The Thirtieth AAAI Conference on Innovative Applications of Artificial Intelligence (IAAI-18)

Discovering Program Topoi through Clustering

Carlo Ieva,[1,2] Arnaud Gotlieb,[1] Souhila Kaci,[2] Nadjib Lazaar[2]
[1]SIMULA Research Laboratory, Oslo, Norway
[2]LIRMM, University of Montpellier, France
{carlo, arnaud}@simula.no, {souhila.kaci, nadjib.lazaar}@lirmm.fr

Abstract

Understanding source code of large open-source software projects is very challenging when there is only little documentation. New developers face the task of classifying a huge number of files and functions without any help. This paper documents a novel approach to this problem, called FEAT, that automatically extracts *topoi* from source code by using hierarchical agglomerative clustering. Program topoi summarize the main capabilities of a software system by presenting to developers clustered lists of functions together with an index of their relevant words. The clustering method used in FEAT exploits a new hybrid distance which combines both textual and structural elements automatically extracted from source code and comments. The experimental evaluation of FEAT shows that this approach is suitable to understand open-source software projects of size approaching 2,000 functions and 150 files, which opens the door for its deployment in the open-source community.

Introduction

The development of large-scale open-source projects involves many distinct developers contributing to the creation of large code repositories. As an example, the July 2017 release of the Linux kernel (v. 4.12), amounting to almost 20 MLOC (lines of code), requested the effort of 329 developers marking a growth of 1 MLOC respect to its predecessor. This figures make clear that, whenever a new developer wants to become a contributor, he/she has to face the problem of understanding a huge amount of code, organized as an unclassified set of files and functions.

Organizing the code in a more abstract way, closer to humans is an attempt that received interest from the Software Engineering community. Unfortunately, there is no recognized recipe or tool that can concretely provide any help in dealing with large software repositories.

We propose an effective solution to this problem by automatically extracting program topoi which are ordered lists of function names associated with an index of relevant words. How does the sorting take place? FEAT does not consider all functions as equal: some of them are considered like a gateway to the implementation of high-level, user-observable capabilities of a program. We call these special functions *entry points* and the sorting criterion is based on the distance between common functions and entry points. The whole method can be summarized according to its three main steps:

1. *Preprocessing.* Source code, with its comments, is parsed generating for every code unit (either functions in procedural languages or methods in object oriented ones) a corresponding textual document. Additionally, a graph representation of the caller-callee relationship (call graph) is also created in this step.
2. *Clustering.* Code units are grouped together by means of hierarchical agglomerative clustering (HAC).
3. *Entry Point Selection.* Within the context of every cluster, code units are ranked and those placed in higher positions will constitute a program topos.

The contribution of this paper is three-fold:

1. FEAT is a novel, fully automated approach for program topoi extraction based on clustering units directly from source code. To exploit HAC, we propose an original hybrid distance combining structural and semantic elements of source code. HAC requires the selection of a partition among all those produced along the clustering process, our approach makes use of a hybrid criterion based on *graph modularity* (Donetti and Muñoz 2004) and textual coherence (Foltz, Kintsch, and Landauer 1998) to automatically select the appropriate one;
2. Clusters of code units need to be analyzed to extract program topoi. We define a set of structural elements obtained from source code and use them to create an alternative representation of clusters of code units. Principal Component Analysis (PCA), with its capability of dealing with multi-dimensional data, provides us a way to measure the distance of code units respect to an ideal entry point. This distance is the basis for code units' ranking shown to end-users.
3. We implemented FEAT on top of a general-purpose software analysis platform and performed an experimental study over some open-source software projects. During the evaluation we analyzed FEAT under several perspectives: the clustering step, effectiveness in topoi discovery and, scalability of the approach.

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Related work

Our research spans the domain of program understanding focusing mainly on *feature extraction* (Rubin and Chechik 2013; Chen and Rajlich 2000; Marcus and Haiduc 2013) that aims to automatically discover the main characteristics of a software system by analyzing source code or other artifacts. It must be distinguished from *feature location*, whose objective is to locate where and how these characteristics are implemented (Rubin and Chechik 2013). Feature location requires the user to provide an input query where the searched characteristic is already known, while feature extraction tries to automatically discover these characteristics. Since several years, software repository mining is considered mainstream in feature extraction. However, we can distinguish between software-repository mining approaches dealing with software documentation only and, those dealing with source code only.

Mining Software Documentation. In (Dumitru et al.), both text-mining techniques and flat clustering are used to extract feature descriptors from user requirements kept in software repositories. By combining association-rules mining and k-Nearest-Neighbour, the proposed approach makes recommendations on other feature descriptors to strengthen an initial profile. More recently (McBurney, Liu, and McMillan 2016) presented four automatic generators of list of features for software projects, which select English sentences that summarize features from the project documentation.

FEAT has two distinguishing elements w.r.t. these techniques. Firstly, FEAT deals with both software documentation and source code by applying at the same time code and text analysis techniques. Secondly, FEAT uses HAC assuming that software functions are organized according to a certain (hidden) structure that can be automatically discovered.

Mining Source Code. (Linstead et al. 2007) proposes dedicated probabilistic models based on code analysis using *Latent Dirichlet Allocation* to discover features under the form of so-called *topics* (main functions in code). (McMillan et al. 2012) present a source-code recommendation system for software reuse. Based on a feature model (a notion used in product-line engineering and software variability modeling), the proposed system tries to match the description with relevant features in order to recommends the reuse of existing source code from open-source repositories. (Abebe and Tonella 2015) proposes natural language parsing to automatically extract an ontology from source code. Starting from a lightweight ontology (a.k.a, *concept map*), the authors develop a more formal ontology based on axioms.

