The Journal of Supercomputing (2020) 76:87–121
https://doi.org/10.1007/s11227-019-02995-3

Improving the modularization quality of heterogeneous multi-programming software systems by unifying structural and semantic concepts

Masoud Kargar[1] · **Ayaz Isazadeh**[2] · **Habib Izadkhah**[2]

Published online: 23 September 2019

Abstract

Program comprehension plays a significant role in the maintenance of software systems. There has recently been a significant increase in written large-scale applications with a collaboration of several programming languages. Due to the indirect collaboration between different components of a multilingual program, it is difficult to understand such program. Modularization is employed for extracting subsystems to help software system comprehension. The first step in the software modularization process is to extract a dependency graph from the source code. Taking into account all the programming languages used to implement a program, in the literature, there is no method to construct an integrated dependency graph from the source code aiming to support modularization of multilingual programs. To modularize such programs, we, first, create three dependency graphs named Call Dependency Graph (CDG), Semantic Dependency Graph (SDG) and Nominal similarity Dependency Graph (NDG) from the source code. The CDG, as a structural graph, is constructed for homogeneous programming languages and both SDG and NDG are built without taking into account the syntax of the programming languages used in the source code. Then, a genetic algorithm is presented to modularize multilingual programs from the constructed dependency graphs. The experimental results on Mozilla Firefox demonstrate that improvements in the simultaneous use of the SDG and NDG, and structural-based graph are 89%, 85%, 86%, and 59%, respectively, in terms of Precision, Recall, FM, and MoJoFM. The source codes and dataset related to this paper can be accessed at https://github.com/Masoud-Kargar-QIAU.

Keywords Software comprehension · Software evolution · Software architecture · Modularization · Clustering · Multi-programming language · Multilingual programs · Genetic algorithm

✉ Habib Izadkhah
habib_eizadkhah@yahoo.com; izadkhah@tabrizu.ac.ir

Extended author information available on the last page of the article

1 Introduction

Understanding the structure of large and complicated software systems is difficult and sometimes impossible. On the other hand, the software structure deviates from the original architecture in the process of maintaining and evolving an application. Therefore, after a while, an application cannot be comprehended, evolved, and maintained. Modularization is employed to partition a software system into several meaningful and understandable subsystems (modules) to facilitate the comprehension and development process [1].

The first step in the modularization process of a software system is to create a dependency graph named Artifact Dependency Graph (ADG) from the source code. This graph represents the artifacts and relationships between artifacts in a software system. In a program, files, functions, classes, methods, etc. can be considered as an artifact. We formally define this graph first. Suppose an artifact dependency graph is labeled as $ADG = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of n artifacts and $E \subseteq V \times V \times R^+ = \{(v_i, v_j, w(v_i, v_j)\}$ is the set of relationships between artifacts so that $w(v_i, v_j)$ indicates the weight of the edge between artifacts v_i and v_j. This weight can be computed from the number of calls, semantic similarity, or nominal similarity between two artifacts. The modularization process leads to partition all artifacts into k non-overlapping modules $\{m_1, m_2, \ldots, m_k\}$, where $m_1 \cup m_2 \cup \ldots \cup m_k = V$ and $m_i \cap m_j = \phi, i, j = 1, 2, \ldots, k$ and $i \neq j$.

The process of modularization of a software system includes three main steps (Fig. 1). At the first step, the source code is analyzed, then an analysis tool is employed to create the artifact dependency graph indicating the relationships among artifacts in the source code. This graph is regarded as an input at the second step in which modularization techniques are used for the partition of software systems. At the third step, the resulting modules are labeled and displayed [1].

The ADG is classified into two dependency graphs: a structural dependency graph and a non-structural dependency graph, taking into account the type of relationships between artifacts. In a structural dependency graph, the relationships between artifacts can be of different kinds including invocation (calling dependency), inheritance, link to the header; while in a non-structural dependency graph the relationships between artifacts are the similarity between identifier names and the similarity between comments. The Call Dependency Graph (CDG), as a structural dependency graph, is a typical representation of artifacts and relationships between them in a

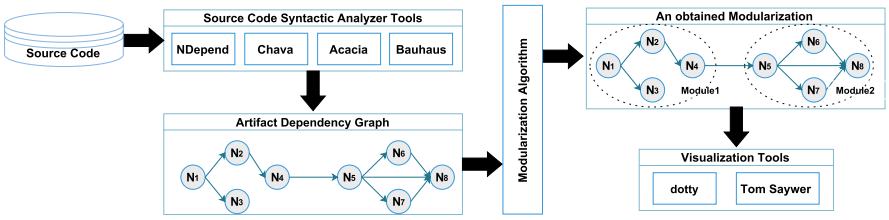


Fig. 1 A general schema of the modularization process

software system [2–4]. In this graph, the nodes and edges represent artifacts and invocation between artifacts, respectively. This graph depends on the structure and syntax of the programming language used. Semantic-based software modularization methods use the non-structural dependency graph for modularization.

Selecting an artifact depends on the size of a software system and the type of programming language used. Usually, a function is regarded as an artifact in the procedural programming languages; however, a class is selected as an artifact in the object-oriented programming languages. The relationships between artifacts are determined by the type of selected artifact. The performance of a modularization algorithm depends on the accuracy and integrity of the created ADG. For this purpose, the ADG should be able to cover the entire source code. The dependency graph extraction tools are available for a small number of programming languages, so that existing tools for extracting dependency graph depend on the syntax of programming languages.

Due to the use of the best library in a project independent of the programming language, taking advantage of reusable components (which may have originated in different source languages), efficiency (each programming language is more suitable for a specific task), multi-platform and multi-developers, current, most large projects are built with multiple programming languages [5, 6]. For example, Fig. 2 shows a part of the content folder of Mozilla Firefox, which a file written in C++ language calls a JavaScript file. In fact, the two files written in two different programming languages depend on each other. Selecting the right programming languages in the large projects reduce the run time and maintenance costs, minimize the time programmers spend programming, as well as increase the ability to adapt to various platforms. For example, the number of languages used to write the files in the Extensions folder of Mozilla Firefox released in July, 2017 are 102 files in C++, 9 files in C, 77 header files in C/C++, 26 files in Java, 33 files in JavaScript, 1 file in Lisp, 1 file in Perl, 2 files in Objective C++, etc.

