Using Machine Learning and Information Retrieval Techniques to Improve Software Maintainability

Anna Corazza [1] and **Sergio Di Martino** [2] and **Valerio Maggio** [3] and
Alessandro Moschitti [4] and **Andrea Passerini** [5] and **Giuseppe Scanniello** [6] and **Fabrizio Silvestri** [7]

Abstract. The software architecture plays a fundamental role in the comprehension and maintenance of large and complex systems. However, unlike classes or packages, this information is not explicitly represented in the code, giving rise to the definition of different approaches to automatically recover the original architecture of a system. Software architecture recovery (SAR) techniques aim at extracting architectural information from the source code by often involving clustering of program artifacts analyzed at different levels of abstraction (e.g, classes or methods).

In this paper, we capitalize our expertise in Machine Learning, Natural Language Processing and Information Retrieval to outline promising research lines in the field of automatic SAR. In particular, after presenting an extensive related work, we illustrate a concrete proposal for solving two main subtasks of SAR, i.e., (I) software clone detection and (II) clustering of functional modules according to their lexical semantics. One interesting aspect of our proposed research is the use of advanced approaches, such as kernel methods, for exploiting structural representation of source code.

1 Introduction

Software maintenance is essential in the evolution of software systems and represents one of the most expensive, time consuming, and challenging phase of the whole development process. As declared in Lehman's laws of Software Evolution [31], a software system must be continuously adapted during its overall life cycle or it progressively becomes less satisfactory (Lehman's first law). Thus, as software applications are doomed to evolve and grow [13], all of the applied changes and adaptations inevitably reduce their quality (Lehman's second law). Moreover, the cost and the effort necessary for both the maintenance and the evolution operations (e.g., corrective, adaptive, etc.) are mainly related to the effort necessary to comprehend the system and its source code [36]. Erlikh estimated that the cost needed to perform such activities ranges from 85% to 90% of the total cost of a software project [15].

According to Garlan [18], architectural information represent an important resource for software maintainers to aid the comprehension, the analysis and the maintenance of large and complex systems. In fact, software architectures provide *models* and *views* representing the relationships among different software components according to a particular set of concerns [40]. However, unlike classes or packages, these information do not have an explicit representation in the source code, and thus several approaches have been proposed in the literature to support software architecture *recovery* (SAR) [13]. Many of these techniques derive *architectural views* of the subject system from the source code by applying clustering analysis techniques to software artifacts, considered at different levels of abstraction (i.e., *classes* or *methods*) [13]. These abstractions represent one of the key aspect of such techniques as they allow to focus on specific maintenance tasks, providing different analysis perspectives. In fact, even if the recovery process is always expressed in terms of a clustering problem, the analysis of classes or methods leads to different SAR analysis, namely the clustering of functional modules and software clone detection respectively.

One of the typical tasks for the maintainers is to locate groups of software artifacts that deals with a specific topic, in order to modify them. For instance, a maintainer could be interested in finding all the classes that handle a given concept in the application domain, or that provide related functionality. The greater part of the approaches for architecture recovery [26, 34] applies clustering algorithms to large software systems, to partition them into meaningful subsystems. Files containing source code are placed in the same cluster only in the case they implement related functions. A number of these approaches generally attempt to discover clusters by analyzing structural dependencies between software artifacts [49, 1, 38, 5]. However, if the analysis is based on the sole structural aspect, a key source of information about the analyzed software system may be lost, i.e. the domain knowledge that developers embed in the source code lexicon. As a consequence, some efforts are being devoted to investigate the use of lexical information for software clustering [29, 8, 7, 10].

On the other hand, the software clone detection task is focused on the analysis and the identification of source code duplications. Duplicated source code is a phenomenon that occurs frequently in large software systems [3]. Reasons why programmers duplicate code are manifold. The most well known is a common bad programming practice, copying and pasting [43], that gives rise to *software clones*, or simply *clones*. These clones heavily affect the reliability and the maintainability of large software systems. The main issue in the management of clones is that errors in the original version must be fixed in every clone. The identification of clones directly depends on the degree of similarity between the two compared fragments: the less their common instructions, the bigger the effort necessary to correctly detect them as clones [43]. In fact, programmers usually adapt the copies to the new context by applying multiple modifications such as adding new statements, renaming variables, etc. In this sce-

[1] University of Naples "Federico II"- email: anna.corazza@unina.it
[2] University of Naples "Federico II"- email: sergio.dimartino@unina.it
[3] University of Naples "Federico II"- email: valerio.maggio@unina.it
[4] University of Trento- email: moschitti@disi.unitn.it
[5] University of Trento- email: passerini@disi.unitn.it
[6] University of Basilicata- email: giuseppe.scanniello@unibas.it
[7] ISTI Institute - CNR- email: fabrizio.silvestri@isti.cnr.it

nario, it could be likely that some clones are not detected, thus affecting the reliability of the system due to the presence of unfixed bugs.