Conversely, FEAT is fully automated and does not require any form of training or any additional modeling activity (e.g., feature modeling). It uses an unsupervised machine learning technique making its usage and application much simpler.

(Kuhn, Ducasse, and Gîrba 2007) use clustering and LSI (Latent Semantic Indexing) to assess the similarity between source artifacts and to create clusters according to their similarity. The most relevant terms extracted from the LSI analysis are reused for labeling the clusters. FEAT instead exploits both text mining and code structure analysis to guide the creation of clusters.

We believe that there are several differences between FEAT and the above mentioned approaches and that FEAT fosters program understanding by creating a novel hybrid approach blending structural and semantic aspects of source code.

Background

Software Clustering

Software clustering methodologies create group of entities, such as classes, functions, etc. of a software system in order to ease the process of understanding the high-level structure of a large and complex software system (Shtern and Tzerpos 2012). The basis for any cluster analysis to group entities is their set of attributes.

The application of clustering to a software system requires the identification of the entities which are the object of the grouping. Several artifacts can be chosen but the most popular one is source code (Mitchell 2003). The selection of entities is affected by the objective of the method. For program restructuring at a fine-grained level, function call statements are chosen as entities (Xu et al. 2004) while for design recovery problems (Andritsos and Tzerpos 2005) entities are often software modules but also classes or routines.

Extracting facts from source code can be done following two different conceptual approaches: *structural and semantic*. Structure-based, approaches rely on static relationships among entities: variable references, procedure calls, inheritance etc. Semantic approaches take into account the domain knowledge information contained in source code and extracted from comments and identifier names (Kuhn, Ducasse, and Gîrba 2005). The software clustering community widely adopts structure-based approaches but it has to be noted that the output produced by semantic approaches tends to be more meaningful. That is why some try to combine the strengths of both methods like (Tzerpos and Holt 2000).

The actual cluster creation is accomplished through a clustering algorithm. Clustering is the most common form of unsupervised learning and the key input to a clustering algorithm is the distance measure. Different distance measures give rise to different clusterings.

There are two categories of hierarchical algorithms: agglomerative (bottom-up) and divisive (top down). In software clustering, according to (Kaufman and Rousseeuw 1990), divisive algorithms offer an advantage over agglomerative ones because users are mostly interested in the structure revealed by the large clusters created during the early stages of the process. On the other hand, the way agglomerative clustering proceeds towards large clusters may be affected by unfortunate decisions made in the first steps. Agglomerative hierarchical clustering are most widely used however. This is because it is infeasible to consider all possible divisions of the first large clusters (Wiggerts 1997).

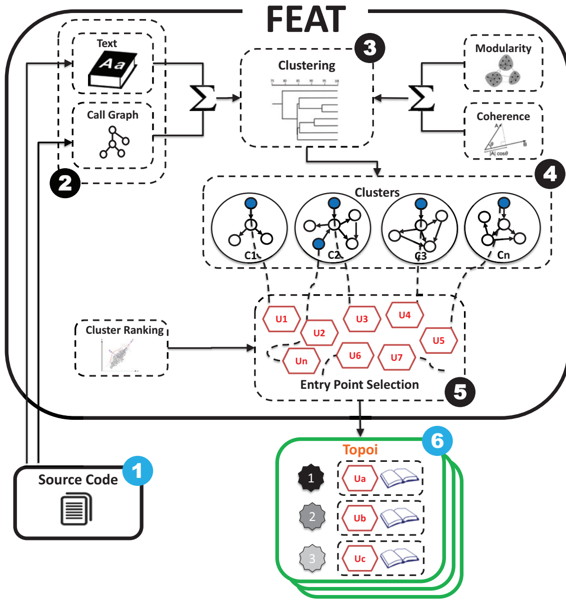

Figure 1: FEAT process's overview

Call Graph

A *Call Graph* (CG) is a convenient way of representing the unit caller-callee relationship in a software program. Given a program P composed of a collection of units $\{u_i\}_{i \in 1 \ldots n}$, the CG of P is a graph where each node is associated with a unit u_i and there is an arc from u_i to u_j if and only if u_j is called by u_i at some location in the source code. Note that even though u_i may call u_j several times, there is only a single arc from u_i to u_j in CG.

FEAT Approach

Understanding a software system through its source code can be pursued following two conceptual approaches: structural and semantic. Structure-based approaches focus on the static relationships among entities while semantic ones include all aspects of a system's domain knowledge that can be obtained from comments and identifier names (Shtern and Tzerpos 2012). Discovering the main capabilities of a software system can benefit from structural information that can be used to identify a capability as a set of structurally *close* code units contributing to its implementation. On the other hand, under the semantic view, those parts of a system showing commonalities in terms of natural language words can be considered part of a system's capability as well.

Structural and semantic approaches convey two different, both valuable, perspectives of a system. FEAT combines them in order to achieve a more accurate picture. FEAT, whose overview is depicted in Fig.1, is based on a three steps process: preprocessing (box noted 2), clustering (3) and, entry point selection (5). The input to FEAT is a software system SW considered as its source code and comments (1). In the preprocessing step (2) FEAT parses source code and comments creating a representation of SW which supports the twofold assumption lying behind the approach. Hence, if SW is a software system counting n code units its representation can be defined through the call graph of SW and the set of unit documents $\mathcal{D}$ as:

Definition 1. (FEAT model) SW $\triangleq \langle \mathrm{CG}, \mathcal{D} \rangle$

The first element (structural) of the pair is the call graph $\mathrm{CG} = (\mathcal{U}, \mathcal{E})$ where $\mathcal{U} = \{u_1, u_2, \cdots, u_n\}$ is the set of units and $\mathcal{E}$ the set of edges representing the caller-callee relationships. The second one (semantic) is the set of *unit-documents*: $\mathcal{D} = \{\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_n\}$.