Evaluation and analysis by the authors on twenty open-source applications (Fig. 3), in July 2019, from different areas such as industrial, web applications, content management, operating system, mobile operating system, cloud computing, etc.

```
nsJSContext::CallEventHandler(nsISupports* aTarget, void *aScope, void
*aHandler, nsIArray *aargv, nsIVariant **arv)
{
  NS_ENSURE_TRUE(mIsInitialized, NS_ERROR_NOT_INITIALIZED);

  if (!mScriptsEnabled) {
    return NS_OK;
  }

  JSObject* target = nsnull;
  nsresult rv = JSObjectFromInterface(aTarget, aScope, &target);
  NS_ENSURE_SUCCESS(rv, rv);

  js::AutoObjectRooter targetVal(mContext, target);
  jsval rval = JSVAL_VOID;
```

Fig. 2 Collaborating two programming languages C++ and Java Script

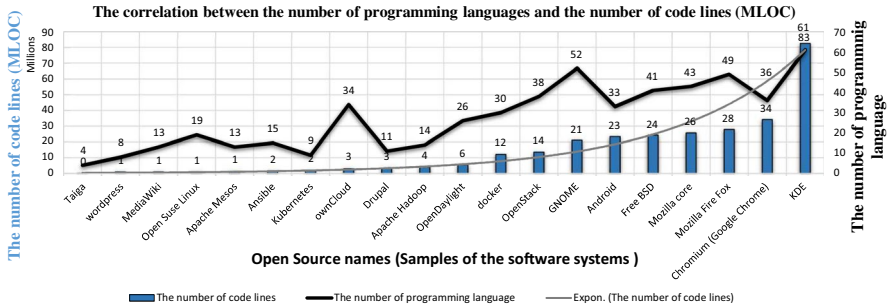


Fig. 3 Increasing the number of used programming language by growing the size of the source code (July, 2019; http://www.openhub.net)

confirms that, usually, the number of programming languages used to write an application increase with growing source code size. According to the investigations of the TIOBE committee into the use of different programming languages in software systems, the first twenty widely used programming languages cover nearly 67% of the source codes. Moreover, the first fifty widely used programming languages cover 74% of the source codes. Our evaluation on applications shown in Fig. 3 confirms that the main programming language covers less than 43% of the source code on average in multilingual software systems (indicated in Fig. 4); also, the comments contribute to nearly 23% of the entire source code.

Understanding the entire of multilingual source code is a complex process, and therefore, automatic modularization will be of great importance. Because the homogeneous languages can call each other directly, existing toolsets can analyze the code of a program written with a single programming language or homogeneous languages (e.g., C, C++, and Java) and extract the ADG from the source code. Therefore, the use of existing techniques and tools enables a software engineer to understand a program written with homogeneous languages of the multilingual system. But, because of indirect cooperation and the absence of a tool for detecting indirect dependencies, understanding a program written with heterogeneous languages for maintenance operation is not possible using existing modularization methods.

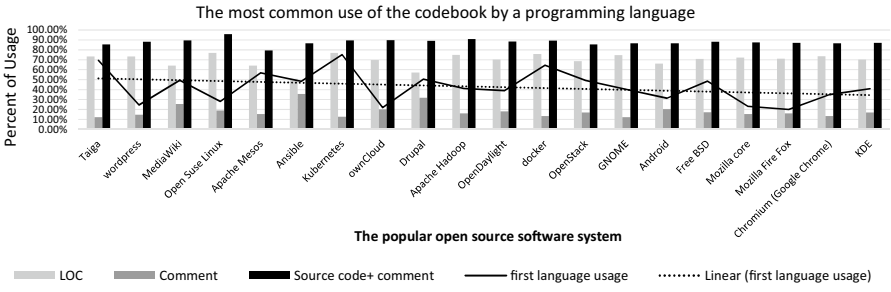


Fig. 4 The usage percentage of source code in software systems (July, 2019; http://www.openhub.net)

The ADG extraction tools exist for a small number of programming languages, hence, in a heterogeneous multilingual software, portion of the source code is ignored by them. This neglected portion can affect the modularization quality of a software system. To overcome these problems, this paper presents an approach to semantic analysis regardless of the programming language syntax. For this purpose, a semantic dependency graph and a nominal similarity dependency graph are presented to extract the non-structural features hidden in the source code languages regardless of the programming languages used. The semantic dependency graph is related to all semantics contained in the vocabulary of source code including statements, libraries, utilities, comments, identifier names (e.g., functions, classes, methods), etc. The nominal similarity graph is only related to the file names, not other artifacts (i.e., functions, classes, methods names). It's worth noting that filenames are not related to the syntax of programming languages. Given the fact that software system modularization is an NP-hard problem, the genetic algorithm is used for modularization here. Then the combination of semantic and nominal similarity graphs with the CDG is described, and an objective function, corresponding to the new graphs, is presented. The resulting combination graph includes structural and non-structural information of the entire source code; therefore, a modularization algorithm can benefit from the integrity and diversity of syntactic and semantic information in the entire source code. This combination graph can also be used as the input by other modularization graphs. The results on Mozilla Firefox (a large-scale and open-source application) indicate that the modularization quality improved significantly by using the new combination graph compared to CDG.

The contributions of this paper are summarized as follows:

- With creating three different dependency graphs, named call dependency graph, semantic dependency graph, and nominal similarity dependency graph, from the source code and combining them, the proposed method makes the modularization possible for programs written with heterogenous multiple programming languages. These dependency graphs cover most parts of a program.
- The proposed method is the first method that uses calling dependency, as a structural feature, semantic information in the entire source code extracted from all vocabulary of source code, and also similarity between artifact names, for modularization.