From the perspective of SAR techniques, the crucial part for both the considered tasks concerns the definition of a proper similarity measure to apply in the clustering analysis, which is able to exploit the considered representation of software artifacts. To this aim, in this paper we explore the possibility of defining novel techniques for automatic software analysis that combine different methods gathered from Information Retrieval (IR), Natural Language Processing (NLP) and Machine Learning (ML) fields to automatically mine information from the source code. In particular, we investigate the application of the so-called Kernel Methods [20, 39] to define similarity measures able to exploit the structural representation of the source code. These techniques provide flexible solutions able to analyze large data set with an affordable computational efficiency. However a trade-off is imposed on their effectiveness as they solely rely on the quality of the analyzed data. To this aim, some part of our proposal will be specifically focused on the definition of a publicly available data set necessary for the assessment of the proposed approaches.

In the reminder of this paper, Section 2 provides an extensive state-of-the-art for the two considered SAR tasks, i.e., the clustering of functional modules and the software clone detection. Section 3 illustrates our proposal for automatic clone detection whereas Section 4 propose advanced machine learning methods, such as supervised clustering for SAR. Finally, Section 5 derives the conclusions.

2 State-of-the-Art in automatic SAR

A complete and extensive survey of SAR techniques is proposed by Ducasse et al. [13] where authors provide an accurate taxonomy of different approaches according to five distinct aspect, namely the *goals*, the *process*, the *inputs*, the *techniques* and the *outputs*. In this paper, we limit our analysis only to approaches and techniques for automatic SAR involving clustering analysis techniques. In particular, we focus on two different SAR subtasks, related to the clustering of functional modules (Section 2.1) and the identification of cloned code (Section 2.2).

2.1 State-of-the-art of Software Clustering methods

The definition of effective methods to automatically partition systems into meaningful subsystems, requires that several non trivial issues have to be considered [26]: *(i)* the level of granularity for the software entities to consider in the clustering; *(ii)* the information used to compare software entities, and *(iii)* the clustering algorithm to be exploited in order to group similar artifacts.

In Table 1, we summarize the state of the art regarding software clustering for the recovery of software architectures.

To better provide a detailed overview of different approaches, in the following we present the related literature with respect to the information exploited in the clustering process, namely structural information, lexical information, and their combinations.

Structural based approaches The works proposed by Wiggerts [49] and by Anquetil and Lethbridge [1] represent the first two contributions to semi-automatic approaches for the clustering of software entities. In particular, in [1] authors present a comparative study of different hierarchical clustering algorithms based on structural information. However the proposed solutions require human decisions

Table 1. Overview of architecture recovery approaches

Approach	Used Information	Clustering Algorithm	Automatic or Semi-automatic
Anquetil and Lethbridge [1]	structural	Bunch; hierarchical	semi-automatic
Mitchell and Mancoridis [38]	structural	hill climbing	automatic
Doval et al. [12]	structural	genetic algorithm	automatic
Bittencourt and Guerrero [5]	structural	edge betweenness; k-means; modul. quality; design struct. matrix	semi-automatic
Wu *et al.* [50]	structural	hierarchical; prog. compr. patterns; Bunch	semi-automatic
Tzerpos and Holt [46]	structural	hierarchical	semi-automatic
Kuhn et al. [29]	lexical	hierarchical	semi-automatic
Risi et al. [41]	lexical	k-means	automatic
Corazza et al. [8, 7, 10]	lexical	k-medoids; hierarchical	automatic
Maqbool and Babri [34]	lexical structural	hierarchical	semi-automatic
Maletic and Marcus [33]	lexical structural	minimum spanning tree	semi-automatic
Scanniello et al. [45]	lexical structural	k-means	automatic

(e.g., cutting points of the dendrograms) to get the best partition of software entities into clusters.

