Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 792024, 18 pages
doi:10.1155/2012/792024

Review Article

Clustering Methodologies for Software Engineering

Mark Shtern and Vassilios Tzerpos

Department of Computer Science and Engineering, York University, Toronto, ON, Canada M3J 1P3

Correspondence should be addressed to Mark Shtern, mark@cse.yorku.ca

Received 25 November 2011; Revised 28 February 2012; Accepted 28 February 2012

Academic Editor: Letha Hughes Etzkorn

The size and complexity of industrial strength software systems are constantly increasing. This means that the task of managing a large software project is becoming even more challenging, especially in light of high turnover of experienced personnel. Software clustering approaches can help with the task of understanding large, complex software systems by automatically decomposing them into smaller, easier-to-manage subsystems. The main objective of this paper is to identify important research directions in the area of software clustering that require further attention in order to develop more effective and efficient clustering methodologies for software engineering. To that end, we first present the state of the art in software clustering research. We discuss the clustering methods that have received the most attention from the research community and outline their strengths and weaknesses. Our paper describes each phase of a clustering algorithm separately. We also present the most important approaches for evaluating the effectiveness of software clustering.

1. Introduction

Software clustering methodologies group entities of a software system, such as classes or source files, into meaningful subsystems in order to help with the process of understanding the high-level structure of a large and complex software system. A software clustering approach that is successful in accomplishing this task can have significant practical value for software engineers, particularly those working on legacy systems with obsolete or nonexistent documentation.

Research in software clustering has been actively carried out for more than twenty years. During this time, several software clustering algorithms have been published in the literature [1–8]. Most of these algorithms have been applied to particular software systems with considerable success.

There is consensus between software clustering researchers that a software clustering approach can never hope to cluster a software system as well as an expert who is knowledgeable about the system [9]. Therefore, it is important to understand how good a solution created by a software clustering algorithm is. The research community has developed several methods to assess the quality of software clustering algorithms [10–14].

In this paper, we present a review of the most important software clustering methodologies that have been presented in the literature. We also outline directions for further research in software clustering, such as the development of better software clustering algorithms or the improvement and evaluation of existing ones.

The structure of the paper is as follows. Section 2 presents three different applications of existing software clustering techniques in order to motivate the need for software clustering. An overview of the state of the art for software clustering algorithms is presented in Section 3. Section 4 classifies and presents the most prominent approaches for the evaluation of software clustering algorithms. Open research challenges related to software clustering are discussed in Section 5. Finally, Section 6 concludes the paper.

2. Software Clustering Applications

Before we present the technical aspects of software clustering in Section 3, we describe three distinct instances where existing software clustering techniques were used to solve important software engineering problems in the context

of reflexion analysis, software evolution, and information recovery.

2.1. Reflexion Analysis. The goal of reflexion analysis is to map components found in the source code onto the conceptual components defined in a hypothesized architecture. This mapping was originally established manually, which required a lot of work for large software systems [15]. Christl et al. [16] presented an approach, in which clustering techniques were applied to support the user in the mapping activity. They developed a semiautomated mapping technique that accommodates the automatic clustering of the source model with the user's hypothesized knowledge about the system's architecture. The role of software clustering was to identify concrete entities for which a mapping decision is "easy enough" to be made automatically. In addition, software clustering supports the user in manual mapping by detecting hypothesized entities that are likely to be the correct entity.

The authors have developed an application called HuGMe based on their concept. HuGMe has been applied successfully to extend partial maps of real-world software applications.

2.2. Software Evolution. Software systems evolve through efforts to add new functionality, to correct existing faults, and to improve maintainability. Typically, software clustering tools attempt to improve the software structure (software restructuring) or to reduce the complexity of large modules (source code decoupling). Sometimes, software clustering can be used to help identify duplicate code [17]. In addition, software clustering is used to predict the fault proneness of software modules [18].

Software restructuring is a form of perfective maintenance that modifies the structure of a program's source code. Its goal is increased maintainability to better facilitate other maintenance activities, such as adding new functionality to or correcting previously undetected errors within a software system.

Lung et al. [19] presented a case study whose goal was to decouple two subsystems in order to reduce development time of new functionality. The goal of software clustering was to identify related components in the software system. The authors compared the produced software clustering output with the software design and detected a gap between the conceptual model and the actual design. After detailed investigation of the gap between both views, a problem with the conceptual model was identified. A new design was developed. The new design significantly improved the coupling of the software system.

The motivation of source code decoupling is to help reduce the complexity of complex modules or functions. The complexity of a module or a function is determined based on software metrics. Xu et al. [17] have presented a case study where software clustering is applied for source code decoupling at the procedure level. Software clustering attempts to group related statements together to produce a dependency rank between the groups. The authors suggest to divide a module or a function according to the result of software clustering. To help reverse engineers, the authors developed a user interface that displays the output of software clustering and the existing structure of the function being analyzed. This visualization tool allows easy identification of ill-structured and duplicated code.

2.3. Information Recovery. The primary goal of reverse engineering is to recover components or to extract system abstractions. Several approaches and techniques, which are based on software clustering, have been proposed in the literature to support information recovery from a software system.

We present two case studies that focus on module and architecture recovery. We select module and architecture recovery because these methods focus on different abstraction levels. We consider module recovery a lower level of abstraction than architecture recovery.

Module recovery software clustering methods focus on discovering modules by analyzing dependencies extracted from the software system, such as function calls. A typical example of module recovery is Robert Schwanke's Arch tool that helps a software engineer to understand a software system by producing a decomposition of the system into subsystems [20]. Schwanke's clustering heuristic is based on the principle of maximizing the cohesion of procedures placed in the same module, while, at the same time, minimizing the coupling between procedures that reside in different modules.

The architecture recovery methods focus on discovering the system architecture by analyzing abstractions extracted from the source code, such as components (modules), subsystems, and design patterns. A typical example of architecture recovery is that presented by Bauer and Trifu [21]. They recover architecture based on clustering information about design patterns. Design patterns are used to identify architectural clues—small structural patterns that provide information to allow for a rating of the dependencies found between entities. These clues are used by software clustering to produce the final system decomposition.

As can be seen from the above examples of applying software clustering, it is an important technique that can be used to solve important problems.

3. Software Clustering

We now present the state of the art of software clustering research. We do so in the context of the more general framework of cluster analysis.

Cluster analysis is a group of multivariate techniques whose primary purpose is to group entities based on their attributes. Entities are classified according to predetermined selection criteria, so that similar objects are placed in the same cluster. The objective of any clustering algorithm is to sort entities into groups, so that the variation between clusters is maximized relative to variation within clusters.

The typical stages of cluster analysis techniques are as follows:

(1) fact extraction (Section 3.1),

(2) filtering (Section 3.2),

(3) similarity computation (Section 3.3),

(4) cluster creation (Section 3.4),

(5) results visualization (Section 3.5),

(6) user feedback collection (Section 3.6).

The process typically repeats until satisfactory results are obtained.

We discuss each stage in detail in the following.

3.1. Fact Extraction. Before applying clustering to a software system, the set of entities to cluster needs to be identified. Entity selection depends on the objective of the method. For example, for program restructuring at a fine-grained level, function call statements are chosen as entities [17]. When the software clustering method applies to design recovery problems [22–25], the entities are often software modules. Classes [21] or routines [26] can also be chosen as the entities.

After entities have been identified, the next phase is attribute selection. An attribute is usually a software artefact, such as a package, a file, a function, a line of code, a database query, a piece of documentation, or a test case. Attributes may also be high-level concepts that encompass software artefacts, such as a design pattern. An entity may have many attributes. Selecting an appropriate set of attributes for a given clustering task is crucial for its success.

Most often, software artefacts are extracted directly from the source code, but sometimes artefacts are derived based on other kinds of information, such as binary modules, software documentation. Most studies extract artefacts from various sources of input and store them in a language-independent model [27]. These models can then be examined and manipulated to study the architecture of the software being built. For instance, FAMIX [28] is used to reverse engineer object-oriented applications; its models include information about classes, methods, calls, and accesses. We often refer to these models as *factbases*.

The Tuple Attribute Language (TA) is a well-known format for recording, manipulating, and diagramming information that describes the structure of large systems. The TA information includes nodes and edges in the graph and attributes of these nodes and edges [29]. It is also capable of representing typed graphs (graphs that can have more than one type of edge). Therefore, the TA language is not limited to recording only the structure of large systems. It can also express facts that are gathered from other sources.

The Dagstuhl Middle metamodel [30], GXL [31], MDG [32], and RSF [33] are other examples of factbase formats.

The Knowledge Discovery Metamodel (KDM) [34] from the Object Management Group (OMG) can also be used for this purpose. KDM is a common intermediate representation for existing software systems and their operating environments, that defines common meta-data required for deep semantic integration of Application Lifecycle Management tools. It can also be viewed as an ontology for describing the key aspects of knowledge related to the various facets of enterprise software.

In the following, we present the most common inputs for the artefact extraction process.

3.1.1. Source Code. Source code is the most popular input for fact extraction. Many researchers [21, 32, 35] are using the source code as the only trusted foundation for uncovering lost information about a software system.

There are two conceptual approaches to extracting facts from source code: syntactic and semantic. The syntactic (structure-based) approaches focus on the static relationships among entities. The exported facts include variable and class references, procedure calls, use of packages, association and inheritance relationships among classes.

Semantic approaches [36] include all aspects of a system's domain knowledge. The domain knowledge information present in the source code is extracted from comments, identifier names [37].