This paper is organized into six sections. Section 2 addresses different tools used to extract the ADG in programming languages. This section also deals with modularization algorithms that use the CDG and the semantic graph. Section 3 describes the proposed method including, how to create the call dependency graph, semantic dependency graph, and the nominal similarity graph and also the new quality function. Section 4 gives the experimental setup. Section 5 compares and analyzes the empirical results of different graphs on the modularization algorithm by using various evaluation criteria. Section 6 addresses the threats to validity. Section 7 presents the conclusion, future works, and the new research areas.

2 Related work

This section presents related work on software modularization techniques presented in the literature. It reports contributions on the following topics:

- Dependency graph extraction tools,
- Modularization techniques based on structural dependency information extracted from the source code,
- Semantic-based modularization techniques,
- Techniques that use a combination of syntactic and semantic features.

The dependency graph extraction tools are available for some popular programming languages, and these tools can only extract the structural dependency from source code. Table 1 shows different tools for extracting structural dependency graphs from the source code. According to this table, there are many programming languages with no ADG extraction tools. Therefore, it is not possible to extract an accurate and integrated ADG from the entire source codes of software systems implemented with several programming languages.

Most existing state-of-the-art modularization algorithms use structural concepts for modularization such as Weighted combined algorithm [30], Bunch [3, 31, 32], DAGC [33], NAHC [32], HC+Genetic [34], ECA [2], MCA [2], LA [35], HSBRA [36], E-CDGM [23], EDA [37], HC-SMC [38], GA-SMCP [38], MAEA-SMCP [38], SLTFIDFW [39], PSO-based algorithm [40], Multi Agent [41], and Neighborhood tree [42]. The main drawback of these algorithms is that they depend on programming language syntax. They do not cover the entire source code.

Table 1 Existing dependency graph extraction tools for different programming languages

Programming language	Tools
C	Acacia [7], CC_Rider [8], CIA [9], Bauhaus ccdiml [10, 11], Rigi [12], SNIFF+ [13], SHriMP [14], CSV [15], Solidsx [16], Columbus [17], Imagix_4D [18], ARCH [19] and Analizo [20]
C++	Acacia [7], Reprise [21], CC_Rider [8], Bauhaus ccdiml [10, 11], Rigi [12], SNIFF+ [13], SHriMP [14], CSV [15], Solidsx [16], Cloumbs [17], Imagix_4D [18], Reveal [15, 22], Analizo [20], NDepend [23] and Understand [23]
Java	Chava [13], AnyJ [8], JDepend [24], Bauhaus ccdiml [10, 11], SHriMP [14], Code Crawler [25], Solidsx [26], Imagix_4D [18], Analizo [20], NDepend [23] and Understand [23]
COBOL	COBOLSRE [27, 28] and Rigi [12]
Ada	Bauhaus ccdiml [10, 11]
.Net C#	Solidsx [16], DPMId [23], NDepend [23] and Understand [23]
Visual Basic	Solidsx [16]
Prolog	Vmax [29]

Bunch [3, 31, 32], as a popular modularization algorithm, uses genetic algorithm and hill climbing to extract software architecture from the source code. The input of the Bunch is call dependency graph, which is constructed from source code. The output of the Bung algorithm is a modularization with minimum coupling and maximum cohesion.

In [2], two multi-objective algorithms named MCA and ECA were proposed for modularization. The objectives used in MCA are "maximizing the sum of intra-edges of all modules", "minimizing the sum of inter-edges of all modules", "maximizing the number of modules", "maximizing TurboMQ", "minimizing the number of isolated modules"; and the objectives used in ECA are the same as MCA with the difference that instead of the last one, the difference between the maximum and minimum number of artifacts in a module be minimum, has been used.

Huang and Liu [38] stated that BasicMQ proposed in [31] does not consider global modules and edge directions between two modules. Therefore, they proposed an objective function, named MS, to overcome the limitations of BasicMQ. To validate the performance of MS, they developed three algorithms named hill-climbing algorithm (HC-SMCP), genetic algorithm (GA-SMCP), and multiagent evolutionary algorithm (MAEA-SMCP). They only compared the performance of their proposed quality function with BasicMQ.

In [40], first, four objectives "intracluster dependency", "intercluster dependency", "number of clusters", and "number of module per cluster" are defined, and then particle swarm optimization algorithm is applied to modularize a program taking into account the presented objectives.

In [36], a harmony meta-heuristic-based algorithm was proposed for modularizing software systems which are written in object-oriented programming language. The objectives used in this algorithm are cohesion, coupling, package count index, and package size index.

In semantic-based methods, the selection of words is the main reason for a difference in the proposed algorithms and its constraints. In [43], the repetitive words of the source code are counted and regarded as the criterion for Latent Semantic Analysis (LSA). In [44], specific sections of the source code such as variables, names of functions, comments, and names of classes are selected for semantic analysis. In [45, 46], the keywords of the programming language are selected as a term in the semantic analysis. In [47], the specialized words of the programming language are chosen as a term. It can be concluded that semantic algorithms depend directly and indirectly on the syntax of the programming language.

The algorithm proposed in [48] uses the syntactic and semantic methods simultaneously. It uses data flow and control for the structural analysis. The algorithm uses conceptual analysis with few artifacts, so that it is not proper for large applications. In [49], information flow-based coupling and conceptual coupling between classes were used only in object-oriented languages for the structural and conceptual analyses, respectively. In [50], the dependence of classes on different types of packets was used in the structural analysis. Moreover, the sharing rates of variables and methods were employed in the conceptual analysis. Investigating these methods and their performances confirmed the dependency on the syntax of programming languages.

In [46], a hierarchical average-linkage-based algorithm was proposed so that it uses semantic information contained in source code for modularization. This algorithm for information retrieval uses identifier names (e.g., class names) and comments. All the applications used for the experiments are written in a single programming language.