The creation of the semantic part of FEAT requires that, for every unit, we extract the following elements: code unit names, variable names, string literals [1] and, comments. All these elements contribute to the creation of a set of textual documents; one for each unit. This preliminary text, where the original words' order is ignored (bag of words), undergoes several transformations (tokenization, stop words removal, stemming, weighting) producing a *unit-document* noted as $\mathbf{d}_u = [w_1, w_2, \cdots, w_m] \in \mathbb{R}^m$ where w_i is the weight associated to the *ith*-word.

Distance between Code Units

The key input to a clustering algorithm is the distance measure allowing to group data points together. In FEAT we have two distances, one for each of the two components included in Def.1. Let us begin with the distance over the structural part of SW.

Given the (undirected) call graph CG, the distance between two code units u_a and u_b is computed by using the length of a shortest path (noted as $|\pi(u_a, u_b)| = k$) between them:

$$\mathrm{d}_{\mathrm{CG}}(u_a, u_b) = \begin{cases} 0 & \text{if } u_a = u_b \\ \frac{1-\lambda}{1-\lambda^D} \sum_{i=0}^{k-1} \lambda^i & \text{if } |\pi(u_a, u_b)| = k \\ 1 & \text{if } |\pi(u_a, u_b)| = \infty \end{cases} \quad (1)$$

where D is the call graph diameter (the length of the longest shortest-path) and $\lambda > 1$ is a parameter used to provide an exponential growing of the distance.

The distance over the semantic part of the system's representation is computed through the angular distance defined as:

$$\mathrm{d}_{\mathcal{D}}(u_a, u_b) = \frac{2}{\pi} \arccos\left(\frac{\mathbf{d}_{u_a} \cdot \mathbf{d}_{u_b}}{\|\mathbf{d}_{u_a}\| \|\mathbf{d}_{u_b}\|} \right) \quad (2)$$

Hybrid distance

Both $\mathrm{d}_{\mathcal{D}}$ and d_{CG} are proper distance measures satisfying the three axioms: symmetry, positive definiteness and, triangular inequality. On their basis we present a novel hybrid distance with the objective of producing clusters whose elements show high internal cohesion under both perspectives: structural and semantic. This combination can mitigate some unwanted effects that may occur if we use only one distance. For instance, two units sharing many words, but not connected in the call graph (or very far from each other), would

[1] Quoted sequence of characters representing a string value i.e. `x="foo"`

be evaluated with high similarity if only $\mathrm{d}_{\mathcal{D}}$ be used, while, without any kind of structural relationship, they cannot belong to the same feature. Similarly, two close units in the call graph, but without any word in common, should not be clustered together because we assume that elements of a capability should share a common vocabulary.

The hybrid distance is defined as a linear combination of $\mathrm{d}_{\mathcal{D}}$ and d_{CG} using a real number α ranging in $[0, 1]$:

$$\mathrm{d}_{\mathrm{FEAT}}(u_a, u_b) = \alpha \mathrm{d}_{\mathcal{D}}(u_a, u_b) + (1 - \alpha)\mathrm{d}_{\mathrm{CG}}(u_a, u_b) \quad (3)$$

The external parameter α is used to tune the impact of one distance value over the other. The choice of a value for α depends on some characteristics of the code under analysis like the quality of comments, naming conventions etc. More details about this will be provided in the experimental section.

Distance over Clusters

In order to merge clusters, HAC requires to compute the distance between sets of units. In FEAT we represent a set of units as its centroid hence, the distance between two clusters corresponds to the distance of their centroids. Following the concept of a hybrid representation of structural and semantic elements we define a hybrid centroid distance. Unit documents lie in a Euclidean space then the centroid of a cluster $\mathcal{C}$ is $\mu_{\mathcal{D}}(\mathcal{C}) = \frac{\mathbf{d}_1 + \mathbf{d}_2 + \cdots + \mathbf{d}_n}{|\mathcal{C}|}$. Instead, the structural part of $\mathcal{C}$ is represented as a *graph medoid*. Medoids are representatives of a discrete set of elements and in FEAT they are defined as:

Definition 2. (Graph Medoid) Let $\mathrm{CG} = (\mathcal{U}, \mathcal{E})$ be a call graph, $\mathcal{C} = \{u_1, u_2, \ldots, u_m\}$ a cluster, $|\pi(u_a, u_b)|$ a shortest path between u_a and u_b, $\sigma_{\mathcal{C}}(u) = \sum_{\forall u_i \in \mathcal{C}} |\pi(u_i, u)|$, $\mathcal{M} = \{u \mid \arg\min_{u \in \mathcal{U}} \{\sigma_{\mathcal{C}}(u)\}\}$ and, $\overline{s} = \frac{1}{|\mathcal{C}|} \min_{u \in \mathcal{U}} \{\sigma_{\mathcal{C}}(u)\}$ then the graph medoid of $\mathcal{C}$ is:

$$\mu_{\mathrm{CG}}(\mathcal{C}) = \arg\min_{u \in \mathcal{M}} \left(\sum_{\forall u_i \in \mathcal{C}} (|\pi(u_i, u)| - \overline{s})^2 \right)$$

In other words, the graph medoid of $\mathcal{C}$ is the unit $u \in \mathcal{U}$ lying at the most *central* position w.r.t. all units in $\mathcal{C}$.
Finally, the hybrid distance between two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ is:

$$\begin{aligned} \mathrm{d}_{\mathrm{FEAT}}(\mathcal{C}_i, \mathcal{C}_j) = \alpha \mathrm{d}_{\mathcal{D}} \left(\mu_{\mathcal{D}}(\mathcal{C}_i), \mu_{\mathcal{D}}(\mathcal{C}_j) \right) \\ + (1 - \alpha)\mathrm{d}_{\mathrm{CG}}(\mu_{\mathrm{CG}}(\mathcal{C}_i), \mu_{\mathrm{CG}}(\mathcal{C}_j)) \end{aligned} \quad (4)$$

Eq.3 can now be seen as a special case of Eq.4 occurring when $|\mathcal{C}_i| = |\mathcal{C}_j| = 1$.