Syntactic approaches can be applied to any software system, whereas semantic approaches usually need to be customized to a specific software system due to domain specific assumptions, since two terms may be related in one domain knowledge and unrelated in another. For instance, stocks and options are related in a financial application and unrelated in a medical application. Knowledge spreading in the source code and absence of a robust semantic theory are other drawbacks of semantic approaches. However, the output from semantic approaches tends to be more meaningful than the one from syntactic approaches.

The software clustering community widely adopts structure-based approaches for large systems. In many cases the boundary between these approaches is not strict. Some clustering methodologies try to combine the strengths of both syntactic and semantic methods. The ACDC algorithm is one example of this mixed approach [9].

3.1.2. Binary Code. Some approaches work with the information available in binary modules. Depending on compilation and linkage parameters, the binary code may contain information, such as a symbol table that allows efficient fact extraction. This approach has three advantages.

(1) It is language independent.

(2) Binary modules are the most accurate and reliable information (source code may have been lost or mismatched to a product version of binary modules. Source mismatch situations occur because of human mistakes, patches, and intermediate/unreleased versions that are working in the production environment, etc.).

(3) Module dependency information is easy to extract from binary modules (linkage information contains module dependency relations).

The main drawbacks of this approach are that binary metadata information depends on building parameters

and that the implementation of the approach is compiler/hardware dependent. Also, binary code analysis cannot always discover all relationships. In particular, the Java compiler erases type parameter information and resolvs references to final static fields of primitive types (constants) [38].

The binary code analysis method has been explored by the SWAG group [39]. SWAG has developed a fact extractor for Java called Javex. The output of the Java compiler is p-code. Javex is capable of extracting facts from the p-code and storing the facts using the TA format. Other researchers extracting information from bytecode are Lindig and Snelting [4] and Korn et al. [40]. An interesting concept of collecting information about a software system is presented by Huang et al. [41]. Their approach collects facts from components. Components developed for frameworks such as CORBA/CCM, J2EE/EJB, COM+. include interface information that can be easily extracted and explored.

Unfortunately, extraction tools based on source or binary code information cannot extract all facts about software systems. For instance, they cannot extract facts related to run-time characteristics of the software system. In addition, configuration information of component-based software systems, that are implemented and executed with the help of some common middleware (e.g., J2EE, COM+, ASP.NET), is unavailable because it is stored in middleware configuration files that are not part of the source or binary code of the software system. This configuration information is important because it includes declaration of required resources, security realm and roles, and component names for runtime binding. Therefore, while the source and binary code contains important information about a software system, it does not contain the complete information about the system.

3.1.3. Dynamic Information. Static information is often insufficient for recovering lost knowledge since it only provides limited insight into the runtime nature of the analyzed software; to understand behavioural system properties, dynamic information is more relevant [42]. Some information recovery approaches use dynamic information alone [43, 44], while others mix static and dynamic knowledge [8, 42, 45].

During the run-time of a software system, dynamic information is collected. The collected information may include the following.

(1) Object construction and destruction.

(2) Exceptions/errors.

(3) Method entry and exit.

(4) Component interface invocation.

(5) Dynamic type information.

(6) Dynamic component names.

(7) Performance counters and statistics:

   (a) number of threads,
   (b) size of buffers,
   (c) number of network connections,
   (d) CPU and memory usage,
   (e) number of component instances,
   (f) average, maximum, and minimum response time.

There are various ways of collecting dynamic information, such as instrumentation methods or third-party tools (debuggers, performance monitors). Instrumentation techniques are based on introducing new pieces of code in many places to detect and log all collected events. Such techniques are language dependent and not trivial to apply. The biggest concern with these techniques is ensuring that the newly generated software system has the same run-time behaviour as the original one. One option for implementing this approach is based on the Java Probekit [46]. Probekit is a framework on the Eclipse Platform that you can use to write and use probes. Probes are Java code fragments that can be inserted into a program to provide information about the program as it runs. These probes can be used to collect run-time events needed for dynamic analysis. An alternative way to collect run-time information is to use debugger-based solutions [43, 47, 48]. One advantage of using debuggers is that the source code remains untouched. Unfortunately, debugger information may not be sufficient to record the same aspects of the software system as instrumentation techniques. Also, some of the compilers cannot generate correct debug information when source code optimizations are enabled.

Today, managed runtime environments such as .NET and J2EE are popular paradigms. A virtual runtime environment allows gathering run-time information without modification of the source code. For instance, VTune analyzer [49] uses an industry standard interface in the JVM, the JVMPI (JVM Profiling Interface), for gathering Java-specific information. This interface communicates data regarding memory locations and method names of JIT (just-in-time) emitted code, calls between Java methods, symbol information, and so forth.

Performance monitors allow the collection of statistical information about software systems such as CPU, memory usage, and size of buffers. That information may uncover interesting behaviour relationships between software components.

Unfortunately, neither dynamic nor static information contains the whole picture of a software system. For instance, design information and historical information are not part of any input discussed so far.

3.1.4. Physical Organization. The physical organization of applications in terms of files, folders, packages, and so forth often represents valuable information for system understanding [50]. Physical organization is not limited to the software development structure. It may also include the deployment structure and build structure. The deployment structure often follows industrial standards. Therefore, the location of a specific module provides valuable information about its responsibilities.

It is important to consider the physical organization of the software system because it often reflects the main ideas of the system design.

3.1.5. Human Organization. Human organization often reflects the structure of the system. Usually a developer is responsible for associated components. According to Conway, "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" [51].

3.1.6. Historical Information. Historical information explains the evolution of a software product. Recently, more research [52–54] using historical information to reveal software design has appeared. Historical information is collected from version management systems, bug tracking systems, release notes, emails, and so forth. Software evolution contains valuable information for the solution of the software understanding problem [55]. For example, release notes contain a lot of valuable knowledge about product features and product releases.

Unfortunately, it is usually difficult to automatically/semiautomatically recover important system knowledge from historical sources due to the size and the lack of formatting of the extracted information.

3.1.7. Software Documentation. Software documents contain a lot of helpful information about software systems. However, they cannot be entirely trusted since they are often outdated or unsynchronized [9, 56]. Facts extracted from software documents may not reflect the current state of the system. Therefore, the extracted facts should be validated with the current system. Hassan and Holt [57] present such an approach. The idea of the method is to collect information about a software system from existing documentation and domain knowledge. The gathered information is then verified against the current stage of the software implementation.

3.1.8. Persistent Storage. Persistent repositories, such as databases, and output files, contain information that can be helpful for software understanding. Developers often attempt to understand a software system by analyzing the application repositories. Software clustering methods should be able to utilize this information to their advantage as well [58].

3.1.9. Human Expertise. Humans may provide valuable facts based on their knowledge of requirement documents, high-level design, and other sources.

Every input source has different advantages and disadvantages. Even though the end result of the clustering process will likely improve if the input factbase contains information from different sources [59], the mixing of information from various sources is a challenging problem [23].

After the extraction process is finished, a filtering step may take place to ensure that irrelevant facts are removed, and the gathered facts are prepared for the clustering algorithm.

3.2. Filtering. The filter phase is the final stage of preparing a factbase. The main goal of this stage is to discard unnecessary information, calculate facts that are a composition of existing facts, and apply a weighting scheme to the attributes.

For instance, all meaningless words extracted from source comments are discarded during the filter step [37]. In an ideal situation, the factbase should be small and consist of enough information to ensure meaningful clustering.

The information contained in the final factbase may be dependent on assumptions of a specific software clustering algorithm. Some algorithms expect that the factbase includes only relations between modules [25]; other algorithms do not make any assumptions about the facts [24].

The software research community most often applies the following filters:

(1) utility module processing,

(2) granularity projection.

These filters process the collected factbase and discard unnecessary information. The study presented in [60] shows that using a different methodology for the construction of the final factbase may affect results significantly.

This discovery emphasizes the importance of preparing the final factbase. Several tools have been developed that allow fact manipulation. For instance, Grok [29] is specifically developed for the manipulation of facts extracted from a software system. Other methods utilize SQL [61] or Prolog [21] for fact manipulations.

3.2.1. Utility Module Processing. In many cases, modules containing utilities do not follow common design practices, such as high cohesion and low coupling. For example, utility modules may include drivers, commonly used methods. As a result, utilities may require special treatment. Some software clustering algorithms are not affected by utilities [24]; others, such as Bunch [62], may be affected [56]. Some authors have argued that removing utilities improves the overall results [1]. Others suggest considering utilities as a class that should be kept for further investigation, because it plays an important role in the implementation of the overall solution by allowing communication between other classes [56]. Therefore, the research community has not reached a conclusion about the best approach to dealing with utilities.

In cases where the software clustering method is affected by utilities, a utility filter should be applied. Such a filter identifies utilities based on the facts present in the factbase. There are different ways to do this. Hamou-Lhadj et al. [63] identified utilities as classes that have many direct client classes. Wen and Tzerpos [64] present a utility module detection method where a module is identified as a utility if it is connected to a large number of subsystems (clusters) rather than entities.

3.2.2. Granularity Projection. A large number of software clustering algorithms attempt to cluster only course-grained software entities, such as modules or classes, rather than more fine-grained ones, such as functions or variables. However, facts extracted from software systems often contain

dependencies only between the fine-grained software entities.

The goal of a granularity projection filter is to use such low-level facts, such as function calls or variable references, in order to calculate dependencies between classes, and then remove the low-level facts from the factbase in order to retain facts only about the entities to be clustered. For instance, only classes and their dependencies might remain in the factbase after such a filter has been applied.