In [51], a local search-based algorithm was proposed to utilize the structural features (such as CDG and inheritance dependency) and semantic contained in the comments and identifier names for modularization. In [39], a genetic-based algorithm was proposed for modularizing programs which are written in object-oriented programming languages. The algorithm considers several coupling schemes such as structural and lexical dependencies. Like other algorithms, these two algorithms cannot be applied to software systems written with several programming languages.

In [4], an entropy-based measure was presented to compute the similarity between artifacts; then, a greedy algorithm was proposed for modularization considering structural and non-structural features.

To conclude, the major limitations of the existing methods with the concern for multilingual software systems modularization are summarized below:

- Most modularization algorithms utilize structural features of the source code, e.g., calling dependency, inheritance dependency, etc., to modularize a program. There exist two problems in this regard: (1) due to the syntactically diverse of the programming languages, extracting an integrated ADG for a multilingual program is not possible, and the use of existing methods leads to separate and island graphs that are not practically functional; and also (2) for some programming languages there is no tool to create an ADG and for some programming languages, there are such tools, hence, the constructed ADG will not be able to cover the whole program.
- Some methods utilize semantic features, e.g., the semantic contained in comments, identifier names or keyword, for modularization. There exist two problems in this regard: (1) because identifier names and keywords are dependent on the syntax of the programming language used, they can not be used; and (2) comments are independent of the syntax of programming languages, but they form only a small fraction of the entire source code. However, the semantic-based methods can be used to modularize multilingual programs. But the modularization quality achieved by these methods will be low.

3 The proposed approach

In this section, we present a method to modularize multilingual programs. The preliminary activity in the software modularization process is to extract an intermediate model, called artifact dependency graph, from the source code. Taking into account all the programming languages used to write a program, there is no method to construct an integrated dependency graph from the source code aiming to support modularization of multilingual programs. To modularize such programs, we in this section, first, create three dependency graphs named Call Dependency Graph

(CDG), Semantic Dependency Graph (SDG) and Nominal similarity Dependency Graph (NDG) from the source code. The CDG, as a structural graph, is constructed for homogeneous programming languages and both SDG and NDG are constructed without taking into account the syntax of the programming languages used in the source code. In all these dependency graphs, the nodes represent artifacts and edges in CDG, SDG, and NDG, represent, respectively, calling dependency among artifacts, semantic contained among vocabulary of two artifacts, and similarity among names of two artifacts. Then, a genetic algorithm with an updated objective function is presented for modularization from the constructed dependency graphs. The proposed approach consists of three steps:

- **Step 1** extracting the CDGs depending on the programming languages used;
- **Step 2** extracting the semantic dependency graph and the nominal similarity dependency graph, aiming to cover the entire source code;
- **Step 3** modularizing the source code using the genetic algorithm considering semantic, nominal similarity, and CDG.

Figure 5 shows these steps. The first step depends on the syntax of the programming language. Depending on each programming language used in the source code, the corresponding tool is used to create the CDG, which is regarded as the input of the modularization algorithm in the third step. The second step does not depend on the programming language and covers the entire source code. The outputs of this step are the semantic graph and the nominal similarity graph.

The steps in creating the CDG, semantic graph, the nominal similarity graph, and the genetic modularization algorithm are described below.