Cutting Criterion

HAC does not require a prespecified number of clusters, it creates a hierarchy of them through an iterative process producing a new partition of clusters at every merging step. How to choose a partition $\mathcal{P} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ among the $n - 1$ possible ones (where n is the number of data points) depends on the objectives of the domain at hand (Manning, Raghavan, and Schütze 2008). Following the idea of a combination of structural and semantic aspects of source code, we introduce our hybrid cutting criterion.

Given a partition of vertices of a graph, modularity (Aaron, Newman, and Moore 2004) reflects the concentration of edges within clusters compared with a random distribution of links between all nodes regardless of clusters. We use modularity to measure the division of a call graph in clusters having high inter-cluster sparsity and high intra-cluster density. Modularity is defined as:

$$\mathrm{Q}(\mathcal{P}) = \frac{1}{2m} \sum_{i,j} \left(A_{i,j} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j) \quad (5)$$

where A is the adjacency matrix[2], k_i (resp. k_j) is the degree of node i (resp. j), c_i (resp. c_j) is the cluster of i (resp. j), and m is the total number of edges. Function δ is the Kronecker delta: $\delta(c_i, c_j) = 1$ iff $c_i = c_j$ (nodes i,j are in the same cluster), 0 otherwise. High values of modularity (knowing that $\mathrm{Q} \in [-\frac{1}{2}, 1]$) correspond to interesting partitions of a call graph.

A measure adopted in natural language processing (NLP), used for assessing how similar are the segments of a text, is *coherence* (Foltz, Kintsch, and Landauer 1998). Coherence is based on the measure of words overlapping. We consider all unit-documents belonging to a cluster as sections of a whole text. Developers, within the context of the units participating in the implementation of a system capability, use a consistent language revealed through the choice of names for variables, the text in comments, etc. So, while looking at clusters as they were textual documents, we want to find the partition showing the highest coherence defined as:

$$\mathrm{H}(\mathcal{P}) = \sum_{\forall \mathcal{C} \in \mathcal{P}} \left(1 - \frac{2}{|\mathcal{C}|(|\mathcal{C}| - 1)} \sum_{k=1}^{|\mathcal{C}|} \sum_{j=k+1}^{|\mathcal{C}|} \mathrm{d}_{\mathcal{D}}(u_k, u_j) \right) \quad (6)$$

Finally, we can define a new hybrid measure as the combination of normalized coherence and modularity:

$$\mathrm{T}_{\mathrm{FEAT}}(\mathcal{P}) = \alpha \frac{\mathrm{H}(\mathcal{P})}{|\mathcal{P}|} + (1 - \alpha) \frac{2\mathrm{Q}(\mathcal{P}) + 1}{3} \quad (7)$$

The cutting criterion is then determined as the HAC's iteration where the maximum value of $\mathrm{T}_{\mathrm{FEAT}}$ is reached.

Algorithm 1 presents our clustering step with a hybrid representation of units. We use the *priority queue* version of HAC having a time complexity of $\Theta(n^2 \log(n))$. Alg.1 starts by considering each unit as a cluster at lines 4-6. It computes the pairwise distances between clusters at lines 7-8. Then, it iteratively merges pair of clusters according to a minimal $\mathrm{d}_{\mathrm{FEAT}}$ distance value (lines 11-12) until either a partition is reduced to one cluster or the cutting criterion is reached (i.e., $\mathrm{T}_{\mathrm{FEAT}}$ value cannot be improved) (line 9). At each iteration, the algorithm updates the pairwise distances Δ of the new partition (lines 14-15). At the end, the algorithm returns a partition $\mathcal{P}$ of m clusters.

[2] $A_{ij} = 1$ if there exists an edge between vertices i and i and $A_{ij} = 0$ otherwise

Algorithm 1: Priority Queue HAC with hybrid cutting criterion

```
1  In U = {u_1, ..., u_n}: n units; α ∈ [0,1], τ < 0;
2  Out P = {C_1, ..., C_m}: partition of m clusters;
3  P ← ∅; Δ ← ∅; ⟨y_p, y_c⟩ ← ⟨1,0⟩;
4  foreach u_i ∈ U do
5  |  C_i ← {u_i};
6  |  P ← P ∪ C_i;
7  foreach C_i, C_j ∈ P : i < j do
8  |  Δ ← Δ ∪ d_FEAT(C_i, C_j, α);
9  while (|P| ≠ 1) ∧ (y_c − y_p > τ) do
10 |  pick C_i, C_j ∈ P s.t., d_ij = min(Δ);
11 |  C_i ← C_i ∪ C_j; P ← P \ C_j;
12 |  Δ ← Δ \ d_i*; Δ ← Δ \ d_*i;
13 |  y_p ← y_c; y_c ← T_FEAT(P);
14 |  foreach C_j ∈ P : j ≠ i do
15 |  |  Δ ← Δ ∪ d_FEAT(C_i, C_j, α);
16 |  return P
```

Entry Point Selection

HAC's output is a partition of disjoint clusters where every cluster is made of a set of units, with their names, unit-documents, and the related, induced sub-graph of the call graph (Fig.1 box 4).