After the final factbase is constructed, the next step is to compute similarities.

3.3. Similarity Computation. Most software clustering methods initially transform a factbase to a data table, where each row describes one entity to be clustered. Each column contains the value for a specific attribute. Table 1 presents an example of a data table. It contains information about file-to-file relationships, where each entity and each attribute are files of the software system. The values of the attributes are calculated based on the dependencies between the files. In this example, file $f1.c$ depends on file $f3.c$ (possibly by calling a function in $f3.c$), while file $f3.c$ does not depend on $f1.c$.

In most cases, a different column corresponds to a different attribute. Sometimes, different columns contain information about related attributes. For example, categorical attributes (a categorical attribute is an attribute with a finite number of values (in practice, a *small* number of *discrete* values, such as developer names) and no inherent ranking) are often represented as a composition of multiple binary attributes [24]. Table 2 is an example of such a situation. It is an extension of Table 1 with a new categorical attribute representing developer names.

Categorical data can be represented in compact form as well. Table 3 is an example of compact form representation of the same data as in Table 2.

Clustering algorithms are based on a similarity function between entities [65]. However, some algorithms, such as hierarchical agglomerative ones, are applying the similarity function explicitly, while others, such as search-based algorithms, are using the similarity function only implicitly.

The most common type of similarity functions is resemblance coefficients. Other similarity functions include probabilistic measures and software-specific similarities.

3.3.1. Resemblance Coefficients. The input data matrix for a resemblance coefficient may contain different types of data, such as binary, numerical, categorical, or mixed. In the following, we discuss the most well-known resemblance coefficients developed for each one of these types of data.

Binary Resemblance. The intuition behind the calculation of resemblance coefficients is to measure the amount of relevant matches between two entities. In other words, the more relevant matches there are between two entities, the more similar the two entities are. There are different methods for counting relevant matches, and many formulas exist to calculate resemblance coefficients [66–68]. Some well-known examples are given in Table 4. In these formulas, a represents the number of attributes that are "1" in both entities, b and c represent the number of attributes that are "1" in one entity and "0" in the other, and d represents the number of attributes that are "0" in both entities.

Table 1: Data table example.

	$f1.c$	$f2.c$	$f3.c$
$f1.c$	1	1	1
$f2.c$	0	1	0
$f3.c$	0	0	1

Table 2: Categorical attribute presented as binary data.

	$f1.c$	$f2.c$	$f3.c$	Alice	Bob
$f1.c$	1	1	1	1	0
$f2.c$	0	1	0	0	1
$f3.c$	0	0	1	1	0

Table 3: Categorical attribute in compact form.

	$f1.c$	$f2.c$	$f3.c$	Developer
$f1.c$	1	1	1	Alice
$f2.c$	0	1	0	Bob
$f3.c$	0	0	1	Alice

Table 4: Examples of binary resemblance coefficients.

Similarity measure	Formula
Simple matching coefficient	$\frac{a+d}{a+b+c+d}$
Jaccard coefficient	$\frac{a}{a+b+c}$
Sorenson coefficient	$\frac{2a}{2a+b+c}$
Rogers and Tanimoto	$\frac{a+d}{a+2(b+c)+d}$
Russel and Rao	$\frac{a}{a+b+c+d}$

A binary resemblance coefficient that is suitable for software clustering will ideally include the following two properties.

(1) 0-0 matches are ignored; that is, d is not part of the formula. The joint lack of attributes between two entities should not be counted toward their similarity [60].

(2) Heavier weight is assigned to more important factors [69].

A binary resemblance coefficient that fits these software clustering assumptions is the Sorenson coefficient [60]. Research [70] concludes that Jaccard and Sorenson have performed well, but the authors recommend using the Jaccard algorithm because it is more intuitive.

Categorical Resemblance. There are similarities between binary and categorical resemblance coefficients. The calculation of a categorical resemblance coefficient, similar to that of a binary resemblance coefficient, is based on the number of matches between two entities. When categorical attributes are represented as a set of binary attributes, then the calculation of the categorical coefficient is based on the calculation of the binary resemblance coefficient. When categorical attributes are represented in compact form, then the categorical coefficient is calculated based on the simple matching formula (See Table 4).

Numerical Resemblance. Numerical resemblance coefficients calculate distance between entities. Each entity is represented as a vector. For instance, entity $f1.c$ from Table 1 can be represented as vector $(1, 1, 1)$. Its distance to other vectors can be calculated using formulas such as

(1) Euclidean: $\sqrt{\sum_{i=1}^{n}((x_i - y_i)^2)}$,

(2) Maximum: $\max |x_i - y_i|$,

(3) Manhattan: $\sum_{i=1}^{n}(|x_i - y_i|)$.

Mixed Resemblance. An entity in the data table may be described by more than one type of attributes. At the same time, some values in the data table may be missing. For those cases, the widely used general similarity coefficient was developed by Gower [71]. Let x and y denote two entities and describe over d attributes. Then, the general similarity coefficient $S_{\text{Gower}}(x, y)$ is defined as

$$S_{\text{Gower}}(x, y) = \frac{1}{\sum_{k=1}^{d} w(x_k, y_k)} \sum_{k=1}^{d} w(x_k, y_k) s(x_k, y_k), \quad (1)$$

where $s(x_k, y_k)$ is a similarity component for the kth attribute and $w(x_k, y_k)$ is either one or zero, depending on whether or not a comparison is valid for the kth attribute of the entities.

3.3.2. Probabilistic Measures. Probabilistic measures are based on the idea that agreement on rare matches contributes more to the similarity between two entities than agreement on more frequent ones [72]. The probabilistic coefficients require the distribution of the frequencies of the attributes present over the set of entities. When this distribution is known, a measure of information or entropy can be computed for each attribute. Entropy is a measure of disorder; the smaller the increase in entropy when two (sets of) entities are combined is, the more similar the two entities are. For a more detailed discussion on probabilistic coefficients, we refer to [73].

3.3.3. Software-Specific Similarity. There are also similarity functions that have been developed specifically for the software clustering problem. Schwanke [74] introduced the notion of using design principles, such as low coupling and high cohesion. Koschke [75] has developed an extension of Schwanke's metric-based hierarchical clustering technique. The Koschke similarity functions include global declarations, function calls. Also, the similarity method is considering name similarities between identifiers and filenames. Choi and Scacchi [2] also describe a similarity function based on maximizing the cohesiveness of clusters.

3.4. Cluster Creation. At this point, all preparation steps are completed, and the clustering algorithm can start to execute. In this section, we discuss various software clustering algorithms. Wiggerts [72] suggests the following classification of software clustering algorithms:

(1) graph-theoretical algorithms,

(2) construction algorithms,

(3) optimization algorithms,

(4) hierarchical algorithms.

3.4.1. Graph-Theoretical Algorithms. This class of algorithms is based on graph properties. The nodes of such graphs represent entities, and the edges represent relations. The main idea is to identify interesting subgraphs that will be used as basis for the clusters. Types of subgraphs that can be used for this purpose include connected components, cliques, and spanning trees. The two most common types of graph-theoretical clustering algorithms are aggregation algorithms and minimal spanning tree algorithms.

Aggregation algorithms reduce the number of nodes (representing entities) in a graph by merging them into aggregate nodes. The aggregates can be used as clusters or can be the input for a new iteration resulting in higher-level aggregates.

Common graph reduction techniques are the notion of the neighbourhood of a node [76], strongly connected components [77], and bicomponents [77].

Minimal spanning tree (MST) algorithms begin by finding an MST of the given graph. Next, they either interactively join the two closest nodes into a cluster or split the graph into clusters by removing "long" edges. The classic MST algorithm is not suited for software clustering due to the fact that the algorithm tends to create a few large clusters that contain many entities while several other entities remain separate [78]. Bauer and Trifu [21] suggest a two-pass modified MST algorithm. The first pass, which follows the classic MST concept, iteratively joins the two closest nodes into a cluster while the second pass assigns the remaining unclustered entities to the cluster they are the "closest" to.

3.4.2. Construction Algorithms. The algorithms in this category assign the entities to clusters in one pass. The clusters may be predefined (supervised) or constructed as part of the assignment process (unsupervised). Examples of construction algorithms include the so-called geographic techniques and the density search techniques. A well-known geographic technique is the bisection algorithm, which at each step divides the plain in two and assigns each entity according to the side that it lies on.

An algorithm based on fuzzy sets was presented in [79]. An ordering is defined on entities determined by their grade

of membership (defined by the characteristic function of the fuzzy set). Following this order, each entity is either assigned to the last initiated cluster or it is used to initiate a new cluster, depending on the distance to the entity which was used to initiate the last initiated cluster.

Mode analysis [80] is another example of a construction clustering algorithm. For each entity, it computes the number of neighbouring entities that are "closer" than a given radius. If this number is large enough, then the algorithm clusters the entities together.

3.4.3. Optimization Algorithms. An optimization or improvement algorithm takes an initial solution and tries to improve this solution by iterative adaptations according to some heuristic. The optimization method has been used to produce both hierarchical [81] and nonhierarchical [82] clustering.

A typical nonhierarchical clustering optimization method starts with an initial partition derived based on some heuristic. Then, entities are moved to other clusters in order to improve the partition according to some criteria. This relocating goes on until no further improvement of this criterion takes place. Examples of clustering optimization methods are presented in [83, 84].