Maqbool and Babri in [34] highlight the features of hierarchical clustering research in the context of software architecture recovery. Special emphasis is posed on the analysis of different similarity and distance measures that could be effectively used in clustering software artifacts. The main contribution of the paper is, however, the analysis of two clustering based approaches and their experimental assessment. The discussed approaches try to reduce the number of decisions to be taken during the clustering. They also conducted an empirical evaluation of the clustering based approaches on four large software systems.

Mitchell and Mancoridis in [38] present a novel clustering algorithm, named *Bunch*. Buch produces system decompositions applying search based techniques in combination with several heuristics, such as the *coupling* and *cohesion* of produced partitions, specifically designed for the clustering of software artifacts. In particular, the coupling and the cohesion heuristics are defined in terms of *intra-* e *inter-* clusters dependencies respectively. The evaluation of the produced partitions has been conducted according to qualitative and quantitative empirical investigations. Similarly, Dove et al. [12] propose a structural approach based on genetic algorithms to group software entities in clusters.

Clustering algorithms based on structural information have been also used in the analysis of the software architecture evolution [5], [50]. Wu et al. in [50] present a comparative study of a number of clustering algorithms: *(a)* hierarchical agglomerative clustering algorithms based on the Jaccard coefficient and the single/complete linkage update rules; *(b)* an algorithm based on program comprehension patterns that tries to recover subsystems that are commonly found in manually-created decompositions of large software systems; and *(c)* a customized configuration of an algorithm implemented in Bunch [38]. Similarly, Bittencourt and Guerrero [5] present an empirical study to evaluate four widely known clustering algorithms on a number of software systems implemented in Java and C/C++. The analyzed algorithms are: Edge betweenness clustering, k-means clustering, modularization quality clustering, and design structure matrix clustering.

Lexical based approaches Software clustering approaches exploiting lexical information are based on the idea that the lexicon provided by developers in the source code represent a key source of information. In particular, such techniques mine relevant information from source code identifiers and comments based on the assumption that related artifacts are those that share the same vocabulary.

The approach proposed by Kuhn et al. [29] constitutes one of the first proposals in this direction defining an automatic technique based on the application of the Latent Semantic Indexing (LSI) method [11]. The approach is language independent and mines the lexical information gathered from source code comments. In addition, the approach enables software engineers to identify topics in the source code by means of labeling of the identified clusters.

Similarly, Risi et al. [41] propose an approach that uses the LSI and the k-means clustering algorithm to form groups of software entities that implement similar functionality. A variant based on fold-in and fold-out is introduced as well. This approach can be used to automatically recover the architectural view of a software system and provides an important contribution on the analysis of computational costs necessary to assess the validity of a clustering recovery technique.

Corazza et al. [8] propose a clustering based approach that considers the source code text as structured in different zones providing different relevance of information. In particular, the relevance of each zone is automatically weighted thanks to the definition of a probabilistic generative model and the application of the Expectation-Maximization (EM) algorithm. Related artifacts are then grouped accordingly using a customization of the k-medoids clustering algorithm. More recently the same authors propose an investigation on the effectiveness of the EM algorithm in combination with different code zones [7] and different clustering algorithms [10].

Approaches based on lexical and structural information Maletic and Marcus in [33] propose an approach based on the combination of lexical and structural information to support comprehension tasks within the maintenance and reengineering of software systems. From the lexical point of view they consider problem and development domains. On the other hand, the structural dimension refers to the actual syntactic structure of the program along with the control and dataflow that it represents. Software entities are compared using LSI, while file organization is used to get structural information. To group programs in clusters a simple graph theoretic algorithm is used. The algorithm takes as input an undirected graph (the graph obtained computing the cosine similarity of the two vector representations of all the source code documents) and then constructs a Minimal Spanning Tree (MST). Clusters are identified pruning the edges of the MST with a weight larger than a given threshold. To assess the effectiveness of the approach some case studies on a version of Mosaic are presented and discussed.

Scanniello et al. [45] present a two phase approach for recovering hierarchical software architectures of object oriented software systems. The first phase uses structural information to identify software layers [44]. To this end, a customization of the Kleinberg algorithm [24] is used. The second phase uses lexical information extracted from the source code to identify similarity among pairs of classes and then partitions each identified layer into software modules. The main limitation of this approach is that it is only suitable for software systems exhibiting a classical tiered architecture.