The key to topoi's discovery are *entry points*. An entry point is a unit that gives access to the implementation of an observable system functionality, such as menu click handlers in GUI, public methods of an API, etc. having some peculiar characteristics leading us to the following assumptions. Unlike ordinary units, entry points originate longer and diverse calling chains. Calling chains ending in an entry point are shorter and fewer than those of ordinary units. For a unit u these assumptions can be expressed as a vector $\mathbf{v}_u = [\deg^{-}(u), \deg^{+}(u), \mathrm{RI}(u), \mathrm{RO}(u), \mathrm{SI}(u), \mathrm{SO}(u)]$ whose 6 elements are:

1. Input Degree ($\deg^{-}$): number of incoming arcs of u;
2. Output degree ($\deg^{+}$): number of outgoing arcs of u;
3. Input Reachability (RI): number of paths ending in u;
4. Output Reachability (RO): number of paths starting from u;
5. Input Path Length (SI): Sum of the lengths of all paths having u as destination;
6. Output Path Length (SO): Sum of the lengths of all paths having u as source;

Hence, every cluster $\mathcal{C} \in \mathcal{P}$ is represented as a matrix $\underset{\forall u \in \mathcal{C}}{\mathrm{U}} = [\mathbf{v}_u]$ and transformed according to PCA. Finally, entry points are ranked respect to their distance with an artificial unit, called query vector ($\mathbf{q}_{\mathcal{C}}$). Following the entry point assumptions mentioned above, within the context of a cluster $\mathcal{C}$, we set the components' values of $\mathbf{q}_{\mathcal{C}}$ following two rules: (1) *Input*-attributes get minima values and (2) *Output*-attributes get maxima values. With these elements the definition of ranking is the following:

Definition 3. (Entry Point Ranking) Let $\mathcal{C}$ be a cluster of n units, the ranking over its units is defined as:

$$\mathcal{K}_{\mathcal{C}} = \{k_1, k_2, \cdots, k_n \mid \mathrm{d}(\mathbf{q}_{\mathcal{C}}, \mathbf{v}_{k_1}) \leq \mathrm{d}(\mathbf{q}_{\mathcal{C}}, \mathbf{v}_{k_2}) \leq \cdots \leq \mathrm{d}(\mathbf{q}_{\mathcal{C}}, \mathbf{v}_{k_n})\}$$

Program Topoi

We can now provide a formal definition of a program topos.

Definition 4. (Program Topos) Let $\Delta_{\mathcal{C}}$ be the set of distances between $\mathcal{C}$'s units and $\mathbf{q}_{\mathcal{C}}$ and, β a known parameter ranging in $(0, 1)$ then the program topos of a cluster $\mathcal{C}$ is the sub-list $\Theta_{\mathcal{C}} \subset \mathcal{K}_{\mathcal{C}}$ such that the distance between any unit of $\Theta_{\mathcal{C}}$ and $\mathbf{q}_{\mathcal{C}}$ is not greater than a threshold value d_β which is the β-order percentile [3].

The units in a program topos $\Theta_{\mathcal{C}}$ are the entry points of $\mathcal{C}$ (Fig.1 box 6).

Experimental Evaluation

Evaluation of the Clustering Step

The main objective of the cutting criterion (Eq.7) is the automatic selection of a partition of clusters to be used in the entry point analysis step. The question we want to address here is:

- **RQ1**: *How effective is the hybrid perspective of* FEAT*, based on* $\mathrm{d}_{\mathrm{FEAT}}$ *and* $\mathrm{T}_{\mathrm{FEAT}}$*, about driving HAC to find optimal partitions in a controlled setting?*

In this experiment we create instances of FEAT models (see Def.1) characterized by the following input variables: number of units, density of the call graph and, a range of values where the length of unit-documents can span. These parameters are used to randomly generate both the call graph and the set of unit-documents. For every instance, the experiment produces all partitions of the set of units, computes modularity and coherence and, find the maximum $\mathrm{T}_{\mathrm{FEAT}}$ value (let us call this part of the experiment *brute force*). Then, we run FEAT and compare $\mathrm{T}_{\mathrm{FEAT}}$ value of the chosen partition to those obtained through brute force.

From combinatorics we know that the number of partitions of a set grows extremely fast. Hence, we let cardinality ($n = |\mathcal{U}|$) of sets of units $\mathcal{U}$ varies in the range $n \in \{10, 11, 12\}$. Correspondingly, the number of partitions are $B_{10} = 115,975$, $B_{11} = 678,570$ and, $B_{12} = 4,213,597$ (B_n is the Bell number of n). Call graph's random creation follows Gilbert's approach (Gilbert 1959) and is ruled by a density value (ρ) corresponding in our context to the probability of having an edge between any pair of vertices. Density's range is $\rho \in \{0.1, 0.2, \ldots, 0.9\}$.

Starting from a fixed alphabet (11 symbols) all words of a given size (4) are generated to form a dictionary of 14,641 words. Unit-documents' lengths span at random having between $[5, 20]$ words. Documents' content is randomly created as well and words' distribution follow Zipf's law (Siddharthan 2002).

[3] The τ-order percentile of a distribution X is the value (x_τ) below which the fraction τ of elements in the ranking fall. Defined as: $P(X \leq x_\tau) = \tau$.

The generation of any instance of the model is replicated 10 times and the results, shown in Tab.1, are averaged out. HAC merges units and clusters, driven by $\mathrm{d}_{\mathrm{FEAT}}$, producing partitions and, selects the partition where the maximum value for $\mathrm{T}_{\mathrm{FEAT}}$ occurs. To compare FEAT's value to brute force approach ones, we provide an indicator called T_{score}. Let us call $\mathcal{T}$ the set of $\mathrm{T}_{\mathrm{FEAT}}$ values of all partitions of $\mathcal{U}$ generated through the brute force approach, then we define the *cutting criterion score* as: $T_{score} = \frac{\mathrm{T}_{\mathrm{FEAT}} - \min(\mathcal{T})}{\max(\mathcal{T}) - \min(\mathcal{T})} \in [0, 1]$ representing how *close* is the partition found from FEAT to the optimal ones.