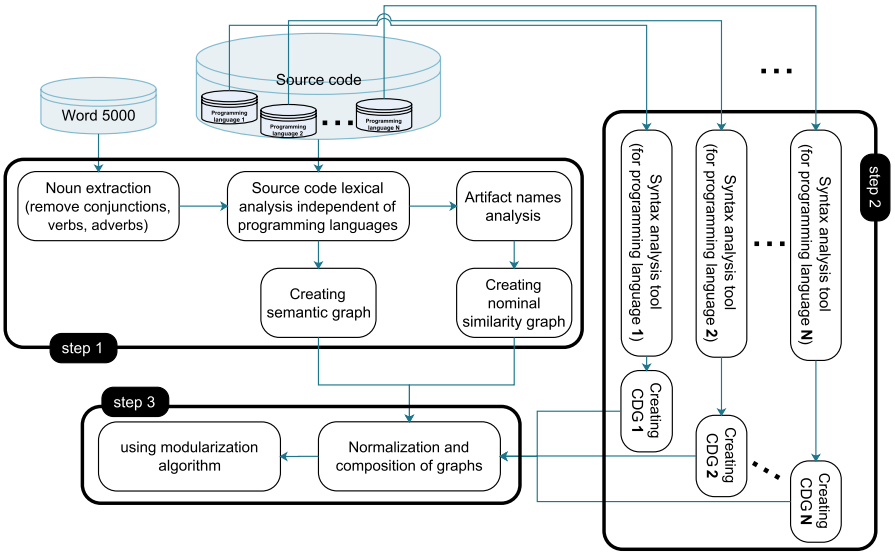


Fig. 5 The general schema of the proposed approach

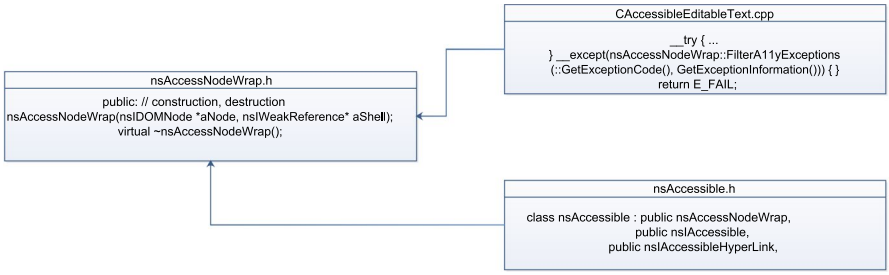


Fig. 6 A sample code from Accessible folder of Mozilla Firefox

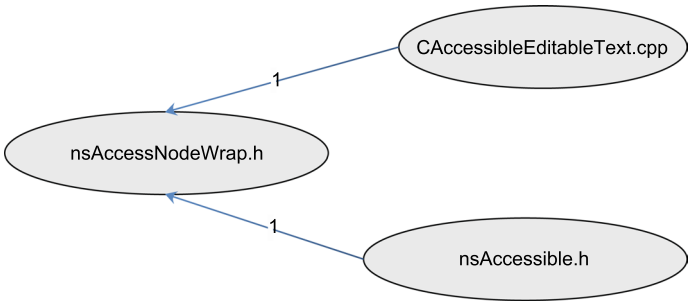


Fig. 7 Call dependency graph constructed for code of Fig. 6

3.1 Call dependency graph

The call dependency graph (CDG) is used to illustrate the structural dependence between the software system artifacts. This graph is directional and indicates the calling dependency, e.g., function call, between artifacts. The vertices of this directed graph are the software system artifacts and its edges are calls between artifacts. When artifact i calls a function of artifact j, then an edge between the two vertices is defined. If the graph is defined weighted; in that case, the weight of the edges will point to the number of calls between artifacts. The call dependency graph is defined as follows:

Suppose $V = \{\text{Software system artifacts such as} A_1, \ldots, A_n\}$

$G = \{V, E\}$ s.t. $E \subseteq V \times V \times R^+$

$if < i, j > \in E. \therefore \exists f, f \in A_j \vdash \exists \text{function call}(A_i, f) \triangleq$ f call by A_i

For example, Fig. 6 shows three artifacts form Accessible folder of Mozilla Firefox 3.7, and CDG constructed for them is shown in Fig. 7. The numbers on the edges in Fig. 7 show the number of calls between artifacts.

3.2 Semantic dependency graph

In the literature, there are two semantic-based methods to compute the similarity between two concepts, including semantic similarity and semantic relatedness.

"Semantic similarity has been identified as a particular subset of this general notion of semantic relatedness. Semantic similarity relies on the general cognitive ability to detect similar patterns in stimuli, which attracts considerable attention in cognitive science" [52]. In this sense, "car" and "bus" are similar (they are both means of transport), while are also related but not similar to "road" and "driving". Semantic relatedness between documents can be computed using statistical methods such as a vector space model to correlate words and textual contexts from a suitable text corpus [53].

In this sub-section, a dependency graph, called semantic dependency graph, using semantic relatedness similarity is created according to the contents of source code without considering any assumptions on their syntax. To achieve this aim, we use probabilistic latent semantic analysis (PLSA) technique as a semantic relatedness measure. In this dependency graph, the number of vertices is equal to the number of artifacts. Each element of it indicates the semantic relatedness and content similarities between two artifacts. We first formally define this graph. Suppose a semantic graph is labeled as $SG = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of n artifacts and $E \subseteq V \times V \times R^+ = \{(v_i, v_j, s(v_i, v_j)\}$ is the set of links between artifacts so that $s(v_i, v_j)$ indicates the similarity value between v_i and v_j calculated by $cos(\vec{v_i}, \vec{v_j})$. The vector $\vec{v_i}$ is defined as follows:

$\vec{v_i} = \{N_i(f_1), N_i(f_2), \ldots, N_i(f_n), \quad \forall f_i \in$ word5000 dataset s.t. $N_i(f_k)$ is the number of f_k in $v_i\}$

The following steps are taken to create this graph:

- **Step 1** Selecting the nouns from Word5000
  The data of COCA (the Corpus of Contemporary American English) were checked from 1990 to 2015 that 520 million words in different contexts were repeated frequently. Then words were sorted descend for variant datasets and denominated Word5000, Word60000, and Word100000. The words in datasets were major genres (popular, fiction, spoken, magazines, newspapers, and academic) and more than 40 subgenres and then amount of occurrence counted in per genre and subgenre. At http://www.wordfrequency.info has word5000 as ranked according to the frequency so the highest one is 22,038,615 in this dataset and the lowest one is 4875. When the number of words in the dataset exceeds from 10,000 words, the frequency of word declines to 300 or fewer, accordingly, with increasing the words in the datasets, accuracy increase too. Expect nouns, all part of datasets (verbs, conjunctions, adverbs, etc.) were disregarded because of increasing the calculation speed and preventing the side effect. Note that Word5000 has 2543 nouns.
- **Step 2** Lexical analysis and conversion of all the letters into small case letters
  The contents and components of all artifacts are detected according to separators such as distance, mathematic symbols, parentheses, and the end of lines. Then all the letters of components are converted into small case letters. After that, an array of components is created for each artifact.
- **Step 3** Creating an Artifact×Noun matrix named M
  This matrix indicates the importance of each noun in an artifact (here file). Let N denote the number of artifacts, M is an N × 2543 matrix in which each

row belongs to an artifact, and each column belongs to a noun selected from Word5000. Each element like M_{ij} shows the number of noun j in artifact i. Each element of this matrix is replaced by the value calculated using Eq. 1, which $freq_{ij}$ is the frequency of noun j in artifact i.

$$M_{ij} = \frac{\log(freq_{i,j} + 1)}{-\sum(p(noun_j|artifact_i) \times \log(p(noun_j|artifact_i)))} \tag{1}$$

where

$$p(noun_j|artifact_i) = \frac{freq_{i,j}}{\sum_{a=1}^{2543} freq_{i,a}}$$

Equation 1 is intended to reflect how important a noun is to an artifact.