One of the famous representatives of the optimization class of algorithms is ISODATA [85]. Its effectiveness is based on the successful initial choice of values for seven parameters that control factors such as the number of expected clusters, the minimum number of objects in the cluster, and the maximum number of iterations. The algorithm then proceeds to iteratively improve on an initial partition by joining and splitting clusters, depending on how close to the chosen parameters the actual values for the current partition are.

Other optimization algorithms can be classified in four categories presented in detail below: genetic, hill-climbing, spectral, and clumping techniques.

Genetic clustering algorithms are randomized search and optimization techniques guided by the principles of evolution and natural genetics, having a large amount of implicit parallelism. Genetic algorithms are characterized by attributes, such as the objective function, the encoding of the input data, the genetic operators, such as crossover and mutation, and population size.

A typical genetic algorithm runs as follows.

(1) Select a random population of partitions.

(2) Generate a new population by selecting the best individuals according to the objective function and reproducing new ones by using the genetic operations.

(3) Repeat step (2) until a chosen stop criterion is satisfied.

Doval et al. [86] present a schema for mapping the software clustering problem to a genetic problem. The quality of a partition is determined by calculating a modularization quality function. There are several ways for the calculation of modularization quality. In general, the modularization quality measures the cohesion of clusters and their coupling. The result of the algorithm is a set of clusters for which an optimal modularization quality was detected, in other words, clusters that feature an optimal tradeoff between coupling and cohesion. The Doval et al. genetic algorithm has been implemented as part of the Bunch software clustering tool [87].

Shokoufandeh et al. claim that the genetic Bunch algorithm is especially good at finding a solution quickly, but they found that the quality of the results produced by Bunch's hill-climbing algorithms is typically better [88].

Seng et al. proposed an improved software clustering genetic algorithm [89] based on Falkenauer's class of genetic algorithms [90]. According to the author, the algorithm is more stable than Bunch, and its objective function evaluates additional quality properties of the system's decomposition, such as individual subsystem size.

Hill-Climbing clustering algorithms perform the following steps.

(1) Generate a random solution.

(2) Explore the neighbourhood of the solution attempting to find a neighbour better than the solution. Once a neighbour is found, it becomes the solution.

(3) Repeat step (2) until there is no better neighbour.

Hill-climbing search methods have been successfully employed in various software clustering algorithms [83]. Mitchell's Ph.D. dissertation [32] shows promising results in terms of the quality and performance of hill-climbing search methods. His approach has been implemented as part of the Bunch software clustering tool [91, 92].

Bunch starts by generating a random partition of the module dependency graph. Then, entities from the partition are regrouped systematically by examining neighbouring partitions in order to find a better partition. When an improved partition is found, the process repeats; that is, the found partition is used as the basis for finding the next improved partition. The algorithm stops when it cannot find a better partition. The objective function is the modularization quality function used also in Bunch's genetic algorithm. Mahdavi et al. spresent an improvement to existing hill-climbing search approaches based on applying a hill-climbing algorithm multiple times. The proposed approach is called multiple hill climbing [22]. In this approach, an initial set of hill-climbing searches is performed. The created partitions are used to identify the common features of each solution. These common features form building blocks for a subsequent hill climb. The authors found that the multiple hill-climbing approach does indeed guide the search to higher peaks in subsequent executions.

Spectral clustering algorithms operate as follows.

(1) Build the Laplacian matrix corresponding to the system's dependency graph.

(2) Determine the dominant eigenvalues and eigenvectors of the Laplacian matrix.

(3) Use these to compute the clustering.

Some researchers have adapted spectral graph partitioning to the decomposition of software systems [88, 93]. This

is based on a construction graph that represents relations between entities in the explored system. Spectral clustering algorithms are recursive. Each iteration splits the graph to two subgraphs and calculates the new value of the objective function. The recursion terminates as soon as the objective function stops improving.

First, the Laplacian matrix is calculated, and the smallest nonzero eigenvalue is found. This value will be used for the calculation of the characteristic vector (the characteristic vector is a n-dimensional vector $(x_1, \ldots, x_n)$ that defines two clusters. Entities whose x_i is 0 belong in the first cluster, entities whose x_i is 1 belong in the second cluster), which is used to partition the graph. The graph is divided into subgraphs based on the values of the characteristic vector. The algorithm uses the entries of the characteristic vector to split the entities so that the break-point maximizes the goal function. If the bisection improves the objective function, then the algorithm goes to the next iteration by splitting each sub-graph obtained in the previous step recursively. If splitting does not improve the solution, then the algorithm stops.

Xanthos [93] developed a spectral software clustering method that guarantees that the constructed partition is within a known factor of the optimal solution. The objective function used is the same as in the Bunch algorithms.

Finally, another form of algorithms that perform optimization is the so-called *clumping techniques* [72]. In each iteration, one cluster is identified. Repeated iterations discover different clusters (or clumps) which may overlap. A negative aspect of this method is that finding the same clump several times cannot be avoided completely.

3.4.4. Hierarchical Algorithms. There are two categories of hierarchical algorithms: agglomerative (bottom-up) and divisive (top-down).

Divisive algorithms start with one cluster that contains all entities and divide the cluster into a number (usually two) of separate clusters at each successive step. Agglomerative algorithms start at the bottom of the hierarchy by iteratively grouping similar entities into clusters. At each step, the two clusters that are most similar to each other are merged, and the number of clusters is reduced by one.

According to [94], divisive algorithms offer an advantage over agglomerative algorithms because most users are interested in the main structure of the data which consists of a few large clusters found in the first steps of divisive algorithms. Agglomerative algorithms start with individual entities and work their way up to large clusters which may be affected by unfortunate decisions in the first steps. Agglomerative hierarchical algorithms are most widely used however. This is because it is infeasible to consider all possible divisions of the first large clusters [72].

Agglomerative algorithms perform the following steps [95].

(1) Compute a similarity matrix.

(2) Find the two most similar clusters and join them.

(3) Calculate the similarity between the joined clusters and others obtaining a reduced matrix.

(4) Repeat from step (2) until two clusters are left.

The above process implies that there is a way to calculate the similarity between an already formed cluster and other clusters/entities. This is done via what is called the update rule function. Suppose that cluster i and cluster j are joined to form cluster ij. Typical update rule functions are

(1) single linkage: $\text{sim}(ij,k) = \min(\text{sim}(i,k), \text{sim}(j,k))$,

(2) complete linkage: $\text{sim}(ij,k) = \max(\text{sim}(i,k), \text{sim}(j,k))$,

(3) average linkage: $\text{sim}(ij,k) = (1/2)[\text{sim}(i,k) + \text{sim}(j,k)]$.

Maqbool and Babri [95] has concluded that for software clustering, the complete linkage update rule gives the most cohesive clusters. The same work introduced a new update rule called weighted combined linkage that provided better results than complete linkage. This result was achieved by applying the unbiased Ellenberg measure [95] and utilizing information regarding the number of entities in a cluster that accesses an artefact, thereby substantially reducing the number of arbitrary decisions made during the algorithm's clustering process.

UPGMA (Unweighted Pair Group Method with Arithmetic mean) [66] is an agglomerative hierarchical method used in bioinformatics for the creation of phylogenetic trees. Lung et al. [60] have shown applications of the UPGMA method in the software clustering context.

Andritsos and Tzerpos [24] presented the Scalable Information Bottleneck (LIMBO) algorithm, an agglomerative hierarchical algorithm that employs the Agglomerative Information Bottleneck algorithm (AIB) for clustering. LIMBO uses an information loss measure to calculate similarity between entities. At every step, the pair of entities that would result in the least information loss is chosen.

ACDC [96] is a hierarchical clustering algorithm that does not follow a standard schema. It cannot be assigned to the agglomerative or divisive category because the algorithm does not have an explicit iterative split or merge stage. ACDC uses patterns that have been shown to have good program comprehension properties to determine the system decomposition. ACDC systematically applies these subsystem patterns to the software structure. This results in most of the modules being placed into hierarchical categories (subsystems). Then, ACDC uses an orphan adoption algorithm [97] to assign the remaining modules to the appropriate subsystem.

We have discussed various algorithms and similarity techniques that have been adapted to the software clustering context.

An important observation is that there are two different conceptual approaches to developing a software clustering methodology. The first one attempts to develop a sophisticated structure discovery approach such as ACDC and Bunch. The second approach concentrates more on

developing similarity functions [74, 75]. An important open research question is the following: which approach is the most promising for the future of software clustering?

3.5. Results Visualization. The output of the cluster discovery stage needs to be presented in a user friendly manner. The challenge is to present a large amount of data with its relations in such a way that a user can understand and easily work with the data. The presentation of software clustering results is an open research question that is related to scientific visualization (the computer modeling of raw data) and human-computer interaction.

The software reverse engineering community has developed various tools for the presentation of the output of software clustering. These often include the traditional software representation of a graph. Such tools include CodeCrawler [98], Rigi [33], and LSEdit [39]. Rigi and LSEdit allow manual modification of the clustering result.

A different research direction is to develop visualizations of software systems by using metaphors and associations. For instance, Code City [99] is a 3D visualization tool that uses the City metaphor to visualize software systems. It is an integrated environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities. The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside. The visible properties of the city artefacts depict a set of chosen software metrics.

A reverse engineer will understand the output produced by a software clustering method better if the visualization techniques improve. Once a user has understood the clustering results, they may be able to provide feedback that can be used to refine method parameters. Therefore, effective visualization of results opens the way for the development of interactive software clustering algorithms where a user can iteratively assess the current results and steer the software clustering process. This concept is described in the next section.