Every line in Tab.1 contains the average values of 10 tests with the same input values ($n, \rho, \alpha = 0.5$).

n	ρ	T_{score}	n	ρ	T_{score}	n	ρ	T_{score}
10	0.1	0.887	11	0.1	0.880	12	0.1	0.903
	0.2	0.811		0.2	0.893		0.2	0.801
	0.3	0.759		0.3	0.822		0.3	0.848
	0.4	0.740		0.4	0.792		0.4	0.738
	0.5	0.731		0.5	0.750		0.5	0.671
	0.6	0.762		0.6	0.766		0.6	0.795
	0.7	0.780		0.7	0.817		0.7	0.778
	0.8	0.872		0.8	0.849		0.8	0.828
	0.9	0.973		0.9	0.908		0.9	0.977
		0.813			**0.831**			**0.815**

Table 1: Experimental results of the clustering step's evaluation

We observe that when HAC is driven by $\mathrm{d}_{\mathrm{FEAT}}$ and $\mathrm{T}_{\mathrm{FEAT}}$ it shows an interesting performance in finding partitions with both high modularity and coherence. With a very low number of iterations, which in the worst case can be equal to $n-1$, FEAT shows an average $T_{score} > 0.8$ in all combinations that we experimented.

Discovering Topoi through FEAT

This section presents an experiment for the evaluation of FEAT as a technique for recovering the main capabilities of a software system. We selected two open-source softwares (more details about them can be found in Tab.4 lines 2 and 7):

- **Hexdump**. It is an hexadecimal viewer, i.e. an application that displays binary data contained in files as readable sequences of characters.
- **gEdit**. The default text editor of the GNOME desktop environment.

To test the approach we created one oracle for each project by looking at the user's manual and inspecting the source code. gEdit's oracle contains 39 unit names selected as entry points while Hexdump's one contains 12 entry points. The value for β is 0.25 (see Def.4). The research questions we address here are:

- **RQ2**: *What is the impact of α on* FEAT*'s performance?*
- **RQ3**: *How does the selection of textual elements (identifier, comments, etc.) affect* FEAT*'s performance?*
- **RQ4**: *How useful are topoi to identify system's capabilities?*

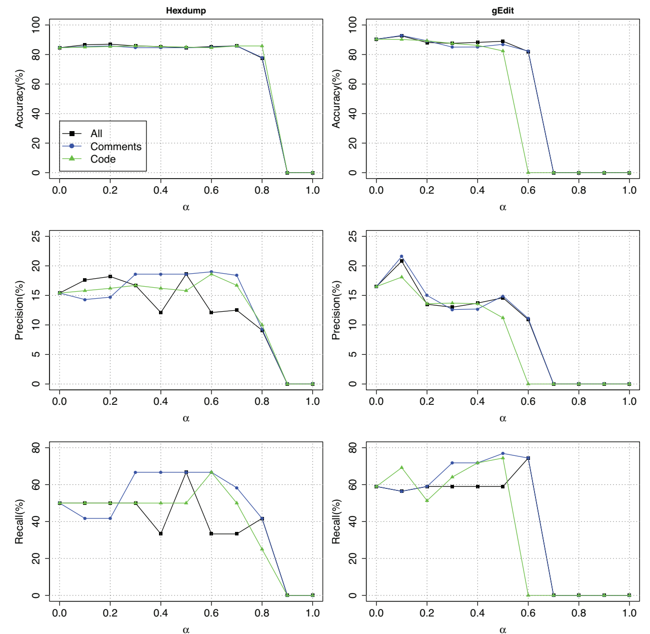

Figure 2: FEAT's performance relative to the analysis of Hexdump and gEdit while α and, extracted textual elements vary

The evaluation proceeds by comparing the entry points automatically extracted from FEAT with those in the oracle and computing *accuracy*, *precision* and, *recall*. The input variables for the experiment are: α and some predefined set of textual elements. The ranges for these variables are:

- $\alpha \in [0, 1]$ with increments of 0.1
- Textual sets of elements: *Code*, *Comments* and, *All*. Code includes: identifiers and literals. Comments: just comments are extracted. All: union of code and comments.

This experiment generated 60 tests whose results are summarized in Fig.2. The first observation we can make is that the cutting criterion drastically affects FEAT's behaviour. As soon as α goes above a given value ($\alpha \geq 0.8$ for Hexdump and $\alpha \geq 0.6$ for gEdit), the quality of classifications drastically drops. The reason for this is that the clustering process, which for $\alpha \geq 0.6$ is mainly driven by semantic elements, is stopped too early creating clusters which are too small and, from the structural standpoint, too fragmented. When clusters are internally too sparse the entry point ranking cannot be executed. Instead, balanced values of $\alpha \approx 0.5$ show the best performance for accuracy ($\geq 80\%$) and recall ($\approx 70\%$).