Table 2. Considered Clone Detection Approaches

Approach	Used Information	Technique
Ducasse et al. [14] Johnson [22]	Textual	String matching
Baker [2] Kamiya et al. [23]	Token	Pattern matching Suffix-tree matching
Yang [51] Baxter et al. [3] Koschke et al. [27] Bulychev et al. [6] Jiang et al. [21]	Syntactic	Dynamic Programming Tree Matching Suffix-tree AST Anti-unification (NLP) LSH
Komondoor and Horwitz [25] Krinke [28] Gabel et al. [17]	Structural	PDG Slicing PDG Heuristics PDG Slicing
Leitão [32] Wahler et al. [48] Corazza et al. [9] Roy and Cordy [42]	Combined	Software metrics Frequent Item-sets Tree Kernels (ML) Code Transformation and Line Comparison

2.2 State-of-the-art of Clone detection techniques

In this section we summarize research in the area of clone detection, grouping the proposals according to the features they exploit to identify similarities among software artifacts (see Table 2). Note that our goal here is not to provide an extensive analysis of the clone detection approaches presented in the literature but to provide an overview of most important techniques together with a general background on the problem, necessary to introduce the proposal presented in Section 3. An exhaustive survey of clone detection tools and techniques is provided in [43].

Textual based approaches Ducasse et al. [14] propose a language-independent approach to detect code clones, based on line-based string matching and visual presentation of the cloned code. A different approach is presented by Johnson [22] where the author applies a string matching technique based on fingerprints to identify exact repetitions of text in the source code of large software systems.

The main feature of these techniques relies in their efficiency and scalability, easily applicable to the analysis of large software systems. However, their detection capabilities are very limited and only restricted to very similar textual duplications (line by line). As a matter of fact these approaches are scarcely usable in practice.

Token based approaches Baker [2] suggests an approach to identify duplications and near-duplications (i.e., copies with slightly modifications) in large software systems. The proposed approach finds source code copies that are substantially the same except for global substitutions. Similarly, Kamiya et al. [23] use a suffix-tree matching algorithm to compute token-by-token matching among source code fragments. The authors adopt optimization techniques that mainly normalize token sequences. This is due to the fact that the underlying algorithm may be expensive when used on large software systems.

The main drawback of these approaches is that they completely disregard the syntactic structure of the analyzed source code, similarly to textual based techniques. As a consequence, these solutions may detect a large number of false clones, usually not corresponding to any actual syntactic unit.

Syntactic based approaches Syntactic based approaches exploit the information provided by Abstract Syntax Trees (AST) to identify similar code fragments. Such techniques are more robust to modifications in code fragments than textual and token based technique. However, they may possibly fail in case modifications concerns the inver-

sion or the substitution of entire code blocks: the so-called *gapped-clones* [28].

Yang [51] uses dynamic programming to find differences between two versions of the same source file. A similar approach is presented by Baxter et al. [3]. It is based on a tree matching algorithm to compare sub-trees of an AST of a given software system. On the other hand, Koschke et al. [27] describe an approach to detect clones based on suffix trees of serialized ASTs. The main contribution of this work is that software clones can be identified in linear time and space. A different approach is presented by Bulychev et al. [6], where authors propose a clone detection technique based on the *anti-unification* algorithm, widely used in Natural Language processing tasks. A novel approach for detecting similar trees has been presented by Jiang et al. [21] in their tool *Deckard*. In their approach, certain characteristic vectors are computed to approximate the structure of ASTs in a Euclidean space. Locality sensitive hashing (LSH) is then used to cluster similar vectors using the Euclidean distance metric.

Structural based approaches Structural based approaches gather information from control and dependency graphs to identify clones. In particular these techniques apply algorithms to identify isomorphic sub-graphs within a graph built considering control and data flow dependencies (i.e., the program dependence graphs, PDG) of the software system to analyze.

Komondoor and Horwitz [25] propose an approach based on program slicing techniques, applied on PDGs. On the other hand, Krinke [28] propose a heuristic based approach to identify isomorphic sub-graphs. More recently, Gabel et al. [17] propose a PDG-based technique that maps slices of PDGs to syntax subtrees and applies the Deckard clone detection tool [21].