- **Step 4** Dividing M into three matrices by using the PLSA technique

M is a matrix indicating the probabilities of artifacts with features in a software system. In this matrix, there are some latent concepts which cannot be detected easily. These concepts are based on the relationships between artifacts and features in *M*. PLSA is a technique which detects and ranks these features based on probability value. PLSA can be computed in two different ways, including Latent Variable Model and Matrix Factorization. In this paper, we use non-negative matrix factorization as a matrix factorization method. Let m, n, and r denote the number of artifacts, the number of features, and the number of concepts of artifacts, respectively. This technique divides the artifact/feature matrix into *L*, *U*, and R^T matrices as Eq. 2.

$$M_{m\times n} = L_{m\times r} U_{r\times r} R^T_{r\times n} \tag{2}$$

In this step, the nouns are selected from Word5000 as features. The importance of these concepts is shown by eigenvalues in a diagonal matrix like *U*. They are put on the main diagonal in descending order. Some large indices can be overlooked. The matrix *L* indicates the concepts and importance of an artifact in a software system. The matrix *L* contains the artifact probabilities *p*(artifact|concept). *U* is a diagonal square matrix which shows r concepts in the order of importance on the main diagonal and shows prior probabilities of the concepts *p*(concept). Each eigenvalue is attributed to a concept in correspondence to *L* and *R* in a way that the first column of *L* indicates the vector corresponding to the first concept for all artifacts. The values of this vector indicate how much artifacts use a concept. In the transposed matrix of *R*, each row shows how effective features are in a concept. The matrix R corresponds to the noun probability *p*(noun|concept). Equation 3 with the corresponding probabilities is as follows:

$$\begin{aligned} p(\text{noun}, \text{artifact}) &= p(\text{noun})p(\text{concept}|\text{noun}) \\ p(\text{noun}|\text{artifact}) &= \sum_{c\in concept} p(\text{noun}|\text{concept})p(\text{concept}|\text{artifact}) \end{aligned} \tag{3}$$

- **Step 5** Calculating the similarity between artifacts

  According to the previous steps, a vector is considered for each artifact. This step employs the cosine for calculating the similarity between two artifacts. Let $\vec{v_i}$ and $\vec{v_j}$ denote the feature vectors for two artifacts v_i and v_j, then, the cosines similarity between them is calculated using Eq. 4. This similarity shows the value of the corresponding elements in the semantic matrix. The resultant matrix is named the semantic matrix. Figure 8 shows these steps.

$$\cos(\alpha) = \frac{\overrightarrow{v_i}.\overrightarrow{v_j}}{|\overrightarrow{v_i}|.|\overrightarrow{v_j}|}. \tag{4}$$

3.3 Nominal similarity dependency graph

Artifacts are usually named meaningfully by expert programmers. It has been shown that the similarity of artifacts names indicates their content similarity. In this section, we propose a new graph named the nominal similarity dependency graph. This graph is a weighted graph, whose nodes are artifacts, and the edge weight indicates the nominal similarity between the components of the artifacts. This graph is formally defined as follows:

Suppose a nominal dependency graph is labeled as $NDG = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of n artifacts and $E \subseteq V \times V \times R^+$ so that E is defined as follows:

$$E = \{(v_i, v_j, nns(v_i, v_j) | v_i, v_j \in V, i \neq j, nns(v_i, v_j) = \frac{\| \cap(\vec{v_i}, \vec{v_j}) \|}{\{ \underset{1 \leq a,b \leq n}{max}(\| \cap(\vec{v_a}, \vec{v_b}) \|)\}}\}$$

E is the set of links between artifacts so that $nns(v_i, v_j)$ indicates the normalized nominal similarity value between v_i and v_j. The vector $\vec{v_i}$ is defined as follows:

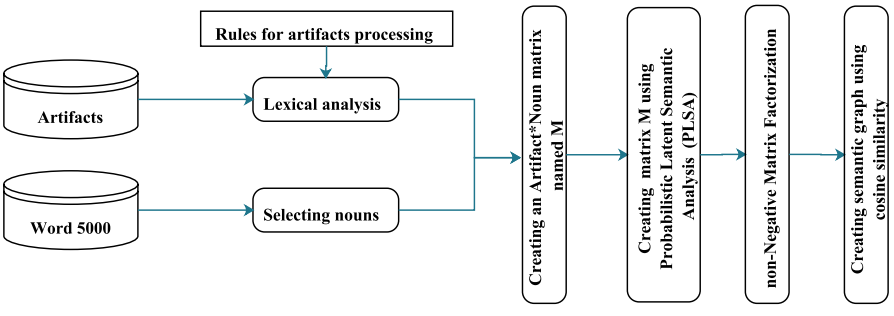


Fig. 8 The steps in creating the semantic dependency graph

$\vec{v_i} = (C_{i1}, C_{i2}, \ldots, C_{ik}), \quad \forall C_{ij} \in$ component of artifact name v_i s.t. $\bigcup_{j=1}^{k} c_{ij} \approx$ artifact name v_i

The following steps are used for the creation of this graph:

- **Step 1** parsing the names of artifacts according to capital letters, numbers, and separators
- **Step 2** converting the name components of an artifact into small case letters (it should be mentioned that separators and numbers are ignored, and capital letters are allocated to the next word.)
- **Step 3** calculating the similarity based on the same components between two artifacts and regarding this value as the weight of an edge between two nodes corresponding to artifacts. Finally, the weight of the edge is divided into the largest weight of the edges; thus the weight of the edges is normalized and mapped to zero and one intervals.

Figure 9 shows the steps in creating this graph. In Fig. 10, these steps are explained by three artifacts to create a part of the nominal similarity graph, before it becomes normal. Figure 11 shows the nominal similarity dependency graph after the normalization process.

3.4 Genetic algorithm for multilingual software system modularization

The multilingual software system modularization aims to put artifacts (i.e., files) of different programming languages into the same module (subsystem or folder). As mentioned in the introduction, the number of programming languages used to implement the files in the Accessible folder of Mozilla Firefox released in July 2017 are 200 files which are written in 29 different programming languages.

The genetic algorithm is a search-based algorithm with global behavior. In this algorithm, a search space is created in each step to select the best solutions (generations) based on the quality function and convey them to the next generation (step). The most important operators of the genetic algorithm are encoding method and quality function.