Let us summarize the results in the light of the research questions. (**RQ2**) A balanced combination of $\alpha \approx 0.5$ leads to higher recall values for every set of extracted textual elements. Accuracy and precision do not seem to be much affected by α. (**RQ3**) More textual elements, as in the *All* combination, does not imply better performance. The blue line in Fig.2 is related to *Comments* which obtains higher values in almost all cases. The last research question (**RQ4**) deals

Unit Name	Index
`_gedit_cmd_file_open`	**chooser**, **cmd**, [...], **dialog**, **document**, **file**, **folder**, [...], **open**
`findit_cb`	[...], **findit**, **found**, [...], **hit**, [...], **mask**, **pattern**, **search**

Table 2: Part of two program topoi obtained from the analysis of gEdit (first line) and Hexdump

	Unit Name	$d(\mathbf{q}_C, \mathbf{v}_i)$
1	`_gedit_cmd_search_find_`{**`prev,next`**}	0.139
2	`_gedit_cmd_edit_`{**`delete,copy, select_all,cut,redo,undo`**}	0.158
3	`findit_cb`	0.302
	`find_next_cb`	0.311

Table 3: Extract of the rankings of two program topoi with distances between entry points and query vector

	Project	LOC	#Unit	#File	Dict.	ρ
1	Linux FS EXT2	8,445	180	14	748	0.0201
2	Hexadec. Viewer	12,053	254	13	764	0.0091
3	GNU bc Calculator 1.06	12,851	215	20	723	0.0204
4	Intel Ethernet Drivers and Util.	30,499	581	16	1,479	0.0062
5	Ultradefrag v7.0	34,637	1,112	74	1,874	0.0054
6	Zint Barcode Generator v2.3.0	38,095	345	43	1,275	0.0134
7	GNU Editor v3.2	42,718	1,370	59	1,048	0.0021
8	bash v1.0	70,955	1,477	128	2,216	0.0027
9	Linux IPv4	84,606	2,216	127	3,211	0.0011
10	x3270 Terminal Emulator v3.5	91,449	1,881	136	3,008	0.0025

Table 4: Open Source software projects used during the scalability evaluation

with the usability of FEAT in a real setting. Tab.2 shows two entry points of two program topoi: the first from gEdit and the second from Hexdump. In the first line we have the *open file* capability of the text editor while in the second one the *find a pattern* capability of Hexdump. Some of the words contained in the indices provide useful clues about the context and purpose of the capability accessed through the entry point to which they belong.

Another interesting point to highlight is shown in Tab.3 where we have three groups of units taken from two topoi of the two projects. Every group lists units at the same (or quite similar) distance to the query vector. The interesting thing is that the units in every group belong to the same functional area and their relatedness can be revealed by the reciprocal proximity in the PCA space. In the first group we have *find* related functionality (the suffixes to complete unit names are between curly braces), in the second one some *clipboard management* units and, in the last one again *find* related functionality. We believe that these information can be profitably used for the discovery of system's capabilities.

Scalability Evaluation of FEAT

The adoption of a new methodology is determined also by its scalability, then in this section we address the following research question:

- **RQ5** *What is the correlation between* FEAT*'s running time and memory usage w.r.t. software projects characteristics such as: number of units, size of the dictionary, LOC and, call graph density?*

FEAT has been implemented on top of a software testing platform based on OSGi (Open Services Gateway initiative) and BPMN (Business Process Modeling Notation). The hardware used for running the experiments is based on an Intel dual core i7-4510U CPU with 8GB RAM.

The last experiment, whose results are reported in Fig.3, involves the projects listed in Tab.4. On the left hand side we have graphs reporting the running time ($RT\,[\mathrm{s}]$) of the experiment while α varies in $\{0, 0.5, 1\}$. For every value of α we have two curves: the total time employed by FEAT (black) and the clustering time (blue). The difference between black and blue curves is equal to the time needed by the preprocessing step. In all cases the time of the entry point selection step can be neglected. The right hand side of Fig.3 contains the graphs about memory usage. In this case we plot only one graph per characteristic because no meaningful differences have been observed among the various values of α. Let us focus on the running time graphs. The analysis of running time leads to two main observations. First, performance is negatively affected by values of $\alpha > 0$ which means that the computation of the textual distance (Eq.2) and the coherence criterion (Eq.6) are demanding. Indeed the graph KWords/Time shows a steep increase around $\approx 2\,\mathrm{KWords}$. Furthermore, looking at the KUnit/Running Time graph for $2\,\mathrm{KUnit}$ we have $RT_{\alpha=0} \approx 1400\,\mathrm{s}$ and $RT_{\alpha=0.5} \approx 10\,000\,\mathrm{s}$. Second, the clustering step, whose complexity is $\Theta(n^2 \log(n))$, is the most costly one when $\alpha > 0$ while for $\alpha = 0$ running time is dominated by the preprocessing step. Density shows a negative correlation w.r.t. both time and memory. Regarding the memory usage no clear patterns emerged, nevertheless we observe a fast growth in relationship with $KLOC \geq 70$. In conclusion, FEAT shows acceptable performance with projects counting up to $1.3\,\mathrm{KUnit}$ and dictionaries with size up to $\approx 2\,\mathrm{KWords}$ producing results in about $20\,\mathrm{min}$, for greater values of these characteristics the full hybrid approach becomes rapidly too costly.

As further work, we believe that improving the efficiency of distances' computation over the semantic part of the source code, would ease the adoption of the tool and maximize its impact.

Conclusion

FEAT is a novel approach for program understanding representing a software system under both structural and semantic perspectives. Program topoi are concrete and useful summaries of systems' capabilities providing a valuable help in several phases of the software life-cycle. FEAT is a feasible, unsupervised approach for automatically discovering program topoi directly from source code.