The main advantage of these techniques is that they do not depend on the particular textual representation of the code, allowing to detect also functional duplications, in addition to the textual based ones considered by previous approaches. However the identification of isomorphic sub-graphs is a NP-hard problem and only approximated solutions may be provided.

Combined approaches In the literature techniques that combine different artifacts representation have been defined. For example, Leitão [32] combines syntactic and semantic techniques using functions that consider various aspects of software systems (e.g., similar call sub-graphs, commutative operators, user-defined equivalences).

Differently, Wahler et al. [48] present an approach based on a data mining technique to detect clones. This approach uses the concept of frequent item-sets on the XML representation of the software system to be analyzed. Morevover, Corazza et al [9] propose an approach for software clone detection based on the application of Tree Kernel functions to compare source code fragment according to their syntactic structure and the associated lexical information. The effectiveness of the approach has been assessed in comparative experiments with another pure syntactic based approach. Finally, Roy and Cordy [42] present an approach based on source transformations and text line comparison to find clones.

3 Clone Detection

As briefly introduced in Section 1, the definition of clones [3] states that two code fragments form a clone if they are similar according to some similarity function. However, such similarity can be based either on their program text, or on their functionality (independent of their text) [43].

In the literature, all these kinds of code similarities correspond to the following taxonomy of clones [43]:

Type 1 : An exact copy of consecutive code fragments without modifications (except for white spaces and comments).

Type 2 : Syntactically identical fragments except for variations in identifiers, literals, and variable types in addition to Type-1s variations;

Type 3 : Copied fragments with further modifications such as changed, added, or deleted statements in addition to Type-2s variations.

Type 4 : Code fragments that perform similar functionality but are implemented by different syntactic variants.

According to this classification, only Type 1 clones are represented by exactly the same set of instructions, while the other three types involve lexical and syntactic variations between the two fragments. As a consequence, an effective similarity measure has to combine both the syntactic and lexical information. Thus, the input representation is the first crucial point to consider when designing a machine learning based clone detector. In addition, annotated data are needed to train the considered techniques. In the rest of this section we discuss these two points in depth and also the assessment protocol.

3.1 Code Similarities and Kernel Methods

Kernel methods [20] have shown to be effective in approaches considering the similarity between complex input structures. In particular, tree kernels have been widely used in fields including natural language processing [39] and bioinformatics [47], applied to parse and phylogenetic trees respectively. Thus, considering the source code, it seems rather intuitive to apply tree kernels to Abstract Syntax Trees (ASTs) of the source code. However, as the sole syntactic information is not sufficient to decide whether two code fragments are clones or not, we enriched the information present in each (internal) node of the AST by annotating them with the *lexemes* gathered from the corresponding leaf nodes. Preliminary results are reported in [9].

Such approach can not be applied in detecting Type 4 clones as their similarity is independent of the corresponding program text. As in this case the information about the *program behavior* becomes relevant for the identification of clones, we consider the source code as represented by the Program Dependency Graph (PDG) onto which we apply a graph kernel method to detect similar subgraphs. A PDG is a representation of a function in which nodes correspond to simple statements and control flow predicates, and edges encode data and control dependencies [17]. However, the main drawback of these kernels with respect to the previous ones regards the computational effort needed in performing the similarity evaluation. As a consequence, it is necessary to find a good trade-off between such computational cost and the information taken into account in the comparison of PDGs. To this aim, we consider Weighted Decomposition Kernels (WDK) [37] as they enable to define criteria to reduce the total number of comparisons. We generate the PDGs for source code written in the C language by using the Code Surfer tool.[8]

3.2 Training data

A crucial problem in adopting machine learning approaches regards the necessity to arrange two different set of annotated data, namely the training and the assessment set respectively. Unfortunately, this

[8] `http://www.grammatech.com`

kind of data set are harder to get in case of clone detection as the manual annotation process is too expensive for large systems. Therefore, in order to alleviate such problem, the generally adopted solution consider the definition of a *pooling process* where the manual check is performed on a limited set of data gathered from different clone detection tools. An example of such process is provided in [4]. However, the effect of such procedure is that there is no guarantee of completeness and only a precision measure can be evaluated, by manually checking the output of the system. Moreover, the so obtained data are not effective for training, as they tend to simulate the system used to generate them, rather than addressing the actual clones.