3.6. Feedback. During the clustering process, an expert user should be able to instruct the tool on how to improve the overall solution. Clustering algorithms that are able to process user feedback are called semiautomatic clustering algorithms. Christl et al. [16] present a semiautomatic algorithm that allows the mapping of hypothesized high-level entities to source code entities. Koschke [75] created a semiautomatic clustering framework based on modified versions of the fully automatic techniques he investigated. The goal of Koschke's framework is to enable a collaborative session between his clustering framework and the user. The clustering algorithm does the processing, and the user validates the results.

Semiautomatic cluster analysis algorithms are more complicated than the fully automatic ones. They produce results that are closer to the expectations of the user than those of fully automatic methods. This is both an advantage and a disadvantage. A software engineer may explore different aspects of the software system by validating clustering results and providing feedback to the semiclustering algorithm. On the other hand, the result of a semiautomatic clustering algorithm may not reflect the actual state of the software system, because in many cases the software engineer may have wrong or incomplete understanding of the current state of the system. This drawback may explain the fact that the reverse engineering research community is mostly developing automatic clustering algorithms.

4. Evaluation of Clustering Algorithms

Software clustering researchers have developed several evaluation methods for software clustering algorithms. This research is important because of these reasons.

(1) Most software clustering work is evaluated based on case studies. It is important that the evaluation technique is not subjective.

(2) Evaluation helps discover the strengths and weaknesses of the various software clustering algorithms. This enables the improvement of the algorithms by eliminating or alleviating the effect of the weaknesses.

(3) Evaluation can help indicate the types of system that are suitable for a particular algorithm. For instance, Mitchell and Mancoridis [12] think that Bunch [92] may not be suitable for event-driven systems.

The importance of evaluating software clustering algorithms was first stated in 1995 by Lakhotia and Gravley [100]. Since then, many approaches to this problem have been published in the literature. These can be divided in two categories depending on whether they rely on the existence of an *authoritative decomposition*, that is, a decomposition of the software system at hand into meaningful subsystems that has been constructed by an expert in this system, for example, the system architect or a senior developer. In contrast, we refer to decompositions created by software clustering algorithms as *automatic decompositions*.

Section 4.1 presents software clustering evaluation approaches that require an authoritative decomposition, while Section 4.2 discusses approaches that are independent of the existence of an authoritative decomposition.

4.1. Evaluation Based on Authoritative Decomposition. Before presenting the various approaches that have been presented in the literature to assess the similarity between automatic and authoritative decompositions, it is important to point out a crucial fact regarding such software clustering evaluation methods.

Different experts on the same software system may construct distinct software decompositions, since a software system may have several equally valid decompositions. In other words, an authoritative decomposition must be considered as one of many possible decompositions that can help understand a software system. As a result, clustering researchers should evaluate their approaches against all available authoritative decompositions. Furthermore, it is important to remember that evaluation results against a given

authoritative decomposition are applicable only in the context of this particular authoritative decomposition, rather than being universally applicable.

With this caveat in mind, we present two orthogonal classifications of evaluation approaches that utilize authoritative decompositions.

The first classification depends on whether the approach considers only the two decompositions, that is, an automatic and an authoritative one or more aspects of the software system. Most evaluation methods do consider only the two decompositions.

However, some methods focus on the evaluation of a specific stage of the software clustering process. Such a method calculates the quality of a specific stage based on analysis of its inputs and outputs. For instance, an evaluation method that focuses on the evaluation of the analysis phase will take into account the input data model, compare the authoritative decomposition and the produced software clustering decomposition, and then calculate a number that reflects the quality of the produced decomposition. A typical example of such a method is EdgeSim [101].

The main reason for the development of both types of evaluation methods is that the quality of a produced decomposition depends on the

(1) selection of an appropriate clustering algorithm—a different clustering algorithm produces different outputs from the same factbase,

(2) selection of input parameters—a clustering algorithm might produce different outputs depending on selected input parameters, such as similarity function or input data model.

An orthogonal classification of software clustering evaluation methods depends on whether the decompositions compared are *nested* or not. Most evaluation approaches assume a *flat* decomposition, that is, one where no clusters are contained within other clusters. A small number of approaches deal with nested decompositions: the END framework [11] that allows for the reuse of flat evaluation approaches for nested decompositions without loss of information and the UpMoJo algorithm presented below.

The rest of this section presents the most prominent approaches for the comparison of an automatic decomposition to an authoritative one.

4.1.1. MoJo Family. Tzerpos and Holt developed a distance measure called MoJo [10]. It attempts to capture the distance between two decompositions as the minimum number of Move and Join operations that need to be performed in order to transform one decomposition into the other. A Move operation involves relocating a single entity from one cluster to another (or to a new cluster), while a Join operation takes two clusters and merges them into a single cluster. MoJo distance is nonsymmetric.

Wen and Tzerpos [102] presented an algorithm that calculates MoJo distance in polynomial time. They also introduced the MoJoFM effectiveness measure which is based on MoJo distance but produces a number in the range 0–100 which is independent of the size of the decompositions [103].

UpMoJo distance is an extension of MoJo distance so that it applies to nested decompositions [104]. An Up operation is added that allows entities to move to higher levels of the containment hierarchy, for example, from a cluster that is contained within other clusters to a top-level cluster.

4.1.2. Precision/Recall. Precision and Recall are standard metrics in Information Retrieval. They have been applied to the evaluation of software clustering by Anquetil and Lethbridge [13]. The method calculates similarity based on measuring intrapairs, which are pairs of entities that belong to the same cluster. The authors suggest to calculate precision and recall for a given partition as follows.

(i) *Precision.* Percentage of intrapairs proposed by the clustering method, which are also intrapairs in the authoritative decomposition.

(ii) *Recall.* Percentage of intra pairs in the authoritative decomposition, which are also intra pairs in the decomposition proposed by the clustering method.

Mitchell and Mancoridis [12] explain a drawback of the Precision/Recall metrics: "An undesirable aspect of Precision/Recall is that the value of this measurement is sensitive to the size and number of clusters."

4.1.3. Koschke and Eisenbarth. Koschke and Eisenbarth [14] have presented a way of quantitatively comparing automatic and authoritative partitions that establishes corresponding clusters (Good match), handles partially matching clusters (OK match), tolerates smaller divergences, and determines an overall recall rate. Their approach is as follows.

(1) Identify immediately corresponding clusters (Good match).

(2) Identify corresponding subclusters, that is, where part of a cluster of one partition corresponds to part of a cluster in the other partition (OK match).

(3) Measure accuracy of the correspondences between two partitions.

This metric has two drawbacks:

(1) it does not penalize the software clustering algorithm when an automatic decomposition is more detailed than the authoritative one,

(2) the recall rate does not distinguish between a Good match and an OK match.

The authors have also developed benchmark scenarios for the evaluation of software clustering and the calibration of software clustering parameters.

4.1.4. EdgeSim and MeCl. Mitchell and Mancoridis [101] developed the first method for the evaluation of a software clustering algorithm in conjunction with an input data

model. They introduced a similarity (EdgeSim) and a distance (MeCl) measurement that considers relations between entities as an important factor for the comparison of software system decompositions. The EdgeSim similarity measurement normalizes the number of intra and intercluster edges that agree between two partitions. The MeCl similarity measurement determines the distance between a pair of partitions by first calculating the "clumps," which are the largest subset of modules from both partitions that agree with respect to their placement into clusters. Once the clumps are determined, a series of Merge operations are performed to convert the first partition into the second one. The actual MeCl distance is determined by normalizing the number of Merge operations.

Unfortunately, these two measures can be applied only when the input data model is a dependency graph. Another drawback of the method is the assumption that the authoritative decomposition is a direct reflection of the dependencies in the factbase. It is difficult to confirm that this assumption holds because the system expert that has constructed the authoritative decomposition has certainly used additional knowledge that may contradict this assumption. Wen and Tzerpos [105] explain the main drawback of EdgeSim and MeCl: "it only considers the misplacement of edges without considering the misplacement of the objects in the wrong clusters."

4.2. Evaluation Methods Not Based on an Authoritative Decomposition. The main drawback of methods that require an authoritative decomposition is that they assume that such a decomposition exists. To construct such a decomposition for a middle-size software system is a challenging task. Various research studies [14, 101, 106] deal with the construction of an authoritative decomposition. This work addresses two questions:

(1) what is the right process for the construction of an authoritative decomposition?

(2) why is the constructed decomposition authoritative?

Lakhotia and Gravley [100] said, "If we know the subsystem classification of a software system, then we do not need to recover it." Another aspect of the problem is that after a researcher constructs an authoritative decomposition, they may find out that the authoritative decomposition cannot be used for evaluation purposes. For instance, if the authoritative decomposition is flat then it is not suitable to evaluate algorithms that produce nested decompositions. Finally, a software system may have a number of different authoritative decompositions.

To overcome these problems several evaluation methods that evaluate a software clustering approach based solely on its output have been developed [13, 107]. Wu et al. [107] suggest comparing software clustering algorithms based on two criteria:

(1) stability,

(2) extremity of cluster distribution.

4.2.1. Stability. Stability reflects how sensitive is a clustering approach to perturbations of the input data. Raghavan [108] has shown the importance of developing such a metric. The intuition behind stability in software clustering is that similar clustering decompositions should be produced for similar versions of a software system. A stability function measures the percentage of changes between produced decompositions of successive versions of an evolving software system. Under conditions of small changes between consecutive versions, an algorithm should produce similar clustering [107].