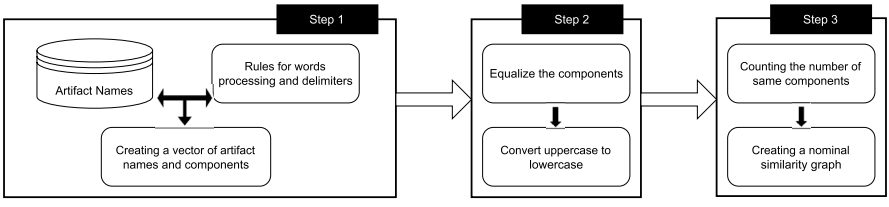


Fig. 9 The steps in creating the nominal similarity graph

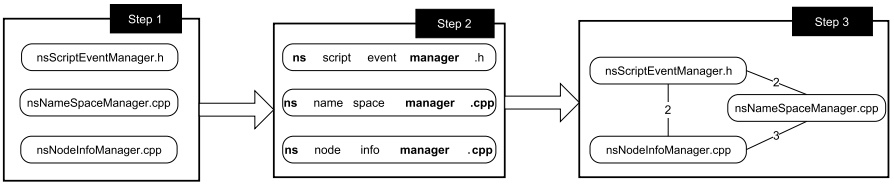


Fig. 10 An example of calculating the nominal similarity

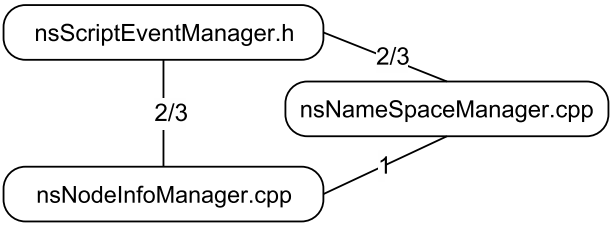


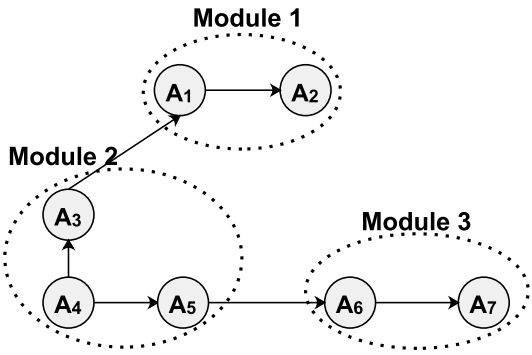
Fig. 11 The nominal similarity dependency graph obtained after normalizing Fig. 10

3.4.1 Encoding

Genetic algorithms operate on set individuals (other names: strings, chromosomes, solutions), where each individual is an encoding of the problem's input data. The encoding used in this paper was first introduced by Mitchell in his Ph.D. thesis in 2002 [31]. In this method, each chromosome is like an array. The number of its genes is equal to the number of vertices (artifacts) in the dependency graph. The content of each gene is the module number. This value ranges between one and the number of vertices of the graph. In this encoding, an artificial number is assigned to each gene, regardless of other genes. Formally, an encoding is defined as:

$$P = p_1 \quad p_2 \quad p_3 \quad p_4 \quad \cdots \quad p_N$$

Fig. 12 Modularization of a sample string P = 1122233



where N is the number of artifacts and p_i, where $1 \leq i \leq N$, identifies the module that contains the ith artifact (i.e., the ith node of the graph). For example, Fig. 12 illustrates a graph in which P = 1122333 is coded. In this encoding, the first number in P, i.e., 1, indicates that the first artifact, A_1, is located in the module labeled 1. Likewise, the second artifact, A_2, is also located in the module labeled 1, the third artifact, A_3, is located in the module labeled 2, and so on.

3.4.2 The quality function

The quality function or objective function is one of the operators having a great effect on the efficiency of the genetic algorithm to find the appropriate solution. The quality function calculates the quality of each module obtained during the evolutionary process of the genetic algorithm. Considering semantic dependency, nominal similarity, and multiple call dependency graphs, this paper presents a new quality function, named Hybrid MQ (HMQ), for calculating an obtained modularization. The quality of module i, denoted by MF_i, is calculated by Eq. 5.

$$MF_i = \begin{cases} 0 & \alpha_{p,i} = 0 \\ \dfrac{9 \sum_{p=0}^{2} \alpha_{p,i}}{9 \sum_{p=0}^{2} \alpha_{p,i} + 4 \sum_{p=0}^{2} \sum_{j=1}^{k} (\beta_{p,i,j} + \beta_{p,j,i})} & \text{Otherwise} \end{cases} \tag{5}$$

where k is the number of modules. Then HMQ for a modularization is calculated as Eq. 6.

$$\text{HMQ} = \sum_{i=1}^{k} MF_i \tag{6}$$

p ranges from zero to two. Zero indicates the semantic graph, one shows the nominal similarity graph, and two shows the CDG. Furthermore, $\alpha_{p,i}$ show the summation of semantic similarity (i.e., p = 0), nominal similarity (i.e., p = 1), and the number of calls (i.e., p = 2) of all the artifacts inside module i. If there is no any dependency among artifacts located into module i, $\alpha_{p,i}$ will be equal to zero. $\beta_{p,i,j}$ indicates the summation of semantic similarity, nominal similarity, the number of calls of all artifacts between the modules i and j. Finally, MF_i shows the quality of module i, and HMQ indicates the total modularization quality, in which 4, 9 was obtained from the experimental tests. If there is no dependency, all of the elements of that matrix are considered zero. It should be mentioned that standardization before any of these graphs is used. In other words, if the total summation of edges is equal to γ in an adjacency matrix named M, then the standard adjacency matrix (indicated by NM) is $NM = (\frac{1}{\gamma} M)$.

The rationale behind this quality function is to maximize the number of relationships within a module, called cohesion, and minimize the number of relationships between modules, called coupling. In the proposed quality function, α_i denotes the

relationships within module i and $\beta_{i,j}$ denotes the relationships between module i and module j. The goal is to maximize this quality function by genetic algorithm.

