7'	52.4	58.1	36.2	55.2	**54.0**
$\text{SCDG}\cup_{[4,4,2]}\text{ECDG}^0_{0.8}$	53.0	69.7'	52.1	58.0	36.3	54.9	**54.0**
$\text{SCDG}\cup_{[8,8,4]}\text{ECDG}^0_{0.8}$	53.1	69.5'	52.5	58.5	36.0	54.6	**54.0**
$\text{SCDG}\cup_{[4,8,2]}\text{ECDG}^0_{0.8}$	52.6'	70.8'	51.1'	58.8	34.7	55.4	**53.9**
$\text{SCDG}\cup_{[4,8,4]}\text{ECDG}^0_{0.8}$	53.1	70.7	53.8'	55.9	34.9	54.8'	**53.9**
$\text{SCDG}\cup_{[1,2,1]}\text{ECDG}^0_{0.8}$	52.6	69.3*	53.1*	56.4	35.3	56.1	**53.8**
$\text{SCDG}\cup_{[1,8,1]}\text{ECDG}^0_{0.8}$	51.2*	70.7*	53.4*	54.3'	36.4	56.4	**53.7**
$\text{SCDG}\cup_{[2,1,1]}\text{ECDG}^0_{0.8}$	52.6	68.8'	52.5	57.9	35.6	54.8	**53.7**
$\text{SCDG}\cup_{[2,4,1]}\text{ECDG}^0_{0.8}$	51.3'	69.9*	50.6'	59.4	35.4	55.4	**53.7**
$\text{SCDG}\cup_{[4,2,2]}\text{ECDG}^0_{0.8}$	52.4	67.8'	52.0'	58.1	36.6	54.8	**53.6**
...							

duces nearly the same result). This confirms our preliminary considerations about emphasizing the duplicate dependencies. Nevertheless, one should not exaggerate their importance: $\cup_{[1,8,1]}$ is already rated lower.

Additionally, we analyzed the entire list of 125 setups and observed some further trends:

- Ignoring one of the dependency groups does not produce good clustering results. Setups that consider all groups (no weight is 0) reach an average quality of 52.7. Leaving out exclusively structural dependencies (only the first weight is 0) produces the worst results of 32.0. Since the other two groups are much smaller, omitting exclusively evolutionary or duplicate dependencies is not as dramatic (exclusively evolutionary: 51.2; duplicate: 52.2).
- Comparing all combinations that consider all dependency groups, setups where the weight of the duplicate dependencies is among the highest perform slightly better (53.2) than setups where it is among the lowest weights (52.6).
- The importance of the structural-only dependencies is slightly higher than the importance of the evolutionary-only dependencies. An average over all setups shows that setups with a higher structural than evolutionary weight produce a average quality of 53.1 while setups with a higher evolutionary than structural weight only produce a quality of 51.4. But as we observed in further experiments, stronger evolutionary filtering setups outweigh this effect.

Result 5. *Each of the dependency groups is valuable. The parameter optimization underlines again that duplicate dependencies are most important. The bests weight distribution of exclusively structural and evolutionary dependencies depends on the strength of the evolutionary filtering.*

IV. THREATS TO VALIDITY

By default, the results of a study are only valid for the examined subjects—here, for the sample software projects in a certain version. The current study is based on six different software projects that cover a wide range of application types, however restricted to *Java* projects and to less than 1000 classes. Nevertheless, the study is one of the most extensive studies in software clustering—only a few studies examine more software projects [15], [17] or larger projects [4], [25]. Due to the wide range of studied software projects, the findings are generalizable to some extent and might be a good indication for other *Java* projects and at least a weak indication for general software projects.

Improving the clustering results of *Bunch* does not imply by default that this is also possible for other clustering approach in the same way. But the example of *Bunch* indirectly shows the increased data quality for software clustering integrating structural and evolutionary data sources. Probably, other approaches are also able to use this data quality improvement to produce better clustering results. Similarly, the use case of architecture recovery limits the validity of the results. Nevertheless, it is plausible that other use cases of software clustering would also profit from the applied approach of integrating structural and evolutionary dependencies. Especially, applications where completeness is not a prime requirement are predestined for the usage of evolutionary dependencies.

Although, our goal was to design a completely fair study that does not bias the result in any direction, we cannot guarantee perfect fairness. Possibly, each step in the experimental design could bias the result. For instance, *MoJoFM* might favor the decomposition structure clustered from one of the data sources. Or *Bunch* may prefer certain dependency graphs. We tried to detect and eliminate such biases as far as possible.

Moreover, the study only showed that it is possible to improve the clustering results by a certain setup, but it cannot make any statements about to what degree the potential of the data sources is already used. Although we aimed to use a high quality clustering setup, it may be possible to get much better results in a different setup (e.g., with a different data integration method or other clustering parameters).