Tzerpos and Holt [109] define a stability measure based on the ratio of the number of "good" decompositions to the total number of decompositions produced by a clustering algorithm. A decomposition, which is obtained from a slightly modified software system, is defined as "good" if and only if the MoJo distance between the decomposition and the decomposition obtained from the original software system is not greater than 1% of the total number of entities.

Wu et al. [107] argued that a drawback of the aforementioned stability measure is that 1% seems too optimistic in reality. In the same paper, the authors suggest a relative stability measure. This measure allows one to say that one algorithm is more stable than another with regard to a software system. To calculate the relative score of two clustering algorithms, the authors suggest constructing sequences of MoJo values calculated based on comparing two consecutive members of the sequence of decompositions obtained from a software system. Then, based on the two sequences of MoJo values, a number is calculated that represents a relative score of one algorithm over the other.

4.2.2. Extremity of Cluster Distribution. An interesting property of a clustering algorithm is the size of the clusters it produces. The cluster size distribution of a clustering algorithm should not exhibit extremity. In other words, a clustering algorithm should avoid the following situations.

(1) The majority of files are grouped into one or few huge clusters.

(2) The majority of clusters are singletons.

Wu et al. [107] presented a measure called NED (nonextreme distribution) that allows to evaluate the extremity of a cluster distribution. NED is defined as the ratio of the number of entities contained in nonextreme clusters to the total number of the entities.

Anquetil and Lethbridge [13] have investigated the causes of extreme cluster distribution. They found that poor input data model and unsuitable similarity metrics can cause such results. For instance, the majority of the clusters being singletons are a sign of a bad descriptive attribute. A reason that the majority of files are grouped into one or few huge clusters is a consequence of an ill-adapted algorithm or similarity metric.

The main drawback of both presented methods is that they can only identify "bad" software decompositions. It is easy to develop an algorithm that produces decompositions that are stable and nonextreme, while being entirely useless for program understanding purposes.

Mitchell and Mancoridis [12] developed a framework called CRAFT for the evaluation of software clustering algorithms without authoritative decomposition. The proposed evaluation process consists of two phases. The first phase is the construction of an authoritative decomposition. The CRAFT framework automatically creates an authoritative decomposition based on common patterns produced by various clustering algorithms. The initial step of the process is to cluster the target system many times using different clustering algorithms. For each clustering run, all the modules that appear in the same cluster are recorded. Using this information, CRAFT exposes common patterns in the results produced by different clustering algorithms, and it constructs an authoritative decomposition. The assumption of the approach is that agreement across a collection of clustering algorithms should reflect the underlying structure of the software system.

The second phase is to compare the created authoritative decomposition with the automatic decomposition by applying an evaluation method that requires an authoritative decomposition. The advantage of CRAFT is that it is applicable when an authoritative decomposition does not exist. The main drawback of the method is that the automatically produced authoritative decomposition may be unacceptable according to a software expert. It can only be as good as the algorithms it utilized.

We presented an overview of the important methods for the evaluation of software clustering algorithms. Most methods require an authoritative decomposition (either flat or nested). Methods that do not require an authoritative decomposition can be used to weed out bad clustering results, that is, to select appropriate parameters for the clustering algorithms.

5. Research Challenges

Having presented the state of the art of software clustering methods, we go on to discuss open research challenges in software clustering. We have selected research questions that are related to the following topics:

(1) attribute gathering,

(2) cluster discovery,

(3) visualization of results,

(4) user feedback,

(5) evaluation of clustering algorithms.

Before we present each topic in detail, we bring attention to the current situation with software clustering tools. The research community has shown many advantages to using software clustering methods in different software engineering areas. Unfortunately, software clustering methodologies are not widely accepted in the industrial environment. There is still a long way to go before software clustering methods become an effective and integral part of the IDE [58]. For that to happen, existing tools must become easier to install, more robust, and significantly more user-friendly in terms of the presentation and manual adjustment of clustering results.

5.1. Fact Extraction. We discussed various input sources for the attribute gathering process. Typically, a clustering method will utilize only one of these sources, such as source or binary code. However, it is possible to construct input metamodels that combine information from different sources [27]. The reason this does not happen more often in practice is the lack of an established methodology. Such a methodology has to answer various questions including the following.

(1) Is it important to mix various sources? How important is a good metamodel?

(2) Which source(s) contain(s) the most important information?

(3) What is an effective way to represent artefacts gathered from various input sources in the same format in one meta-model? In other words, we have to define a language that allows the expression of an artefact extracted from any input source.

(4) What is a reasonable weight scheme for artefacts combined from various sources? Does it vary based on factors such as the type of system or its development methodology?

(5) What are other input sources for extraction of artefacts that could be integrated into the input metamodel?

These are the key questions for the attribute gathering stage. The answers to those questions would allow the development of a good meta-model, which should result in better clustering results. In addition, these answers would allow the research community to merge their efforts to develop a common methodology for gathering attributes. Today, efforts are split on developing advanced methods for extracting data from specific input sources.

5.2. Cluster Discovery. The cluster discovery process encompasses both the similarity computing and cluster creation phases. It is a complicated process that has been discussed in many research studies. Still, several key questions remain unanswered. We present a list of as yet unsolved problems that pose interesting challenges to researchers in the area.

(1) The best direction towards improving this process needs to be determined. As mentioned earlier, the cluster discovery process can be improved by developing a new software clustering algorithm or developing a new resemblance coefficient technique. What is a better way to improve the cluster discovery process is an open question. Since this question is open, the research community is splitting its efforts by developing both software clustering algorithm and resemblance coefficient techniques instead of focusing on one or the other.

(2) Several software clustering algorithms and resemblance coefficient techniques have been developed. Therefore, selecting a software clustering algorithm and a suitable resemblance coefficient is a challenging

task. A comparative study tested on a number of systems is long overdue. It is possible that particular combinations of clustering algorithms with resemblance coefficients are better suited for a particular type of software system. A categorization of algorithms based on the types of software for which they work best would be beneficial to the software field.

(3) The decomposition of a software system into subsystems is a typical task of a reverse engineer. A software clustering method can produce a nested decomposition, typically, based on the output of a hierarchical algorithm. Currently, there is no optimization-based software clustering algorithm that can produce a nested decomposition. We know that optimization algorithms can efficiently construct flat decompositions of a software system. It would be worth investigating whether an optimization algorithm would be an efficient algorithm for extracting a nested decomposition from a software system as well.

(4) Since a particular software system may have various valid decompositions, the output of a software clustering algorithm is often not meaningful to a software engineer. To overcome this problem, some software clustering algorithms assign labels for each cluster. Label assignment is a challenging task, and it is undeveloped in the context of software clustering. The problem is that software clustering algorithms either do not assign labels or assign labels based on simplistic rules. The development of an advanced method for label assignment would be beneficial to the field.

5.3. Results Visualization. As presented earlier, the visualization of results is a challenging problem because often the results of software clustering methods are large decompositions. Therefore, it is complicated to develop a user interface, which is capable of presenting a software decomposition in a meaningful way to a software engineer. In addition, the user interface is supposed to allow navigation through software decomposition results and to link the decomposition to source files, documentation, and so forth. Software clustering methods would be more useful if the visualization of the results of the software clustering was improved.

5.4. User Feedback. The importance of developing semiautomatic clustering algorithms was discussed earlier in this paper. There are two main obstacles to developing advanced semiautomatic clustering algorithms. The first one is the lack of a user interface that allows the users to understand and evaluate the clustering results. The second and more important obstacle is that the results of the semiautomatic algorithm may not reflect the current state of the software system because it is driven by the user whose knowledge of the software system is often incomplete. We think that it is important to develop criteria that allow the algorithm to disregard wrong user feedback. When such criteria have been established, then the research community will be able to pay more attention to semiautomatic software clustering algorithms.

5.5. Evaluation. Evaluation of software clustering algorithms has already been studied from various aspects, but still there are uncharted research areas including the following.

(1) Each clustering approach uses a different set of evaluation measures from the several measurements that have been presented in the literature. Consequently, we cannot compare evaluation results from different studies. To overcome this problem, the evaluation metrics have to be compared to each other. It is worth investigating whether certain metrics are correlated with each other for particular types of software systems.

(2) The research software community needs to define a standardized method of evaluating software clustering. This requires the collection of reference software systems. This collection has to contain many different types of software systems. The construction of such a collection is complicated because it is difficult to find good candidates. Most research studies propose using big open source software systems as reference systems. However typically, a big software system includes several types of software paradigms (event-based, web services, etc.). Another challenge is that new software types constantly appear. Until now, no study has developed a deep theoretic justification for the construction of a collection of reference systems. The reference collection would help to define a standard way of comparing software clustering systems. In addition, it would allow better exploration of properties of existing software clustering methods and development of new more advanced methods.

(3) Most of the research work presented in the literature is about the evaluation of the complete software clustering process. Until now, there is no dedicated study to establish a method for the evaluation of a specific phase of the software clustering process. Such a study would be an important tool for the research community. It would allow reverse engineers to select an appropriate algorithm for their task.

6. Conclusion

This paper presented the state of the art in the development and evaluation of software clustering methodologies. We also outlined the most important research challenges for this important area of research. It should be apparent that while important advances have already taken place, there are still many avenues for further research that will benefit software engineers everywhere.

References

[1] H. A. Muller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse engineering approach to subsystem structure identification," *Journal of Software Maintenance*, vol. 5, pp. 181–204, 1993.