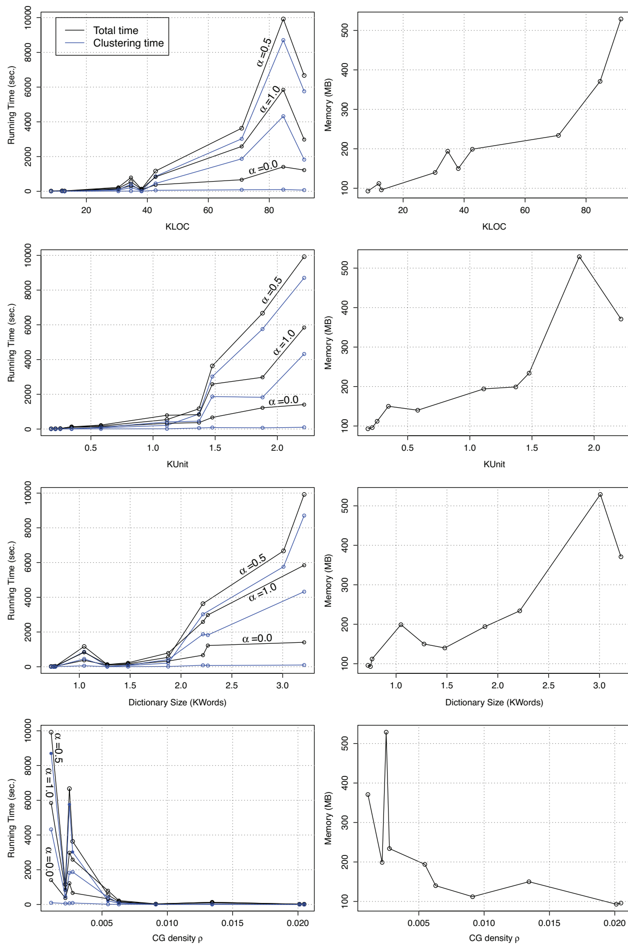

Figure 3: Correlation between running time and memory usage w.r.t. LOC, number of units, size of the dictionary and, density of CG

Our plans for the deployment of FEAT include *Software Heritage*[4](SH). SH is a public, large archive counting almost 65M open-source software projects, we plan to integrate FEAT with SH's platform in order to provide a more abstract approach for searching software artefacts, based on program topoi, allowing the usage of natural text in queries.

References

Aaron, C.; Newman, M.; and Moore, C. 2004. Finding community structure in very large networks. *Physical Reviews E.* 70.

Abebe, S. L., and Tonella, P. 2015. Extraction of domain concepts from the source code. *Science of Computer Programming* 98:680–706.

Andritsos, P., and Tzerpos, V. 2005. Information-theoretic soft. clustering. *IEEE Trans. Software Eng.* 31(2):150–165.

Chen, K., and Rajlich, V. 2000. Case study of feature location using dependence graph. In *Program Comprehension, 2000. Proceedings. 8th International Workshop*, 241–247.

Donetti, L., and Muñoz, M. A. 2004. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*.

Dumitru, H.; Gibiec, M.; Hariri, N.; Cleland-Huang, J.; Mobasher, B.; Castro-Herrera, C.; and Mirakhorli, M. On-demand feature recommendations derived from mining public product descriptions. In *ICSE'11*, 181–190.

Foltz, P. W.; Kintsch, W.; and Landauer, T. K. 1998. The Measurement of Textual Coherence with Latent Semantic Analysis. *Discourse Processes* 25(2-3):285–307.

Gilbert, E. N. 1959. Random graphs. *Ann. Math. Statist.* 30(4):1141–1144.

Kaufman, L., and Rousseeuw, P. J. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley.

Kuhn, A.; Ducasse, S.; and Gîrba, T. 2005. Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering,2005*, 133–142.

Kuhn, A.; Ducasse, S.; and Gîrba, T. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49(3):230–243.

Linstead, E.; Rigor, P.; Bajracharya, S.; Lopes, C.; and Baldi, P. 2007. Mining concepts from code with probabilistic topic models. In *Proc. of Automated Software Eng.*, 461.

Manning, C. D.; Raghavan, P.; and Schütze, H. 2008. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press.

Marcus, A., and Haiduc, S. 2013. *Text Retrieval Approaches for Concept Location in Source Code*. Berlin, Heidelberg: Springer Berlin Heidelberg. 126–158.

McBurney, P. W.; Liu, C.; and McMillan, C. 2016. Automated feature discovery via sentence selection and source code summarization. *Journal of Software: Evolution and Process* 28(2):120–145.

McMillan, C.; Hariri, N.; Poshyvanyk, D.; and Cleland-Huang, J. 2012. Recommending Source Code for Use in Rapid Software Prototypes. In *Proc. of Int. Conf. in Soft. Engineering (ICSE'12)*, 848–858.

Mitchell, B. S. 2003. A heuristic approach to solving the software clustering problem. In *19th International Conference on Software Maintenance,2003*, 285–288.

Rubin, J., and Chechik, M. 2013. A survey of feature location techniques. *Domain Engineering* 29–58.

Shtern, M., and Tzerpos, V. 2012. Clustering Methodologies for Software Engineering. *Advances in Software Engineering* 2012:1–18.

Siddharthan, A. 2002. Christopher d. manning and hinrich schutze. *Foundations of Statistical Natural Language Processing*. MIT press, 2000. ISBN 0-262-13360-1, 620 pp. $64.95/£44.95 (cloth). *Natural Language Engineering* 8(1):91–92.

Tzerpos, V., and Holt, R. C. 2000. ACDC: an algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering, 2000*, 258–267.

Wiggerts, T. A. 1997. Using clustering algorithms in legacy systems remodularization. In *4th Working Conference on Reverse Engineering, 1997*, 33–43.

Xu, X.; Lung, C.; Zaman, M.; and Srinivasan, A. 2004. Program restructuring through clustering techniques. In *4th IEEE International Workshop on Source Code Analysis and Manipulation, 2004*, 75–84.

[4]www.softwareheritage.org