Given such situation, only unsupervised machine learning, i.e. clustering, can be proposed. However, clustering can not be expected to be accurate enough for this application, as only the similarity definition can be exploited to guide the algorithm. As an alternative, we explore the use of simulated data to build a training set and apply supervision to detect the clones. The data set is produced as a variation of a given software project where clones are modified and injected by following predefined probability distributions. In this way, we can control the quality of the training set, without imposing any restriction on its size. A classifier employing the necessary kernels can then be trained to filter the data produced by the clustering step.

3.3 Parameter Setting and Experimental assessment

To understand the effectiveness of the proposal, an extensive experimental assessment is needed. In particular, a lot of parameters need to be set regarding the different input representations, the probability distributions used in the training set generation and the kernel parameters. Given this scenario a k-fold cross validation protocol seems to be appropriate. As this preprocessing step is performed on a fully labeled data set, precision, recall and F-measure are used to estimate the effectiveness of the considered configurations.

Once the best configuration has been identified, it will be used to replicate the few available datasets in the literature, in order to compare the proposal with the state-of-the-art. As previously discussed, since not all the gold positives are labelled, both the recall and F-measure are underestimated.

4 Architecture Recovery

Recovering the architecture of a software system requires to group together portions of code jointly performing a certain function and identifying the structural organization of these functional modules. The problem can be naturally formalized in terms of hierarchical clustering (see Section 2). Within such framework, we aim at improving over existing approaches by leveraging over the following aspects:

1) exploiting the rich structure characterizing software projects, in terms of hierarchical structuring of the code and relationships given by e.g. function calls. As already discussed for the clone detection problem (see Section 3), kernel methods are a natural candidate for learning problems involving richly structured objects. We will thus develop structured kernels on AST and PDG testing them in terms of capacity to recover similarity between related fragments. We will also employ kernel learning approaches [19], where the similarity measure is not fully specified a-priori, but is learned from examples as a combination of similarity patterns. Logic kernels [30] are particularly promising in this context, as they allow to encode arbitrary domain knowledge concerning relationships between code fragments from which similarity measures are to be learned.

2) exploiting all available information, in terms of existing full or partial architecture documentation, in order to improve performance of predictive algorithms. The few existing fully documented software systems can be used as gold standards representing how a correct architecture recovery should appear. The problem can be framed in terms of supervised clustering [16]: gold standards are examples of inputs (the code) and desired outputs (its architectural organization), used to train a predictive machine trying to approximate the desired output when fed with the code. In so doing, the predictor adapts the similarity measure to improve the approximation. When presented with a new piece of code, the trained machine clusters it using the learned similarity measure. We plan to extend this supervised clustering paradigm, mostly developed for flat clustering, to produce a hierarchy of clusters. Partial architecture documentation can also be used in a similar fashion by turning the supervised learning problem into a semi-supervised one: the algorithm is trained to output a full architectural representation which is consistent with the partial information available, possibly accounting for inconsistencies due to labeling errors or ambiguity.

5 Conclusions

In this paper, we presented an extensive related work in the field of automatic SAR. We also illustrated our experience and proposal for using advanced Machine Learning, Natural Language Processing and Information Retrieval for automatic SAR.

In particular, we discussed innovative approaches, i.e., kernel methods, to detect the similarity between complex input structures such as source code represented in terms of Abstract Syntax Trees. We also proposed hybrid methods exploiting Program dependency graphs in machine learning algorithms (MLA) based on graph kernels. Since MLA require training data, we outlined possible approaches to gather it, ranging from manual annotation to artificial data generation. In this respect, we also discussed innovative MLA for learning object similarities, which are able to integrate background knowledge by means of logic predicates.

Finally, we proposed new supervised clustering methods which can automatically learn how to recover software architectures.

REFERENCES

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

[2] B. Baker, 'On finding duplication and near-duplication in large software systems', in *IEEE Proceedings of the Working Conference on Reverse Engineering*, (1995).

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier, 'Clone detection using abstract syntax trees', in *Proceedings of the International Conference on Software Maintenance*, pp. 368–377. IEEE Press, (1998).

[4] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo, 'Comparison and evaluation of clone detection tools', *IEEE Trans. Software Eng.*, **33**(9), 577–591, (2007).