[2] S. C. Choi and W. Scacchi, "Extracting and restructuring the design of large systems," *IEEE Software*, vol. 7, no. 1, pp. 66–71, 1990.

[3] N. Anquetil and T. Lethbridge, "File clustering using naming conventions for legacy systems," in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative research (CASCON '97)*, pp. 184–195, November 1997.

[4] C. Lindig and G. Snelting, "Assessing modular structure of legacy code based on mathematical concept analysis," in *Proceedings of the IEEE 19th International Conference on Software Engineering*, pp. 349–359, May 1997.

[5] R. W. Schwanke, R. Altucher, and M. A. Platoff, "Discovering, visualizing, and controlling software structure," in *Proceedings of the International Workshop on Software Specification and Design (IWSSD '89)*, pp. 147–150, IEEE Computer Society Press, 1989.

[6] J. F. Cui and H. S. Chae, "Applying agglomerative hierarchical clustering algorithms to component identification for legacy systems," *Information and Software Technology*, vol. 53, no. 6, pp. 601–614, 2011.

[7] Y. Wang, P. Liu, H. Guo, H. Li, and X. Chen, "Improved hierarchical clustering algorithm for software architecture recovery," in *Proceedings of the International Conference on Intelligent Computing and Cognitive Informatics (ICICCI '10)*, pp. 247–250, Kuala Lumpur, Malaysia, June 2010.

[8] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *Proceedings of the Software Maintenance and Reengineering (CSMR '09)*, pp. 27–36, IEEE Computer Society, Kaiserslautern, Germany, March 2009.

[9] V. Tzerpos, *Comprehension-Driven Software Clustering*, Ph.D. thesis, University of Toronto, Toronto, Canada, 2001.

[10] V. Tzerpos and R. C. Holt, "MoJo: a distance metric for software clusterings," in *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pp. 187–193, October 1999.

[11] M. Shtern and V. Tzerpos, "A framework for the comparison of nested software decompositions," in *Proceedingsof the 11th Working Conference on Reverse Engineering (WCRE '04)*, pp. 284–292, Delft, The Netherlands, November 2004.

[12] B. S. Mitchell and S. Mancoridis, "Craft: a framework for evaluating software clustering results in the absence of benchmark decompositions," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pp. 93–102, Suttgart, Germany, October 2001.

[13] N. Anquetil and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pp. 235–255, Atlanta, Ga, USA, October 1999.

[14] R. Koschke and T. Eisenbarth, "A framework for experimental evaluation of clustering techniques," in *Proceedings of the International Workshop on Program Comprehension (IWPC '00)*, pp. 201–210, Limerick, Ireland, June 2000.

[15] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," in *Proceedings of the 1995 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 18–27, October 1995.

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

[17] X. Xu, C.-H. Lung, M. Zaman, and A. Srinivasan, "Program restructuring through clustering techniques," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '04)*, pp. 75–84, IEEE Computer Society, Ottawa, Canada, September 2004.

[18] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Analyzing software measurement data with clustering techniques," *IEEE Intelligent Systems*, vol. 19, no. 2, pp. 20–27, 2004.

[19] C. H. Lung, M. Zaman, and A. Nandi, "Applications of clustering techniques to software partitioning, recovery and restructuring," *Journal of Systems and Software*, vol. 73, no. 2, pp. 227–244, 2004.

[20] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Proceedings of the 13th International Conference on Software Engineering*, pp. 83–92, May 1991.

[21] M. Bauer and M. Trifu, "Architecture-aware adaptive clustering of OO systems," in *Proceedings of the 8th European Conference on Software Maintainance and Reengineering (CSMR '04)*, pp. 3–14, Tampere, Finland, March 2004.

[22] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," in *Proceedings of the19th IEEE International Conference on Software Maintenance (ICSM '03)*, p. 315, IEEE Computer Society, Amsterdam, The Netherlands, September 2003.

[23] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Proceedings of the Software Maintenance and Reengineering (CSMR '05)*, pp. 124–133, IEEE Computer Society, Manchester, UK, March 2005.

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

[25] A. Shokoufandeh, S. Mancoridis, and M. Maycock, "Applying spectral methods to software clustering," in *Proceedings of the Working Conference on Reverse Engineering (WCRE '02)*, p. 3, IEEE Computer Society, Richmond, VA, USA, November 2002.

[26] G. Canfora, J. Czeranski, and R. Koschke, "Revisiting the delta IC approach to component recovery," in *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE '00)*, p. 140, IEEE Computer Society, Brisbane, Australia, 2000.

[27] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cîmpan, and H. Verjus, "Towards a process-oriented software architecture reconstruction taxonomy," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*, pp. 137–148, Amsterdam, Netherlands, March 2007.

[28] S. Demeyer, S. Tichekaar, and S. Ducasse, "FAMIX 2.1—the FAMOOS information exchange model," Tech. Rep., University of Bern, Bern, Switzerland, 2001.

[29] R. C. Holt, "Structural manipulations of software architecture using Tarski relational algebra," in *Proceedings of the 1998 5th Working Conference on Reverse Engineering*, pp. 210–219, October 1998.

[30] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder, "The Dagstuhl Middle Metamodel: a schema for reverse engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 7–18, 2004.

[31] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter, "GXL: a graph-based standard exchange format for reengineering,"

Science of Computer Programming, vol. 60, no. 2, pp. 149–170, 2006.

[32] B. S. Mitchell, *A Heuristic Search Approach to Solving the Software Clustering Problem*, Ph.D. thesis, Drexel University, Philadelphia, Pa, USA, 2002, Adviser-Spiros Mancoridis.

[33] H. A. Muller, S. R. Tilley, and K. Wong, "Understanding software systems using reverse engineering technology perspectives from the Rigi project," in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative research (CASCON '93)*, pp. 217–226, IBM Press, Ontario, Canada, 1993.

[34] J. Bézivin, F. Jouault, and P. Valduriez, "On the need for megamodels," in *Proceedings of the Workshop on Best Practices for Model-Driven Software Development at ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '04)*, Vancouver, British Columbia, Canada, October 2004.

[35] L. Tahvildari, R. Gregory, and K. Kontogianni, "An approach for measuring software evolution using source code features," in *Proceedings of the Asia-Pacific Software Engineering Conference*, p. 10, 1999.

[36] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04)*, pp. 184–193, Busan, Korea, December 2004.

[37] A. Kuhn, S. Ducasse, and T. Gîrba, "Enriching reverse engineering with semantic clustering," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE '05)*, pp. 133–142, Pittsburgh, Pa, USA, November 2005.

[38] J. Dietrich, V. Yakovlev, C. McCartiny, G. Jenson, and M. Duchrow, "Cluster analysis of Java dependency graphs," in *Proceedings of the 4th ACM Symposium on Software Visualization (SOFTVIS '08)*, pp. 91–94, Munich, Germany, September 2008.

[39] http://www.swag.uwaterloo.ca.

[40] J. Korn, Y. F. Chen, and E. Koutsofios, "Chava: reverse engineering and tracking of Java applets," in *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pp. 314–325, October 1999.

[41] G. Huang, H. Mei, and F. Q. Yang, "Runtime recovery and manipulation of software architecture of component-based systems," *Automated Software Engineering*, vol. 13, no. 2, pp. 257–281, 2006.

[42] E. Stroulia and T. Systa, "Dynamic analysis for reverse engineering and program understanding," *ACM SIGAPP Applied Computing Review*, vol. 10, no. 1, pp. 8–17, 2002.

[43] R. J. Walker, G. C. Murphy, B. N. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak, "Visualizing dynamic software system information through high-level models," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, pp. 271–283, Vancouver, Canada, 1998.

[44] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: a system for discovering architectures from running systems," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 470–479, Scotland, UK, May 2004.

[45] Q. Zhang, Q. Qiu, and L. Sun, "Objectoriented software architecture recovery using a new hybrid clustering algorithm," in *Proceedings of the International Conference on Fuzzy Systems and Knowledge Discovery (FSKD '10)*, vol. 6, pp. 2546–2550, Shandong, china, August 2010.

[46] http://www.eclipse.org/tptp/platform/documents/probekit/ /probekit.html.

[47] D. B. Lange and Y. Nakamura, "Object-oriented program tracing and visualization," *Computer*, vol. 30, no. 5, pp. 63–70, 1997.

[48] T. Systa, *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, Ph.D. thesis, Tampere University, Tampere, Finland, 2000.

[49] M. Dmitriev, "Profiling Java applications using code hotswapping and dynamic call graph revelation," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 1, pp. 139–150, 2004.

[50] M. Lungu, M. Lanza, and T. Gîrba, "Package patterns for visual architecture recovery," in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR '06)*, pp. 185–194, Bari, Italy, March 2006.

[51] M. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[52] R. Wuyts, *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, Ph.D. thesis, Vrije Universiteit Brussel, Amsterdam, Netherlands, 2001.

[53] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS '05)*, pp. 261–269, Como, Italy, September 2005.

[54] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance*, pp. 23–32, Amsterdam, The Netherlands, September 2003.

[55] A. Hassan and R. Holt, "Studying the evolution of software systems using evolutionary code extractors," in *Proceedings of the InternationalWorkshop on Principles of Software Evolution (IWPSE '04)*, pp. 76–81, Kyoto, Japan, September 2004.

[56] N. Medvidovic and V. Jakobac, "Using software evolution to focus architectural recovery," *Automated Software Engineering*, vol. 13, no. 2, pp. 225–256, 2006.