V. CONCLUSION

The study shows a positive impact of evolutionary data on software clustering. A clustering exclusively based on evolutionary dependencies, however, is only successful if substantial evolutionary data is available. Evolutionary dependencies often do not cover the set of classes sufficiently. This seems to be the main reason for the better performance of the aggregated structural dependencies.

Thus, when clustering a system by only taking evolutionary dependencies into account, it is most important to rely on extensive historical data that covers the essential parts of the system. The main advantages over a purely structural based clustering are that also non-source files can be considered and that the approach works independent of the programming language (in our study a light-weight parser was just used to identify classes).

An integration of the two data sources unites the advantages of the approaches at the cost of a more complex data

acquisition: In addition to a parsed system core, the clustering approach is still able to handle non-source files and non-parsed source files. The clustering quality increases in our experiments, especially when stressing duplicate dependencies. This confirms the assumption by Andritsos and Tzerpos [1] that integrating evolutionary may have a positive impact on the clustering result.

We filtered evolutionary dependencies successfully by confidence and support to increase efficiency of the dependencies. Thereby, filtering by confidence works better than filtering by support. A slight filtering turns out to be the best strategy when the clustering relies on evolutionary dependencies exclusively. A stronger filtering provides better results if evolutionary data is only an addition to structural data. Integrating unfiltered evolutionary dependencies bears the risk of decreasing the clustering quality.

These data-centered experiments demonstrate how important the choice and preprocessing of data sources in the domain of software clustering is. We believe that studying more data sources (e.g., dynamic dependencies, documentation, bug reports, software metrics), other preprocessing techniques, as well as different data combination strategies is essential for software clustering. Finally, the gained insights will help to tailor and customize software clustering techniques for particular applications like program comprehension, software (re)modularization, or software reuse.

REFERENCES

[1] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[2] N. Anquetil, C. Fourrier, and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 235–255.

[3] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy, "If your version control system could talk ..." in *ICSE '97 Workshop on Process Modeling and Empirical Studies of Software Engineering*. ACM Press, 1997.

[4] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. IEEE Computer Society, 2005, pp. 259–268.

[5] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003.

[6] J. Gargiulo and S. Mancoridis, "Gadget: A tool for extracting the dynamic structure of Java programs," in *SEKE '01: Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering*, 2001, pp. 244–251.

[7] S. Kim, T. Zimmermann, J. E. Whitehead, and A. Zeller, "Predicting faults from cached history," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 489–498.

[8] R. Koschke and T. Eisenbarth, "A framework for experimental evaluation of clustering techniques," in *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 201–210.

[9] A. Kuhn, S. Ducasse, and T. Girba, "Enriching reverse engineering with semantic clustering," in *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 133–142.

[10] K.-L. Ma, "Stargate: A unified, interactive visualization of software projects," in *PacificVis '08: Proceedings of the IEEE VGTC Pacific Visualization Symposium 2008*. IEEE Computer Society, 2008, pp. 191–198.

[11] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 800–813, 1991.

[12] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 50–59.

[13] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 45–52.

[14] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[15] B. S. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *ICSM '01: Proceedings of the 17th IEEE International Conference on Software Maintenance*, 2001, pp. 744–753.

[16] B. S. Mitchell, "A heuristic approach to solving the software clustering problem," Ph.D. dissertation, Drexel University, 2002.

[17] B. S. Mitchell and S. Mancoridis, "On the evaluation of the Bunch search-based software modularization algorithm," *Soft Computing*, vol. 12, no. 1, pp. 77–93, 2007.

[18] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[19] V. Tzerpos and R. C. Holt, "MoJo: A distance metric for software clusterings," in *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 187–193.

[20] A. Vanya, L. Hofland, S. Klusener, P. van de Laar, and H. van Vliet, "Assessing software archives with evolutionary clusters," in *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 192–201.

[21] L. Voinea and A. Telea, "CVSgrab: Mining the history of large software projects," in *EuroVis '06: Joint Eurographics - IEEE VGTC Symposium on Visualization*. Eurographics Association, 2006, pp. 187–194.

[22] P. Weißgerber, "Automatic refactoring detection in version archives," Ph.D. dissertation, Univerisity of Trier, 2009.

[23] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *IWPC '04: Proceedings of the 12th International Workshop on Program Comprehension*. IEEE Computer Society, 2004, pp. 194–203.

[24] A. Wierda, E. Dortmans, and L. L. Somers, "Using version information in architectural clustering - a case study," in *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 214–228.

[25] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 525–535.

[26] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *CSMR '05: Proceedings of the 9th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 124–133.

[27] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 563–572.

[28] T. Zimmermann and P. Weißgerber, "Preprocessing CVS data for fine-grained analysis," in *MSR '04: Proceedings of the 1st International Workshop on Mining Software Repositories*. IEEE Computer Society, 2004, pp. 2–6.

[29] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003.