[5] R. A. Bittencourt and D. D. S. Guerrero, 'Comparison of graph clustering algorithms for recovering software architecture module views', in *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 251–254, Washington, DC, USA, (2009). IEEE Computer Society.

[6] Peter Bulychev and Marius Minea, 'Duplicate code detection using anti-unification.', in *Spring/Summer Young Researcher's Colloquium*, (2008).

[7] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, 'Investigating the use of lexical information for software system clustering', in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pp. 35–44, Washington, DC, USA, (2011). IEEE Computer Society.

[8] A. Corazza, S. Di Martino, and G. Scanniello, 'A probabilistic based approach towards software system clustering', *Proceedings of the European Conference on Software Maintenance and Reengineering*, 88–96, (2010).

[9] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello, 'A tree kernel based approach for clone detection', in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pp. 1–5, Washington, DC, USA, (2010). IEEE Computer Society.

[10] Anna Corazza, Sergio Martino, Valerio Maggio, and Giuseppe Scanniello, 'Combining machine learning and information retrieval techniques for software clustering', in *Eternal Systems*, eds., Alessandro Moschitti and Riccardo Scandariato, volume 255 of *Communications in Computer and Information Science*, 42–60, Springer Berlin Heidelberg, (2012).

[11] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, 'Indexing by latent semantic analysis', *Journal of the American Society of Information Science*, **41**(6), 391–407, (1990).

[12] D. Doval, S. Mancoridis, and B. S. Mitchell, 'Automatic clustering of software systems using a genetic algorithm', in *Proceedings of the Software Technology and Engineering Practice*, pp. 73–82, Washington, DC, USA, (1999). IEEE Computer Society.

[13] S. Ducasse and D. Pollet, 'Software architecture reconstruction: A process-oriented taxonomy', *Software Engineering, IEEE Transactions on*, **35**(4), 573 –591, (july-aug. 2009).

[14] S. Ducasse, M. Rieger, and S. Demeyer, 'A language independent approach for detecting duplicated code', in *Proceedings of the International Conference on Software Maintenance*, pp. 109–118, (1999).

[15] Len Erlikh, 'Leveraging legacy system dollars for e-business', *IT Professional*, **2**, 17–23, (2000).

[16] Thomas Finley and Thorsten Joachims, 'Supervised clustering with support vector machines', in *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pp. 217–224, New York, NY, USA, (2005). ACM.

[17] Mark Gabel, Lingxiao Jiang, and Zhendong Su, 'Scalable detection of semantic clones', in *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pp. 321–330, New York, NY, USA, (2008). ACM.

[18] David Garlan, 'Software architecture: a roadmap', in *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pp. 91–101, New York, NY, USA, (2000). ACM.

[19] Mehmet Gönen and Ethem Alpaydin, 'Multiple kernel learning algorithms', *J. Mach. Learn. Res.*, **999999**, 2211–2268, (July 2011).

[20] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola, 'Kernel methods in machine learning', *Annals of Statistics*, **36**(3), 1171–1220, (2008).

[21] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu, 'Deckard: Scalable and accurate tree-based detection of code clones', in *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pp. 96–105, Washington, DC, USA, (2007). IEEE Computer Society.

[22] J. Howard Johnson, 'Identifying redundancy in source code using fingerprints', in *Proc. Conf. Centre for Advanced Studies on Collaborative research (CASCON)*, pp. 171–183. IBM Press, (1993).

[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, 'Ccfinder: A multilinguistic token-based code clone detection system for large scale source code.', *IEEE Trans. Software Eng.*, **28**(7), 654–670, (2002).

[24] J. M. Kleinberg, 'Authoritative sources in a hyperlinked environment', *Journal of the ACM*, **46**, 604–632, (September 1999).

[25] R. Komondoor and S. Horwitz, 'Using slicing to identify duplication in source code', in *Proceedings of the International Symposium on Static Analysis*, pp. 40–56, (July 2001).

[26] R. Koschke, 'Atomic architectural component recovery for program understanding and evolution', *Softwaretechnik-Trends*, (2000).

[27] Rainer Koschke, Raimar Falke, and Pierre Frenzel, 'Clone detection using abstract syntax suffix trees', in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 253–262, Washington, DC, USA, (2006). IEEE Computer Society.