[57] A. E. Hassan and R. C. Holt, "Reference architecture for web servers," in *Proceedings of the 7th Conference on Reverse Engineering (WCRE '00)*, pp. 150–159, November 2000.

[58] H. A. Muller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proceedings of the Proceedings of International Conference on Software Engineering (ICSE '00)*, pp. 47–60, Limerick, Ireland, June 2000.

[59] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang, "Clustering large software systems at multiple layers," *Information and Software Technology*, vol. 49, no. 3, pp. 244–254, 2007.

[60] C. H. Lung, M. Zaman, and A. Nandi, "Applications of clustering techniques to software partitioning, recovery and restructuring," *Journal of Systems and Software*, vol. 73, no. 2, pp. 227–244, 2004.

[61] D. M. German, D. Cubranic, and M.-A. D. Storey, "A framework for describing and understanding mining tools in software development," in *Proceedings of the International Workshop on Mining Software Repositories (MSR '05), Proceedings of the International Workshop on Mining software repositories*, pp. 1–5, ACM, Saint Louis, Mo, USA, July 2005.

[62] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: a clustering tool for the recovery and maintenanceof software system structures," in *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, IEEE Computer Society Press, Oxford, UK, August 1999.

[63] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering behavioral design models from execution traces," in *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR '05)*, pp. 112–121, Manchester, UK, March 2005.

[64] Z. Wen and V. Tzerpos, "Software clustering based on omnipresent object detection," in *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*, pp. 269–278, St. Louis, Mo, USA, May 2005.

[65] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, Upper Saddle River, NJ, USA, 1988.

[66] H. C. Romesburg, *Clustering Analysis for Researchers*, Krieger, Melbourne, Fla, USA, 1990.

[67] R. Naseem, O. Maqbool, and S. Muhammad, "Improved similarity measures for software clustering," in *Proceedings of the Software Maintenance and Reengineering (CSMR '11)*, pp. 45–54, march 2011.

[68] R. Naseem, O. Maqbool, and S. Muhammad, "An improved similarity measure for binary features in software clustering," in *Proceedings of the 2nd International Conference on Computational Intelligence, Modelling and Simulation (CIMSim '10)*, pp. 111–116, Islamabad, Pakistan, September 2010.

[69] H. Dhama, "Quantitative models of cohesion and coupling in software," in *Proceedings of the Annual Oregon Workshop on SoftwareMetrics (AOWSM '95)*, pp. 65–74, Elsevier Science, June 1995.

[70] J. Davey and E. Burd, "Evaluating the suitability of data clustering for software remodularisation," in *Proceedings of the 7th Conference on Reverse Engineering (WCRE '00)*, pp. 268–276, November 2000.

[71] T. M. Lim and H. W. Khoo, "Sampling properties of Gower's general coefficient of similarity," *Ecology*, vol. 66, no. 5, pp. 1682–1685, 1985.

[72] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Proceedings of the 4th Working Conference on Reverse Engineering*, pp. 33–43, October 1997.

[73] P. H. A. Sneath and R. R. Sokal, *Numerical Taxonomy: The Principles and Practice of Numerical Classification*, Series of books in biology, W. H. Freeman, Gordonsville, VA, USA, 1973.

[74] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Proceedings of the 13th International Conference on Software Engineering*, pp. 83–92, May 1991.

[75] R. Koschke, *Atomic Architectural Component Recovery for Program Understanding and Evolution*, Ph.D. thesis, Stuttgart University, Stuttgart, Germany, 2000.

[76] G. von Laszewski, May 1993, A collection of graph partitioning algorithms:Simulated annealing, simulated tempering, kemighan lin, two optimal, graph reduction, bisection.

[77] R. A. Botafogo and B. Shneiderman, "Identifying aggregates in hypertext structures," in *Proceedings of the ACM Hypertext and Hypermedia*, pp. 63–74, ACM Press, NewYork, NY, USA, 1991.

[78] A. Trifu, *Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems*, M.S. thesis, University of Karlsruhe, Karlsruhe, Germany, 2001.

[79] I. Gitman and M. D. Levine, "An algorithm for detecting unimodal fuzzy sets and Its application as a clustering technique," *IEEE Transactions on Computers*, vol. C-19, no. 7, pp. 583–593, 1970.

[80] D. Wishart, "Mode Analysis: a generalization of nearest neighbour which reduces chaining effects," in *Numerical Taxonomy*, N. Taxonomy, Ed., pp. 282–311, Academic Press, New York, NY, USA, 1969.

[81] R. Lutz, "Evolving good hierarchical decompositions of complex systems," *Journal of Systems Architecture*, vol. 47, no. 6, pp. 613–634, 2001.

[82] A. S. Mamaghani and M. R. Meybodi, "Clustering of software systems using new hybrid algorithms," in *Proceedings of the IEEE 9th International Conference on Computer and Information Technology (CIT '09)*, pp. 20–25, Xiamen, China, October 2009.

[83] J. Clarke, J. J. Dolado, M. Harman et al., "Reformulating software engineering as a search problem," *IEE Proceedings*, vol. 150, no. 3, pp. 161–175, 2003.

[84] K. Praditwong, "Solving software module clustering problem by evolutionary algorithms," in *Proceedings of the 8th International Joint Conference on Computer Science and Software Engineering (JCSSE '11)*, pp. 154–159, Pathom, Thailand, May 2011.

[85] M. R. Anderberg, *Cluster Analysis for Applications*, Academic Press, New York, NY, USA, 1973.

[86] D. Doval, S. Mancoridis, and B. Mitchell, "Automatic clustering of software systems using a genetic algorithm," in *Proceedings of the Software Technology and Engineering Practice (STEP '99)*, pp. 73–81, Pittsburgh, PA , USA, August 1999.

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

[88] A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock, "Spectral and meta-heuristic algorithms for software clustering," *Journal of Systems and Software*, vol. 77, no. 3, pp. 213–223, 2005.

[89] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Searchbased improvement of subsystem decompositions," in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO '05)*, pp. 1045–1051, ACM Press, Washington, DC, USA, June 2005.

[90] P. de Lit, E. Falkenauer, and A. Delchambre, "Grouping genetic algorithms: an efficient method to solve the cell formation problem," *Mathematics and Computers in Simulation*, vol. 51, no. 3-4, pp. 257–271, 2000.

[91] B. Mitchell and S. Mancoridis, "Using heuristic search techniques to extract design abstractions from source code," in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO '02)*, New York, NY, USA, July 2002.

[92] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proceedings of the International Workshop on Program Comprehension (IWPC '98)*, IEEE Computer Society Press, Ischia , Italy, Junuary1998.

[93] S. Xanthos, 2006, Clustering Object-Oriented Software Systems using Spectral Graph Partitioning.

[94] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, John Wiley, New York, NY, USA, 1990.

[95] O. Maqbool and H. A. Babri, "The weighted combined algorithm: a linkage algorithm for software clustering," in *Proceedings of the European Conference on Software Maintainance and Reengineering (CSMR '04)*, pp. 15–24, Tampere, Finland, March 2004.

[96] V. Tzerpos and R. C. Holt, "ACDC: an algorithm for comprehension-driven clustering," in *Proceedings of the 7th Conference on Reverse Engineering (WCRE '00)*, pp. 258–267, November 2000.

[97] V. Tzerpos and R. C. Holt, "Orphan adoption problem in architecture maintenance," in *Proceedings of the 4th Working Conference on Reverse Engineering*, pp. 76–82, October 1997.

[98] M. Lanza and S. Ducasse, "Polymetric views—a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, 2003.

[99] R. Wettel and M. Lanza, "Program comprehension through software habitability," in *Proceedings of the International Conference on Program Comprehension (ICPC '07)*, pp. 231–240, Banff, Canada, June 2007.

[100] A. Lakhotia and J. M. Gravley, "Toward experimental evaluation of subsystem classification recovery techniques," in *Proceedings of the 2nd Working Conference on Reverse Engineering*, pp. 262–269, July 1995.

[101] B. S. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*, pp. 744–753, Florence, Italy, November 2001.

[102] Z. Wen and V. Tzerpos, "An optimal algorithm for MoJo distance," in *Proceedings of the International Workshop on Program Comprehension (IWPC '03)*, pp. 227–235, Portland, Ore, USA, May 2003.

[103] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings of the 12th International Workshop on Program Comprehension (IWPC '04)*, pp. 194–203, Bari, Italy, June 2004.

[104] M. Shtern and V. Tzerpos, "Lossless comparison of nested software decompositions," in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07)*, pp. 249–258, Vancouver, Canada, October 2007.

[105] Z. Wen and V. Tzerpos, "Evaluating similarity measures for software decompositions," in *Proceedings of the International Conference on Software Maintenance (ICSM '04)*, pp. 368–377, IEEE Computer Society, Chicago Ill, USA, September 2004.

[106] J. F. Girard and R. Koschke, "Comparison of abstract data types and objects recovery techniques," *Science of Computer Programming*, vol. 36, no. 2, pp. 149–181, 2000.

[107] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Proceedings of the International Conference on SoftwareMaintenance (ICSM '05)*, pp. 525–535, IEEE Computer Society, Budapest, Hungary, September 2005.

[108] V. V. Raghavan, "Approaches for measuring the stability of clustering methods," *SIGIR Forum*, vol. 17, no. 1, pp. 6–20, 1982.

[109] V. Tzerpos and R. C. Holt, "On the stability of software clustering algorithms," in *Proceedings of the International-Workshop on Program Comprehension (IWPC '00)*, pp. 211–218, Limerick, Ireland, June 2000.