[28] Jens Krinke, 'Identifying Similar Code with Program Dependence Graphs', in *Proc. Working Conf. Reverse Engineering (WCRE)*, pp. 301–309. IEEE Computer Society Press, (2001).

[29] A. Kuhn, S. Ducasse, and T. Gírba, 'Semantic clustering: Identifying topics in source code', *Information and Software Technology*, **49**, 230–243, (March 2007).

[30] Niels Landwehr, Andrea Passerini, Luc Raedt, and Paolo Frasconi, 'Fast learning of relational kernels', *Mach. Learn.*, **78**(3), 305–342, (March 2010).

[31] Meir M. Lehman, 'Programs, life cycles, and laws of software evolution', *Proc. IEEE*, **68**(9), 1060–1076, (September 1980).

[32] António Menezes Leitão, 'Detection of redundant code using r^2d^2', *Software Quality Journal*, **12**(4), 361–382, (2004).

[33] J. I. Maletic and A. Marcus, 'Supporting program comprehension using semantic and structural information', in *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pp. 103–112, Washington, DC, USA, (2001). IEEE Computer Society.

[34] O. Maqbool and H. Babri, 'Hierarchical clustering for software architecture recovery', *IEEE Transactions on Software Engineering*, **33**(11), 759–780, (2007).

[35] Andrian Marcus and Jonathan I. Maletic, 'Identification of high-level concept clones in source code', in *ASE*, pp. 107–114, (2001).

[36] A. Von Mayrhauser, 'Program comprehension during software maintenance and evolution', *IEEE Computer*, **28**, 44–55, (1995).

[37] Sauro Menchetti, Fabrizio Costa, and Paolo Frasconi, 'Weighted decomposition kernels', in *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pp. 585–592, New York, NY, USA, (2005). ACM.

[38] B. S. Mitchell and S. Mancoridis, 'On the automatic modularization of software systems using the bunch tool', *IEEE Transactions on Software Engineering*, **32**, 193–208, (March 2006).

[39] Alessandro Moschitti, Roberto Basili, and Daniele Pighin, 'Tree Kernels for Semantic Role Labeling', in *Computational Linguistics*, pp. 193–224, Cambridge, MA, USACambridge, MA, USA, (2008). MIT Press.

[40] Lih ren Jen and Yuh jye Lee, 'Working group. ieee recommended practice for architectural description of software-intensive systems', *IEEE Architecture*, 1471–2000, (2000).

[41] Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora, 'Using fold-in and fold-out in the architecture recovery of software systems', *Formal Asp. Comput.*, **24**(3), 307–330, (2012).

[42] Chanchal Kumar Roy and James R. Cordy, 'Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization', in *ICPC*, pp. 172–181, (2008).

[43] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke, 'Comparison and evaluation of code clone detection techniques and tools: A qualitative approach', *Sci. Comput. Program.*, **74**(7), 470–495, (2009).

[44] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico, 'Architectural layer recovery for software system understanding and evolution', *Software Practice and Experience*, **40**, 897–916, (September 2010).

[45] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico, 'Using the kleinberg algorithm and vector space model for software system clustering', in *Proceedings of the IEEE 18th International Conference on Program Comprehension*, ICPC'10, pp. 180–189, Washington, DC, USA, (2010). IEEE Computer Society.

[46] V. Tzerpos and R. C. Holt, 'On the stability of software clustering algorithms', in *Proceedings of the 8th International Workshop on Program Comprehension*, pp. 211–218, (2000).

[47] Jean-Philippe Vert, 'A Tree Kernel to analyse phylogenetic profiles', *Bioinformatics*, **18**(suppl_1), S276–284, (2002).

[48] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer, 'Clone detection in source code by frequent itemset techniques', in *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pp. 128–135, Washington, DC, USA, (2004). IEEE Computer Society.

[49] T. A. Wiggerts, 'Using clustering algorithms in legacy systems remodularization', in *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pp. 33–43, Washington, DC, USA, (1997). IEEE Computer Society.

[50] A. E. Wu, J. Hassan and R. C. Holt, 'Comparison of clustering algotithms in the context of software evolution', in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 525–535. IEEE Computer Society, (2005).

[51] Wuu Yang, 'Identifying syntactic differences between two programs', *Software - Practice and Experience*, **21**(7), 739–755, (July 1991).