3.4.3 Genetic operators

In the genetic algorithm, the population is continually updated at each step, and this behavior of the algorithm prevents being trapped in the local optima. For large search spaces, GA is a perfect opportunity to find a global optimum. There are two important aspects of the randomness of the genetic algorithm, which is the selection of the population and the reproduction of new offspring. Each offspring or new modularization is not directly created from the previous modularization. In crossover operation, two individuals that called parents or past modularization create a new offspring (new modularization) and in the mutation, the genes of some of the chromosomes are randomly changed, in other words, the modules of some artifacts changes and mutation operator prevents the algorithm from getting trapped in the local optima. Crossover and mutation are used in random reproduction that occurs after random selection from parents or previous modularization. In the crossover operation, two parents are selected randomly based on the fitness function, then six random numbers have generated that show the cutting points in the parent to divide the parent into seven parts and offspring are reproduced through multi-point crossover method. In the mutation, according to the mutation rate, the number of parents is selected and the seven genes are randomly changed, in other words, in the new modularization, the seven artifacts are randomly assigned to other modules.

4 Experimental setup

This section gives detailed information about the experimental setup to evaluate the proposed algorithm.

4.1 Software system

We selected Mozilla Firefox 3.7 (released in Oct 14, 2015) as a large-scale and open-source application for experimental evaluation and comparison. Different versions of this application are available from 2007. The reason we chose this software is that, as compared to other applications, the largest number of programming languages used to write it. C++, JavaScript, C, HTML, Rust, XML, Python, Java, Assembly and other programming languages are used to implement this application. It is not possible to modularize this application with current methods because heterogeneous programming languages are employed to implement it. The source code of the selected version for experiments contains nearly 5 million lines of code including approximately 22% of comments. We selected ten folders from this application. The file is considered as an artifact. There are 10,245,637 words in Mozilla Firefox. This number of words cannot be managed, so we were able to manage them

Table 2 The name of folders, number of files, number of expert modules, Lines of Code (LOC), number of comments, and languages used in each folder of Mozilla Firefox released in Oct 14, 2015

Folder name	#File	#Modules	LOC	Comments	#Programming language used
Accessible	179	8	57568	15993	10
Browser	45	4	82643	24076	16
Build	21	2	21104	4945	15
Content	881	13	297054	78152	13
Db	97	4	95202	51864	8
Dom	163	5	345616	20648	11
Extensions	179	13	55243	16933	15
Gfx	342	7	222623	46355	12
Intl	573	7	60143	30264	8
Ipc	391	4	157005	38560	8

using word5000 (step 1, Sect. 3.1) with some loss of accuracy. Table 2 shows the specifications of the first ten folders of Mozilla Firefox. Note that there are forty-nine programming languages with twenty-eight million lines of code in the version updated in July, 2019.

4.2 Research questions

To evaluate the effectiveness of the proposed graphs, we answer the following research questions.

RQ1. How close is the semantic, nominal, call dependency graphs and their combinations to the clustering provided by an expert on Mozilla Firefox software?

RQ2. If the proposed graphs are substituted instead of the call graph, how much will improvement in different criteria (see Sect. 4.4)?

RQ3. Which of the proposed dependency graphs produces modularization solution closer to the domain expert clustering by taking into account different experiments and different criteria?

To answer these research questions, we modularize the ten folders of Mozilla Firefox application using the proposed algorithm and some existing algorithms.

4.3 Obtaining an authoritative decompositions

Evaluating the effectiveness of a modularization algorithm is an important problem. To evaluate the effectiveness of an automatic modularization algorithm, the authoritative decomposition (ground-truth structure) is used. There are methods to prepare a ground-truth structure from the source code, for example, directory structure is usually used to show ground-truth structure of a software system. The proximity of the modularization produced by an algorithm to the clustering has been constructed by an expert in this system (e.g., the system architect or a senior developer)

shows the satisfactory fulfillment of that modularization algorithm. In this paper, we selected Mozilla Firefox to assess the proposed algorithm, because whose authoritative decomposition (i.e., directory structure) is there. The directory structure specification of Mozilla Firefox is shown in Table 2. For example, the Accessible folder has 179 files that has been assigned by Mozilla Firefox's developers to 8 clusters (folders). If we consider these 179 files flat, we measure how much the modularization produced by an automatic algorithm is close to the Mozilla Firefox's directory structure (see next sub-section).

4.4 Evaluation metrics

In this paper, MoJo [54], Precision/Recall [1], FM [1], MoJoFM [55] measures are selected to evaluate the proposed modularization algorithm, considering different dependency graphs. These metrics compare the modularization obtained by an algorithm with directory structure of the Mozilla Firefox. In the following, these metrics are briefly described. "The MoJo calculates the number of move and join operations needed to reach from one modularization to another, and is defined as a distance measure" [1]. Precision and Recall are calculated using Eqs. 7 and 8. These metrics to compute the closeness between two modularizations consider the intrapairs (pairs of artifacts) and is not sensitive to the size and number of modules. This, in some cases, makes the metrics unable to accurately compute the closeness between the two modularizations. To reduce this undesirable aspect, the harmonic mean of Precision and Recall, called FM, is used (Eq. 9).

$$\begin{aligned}\text{Precision} = (&\{\text{“Number of co-module pairs in the ground-truth structure”}\}\\ &\cap \{\text{“Number of co-module pairs in the extracted structure”}\})\\ &/\{\text{“Number of co-module pairs in the extracted structure”}\}\end{aligned} \tag{7}$$

$$\begin{aligned}\text{Recall} = (&\{\text{“Number of co-module pairs in the ground-truth structure”}\}\\ &\cap \{\text{“Number of co-module pairs in the extracted structure”}\})\\ &/\{\text{“Number of co-module pairs in the ground-truth structure”}\}\end{aligned} \tag{8}$$

$$FM = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9}$$

Let $mno(A, B)$ capture the distance value between two modularizations as the minimum number of Move and Join operations required to move from one modularization, A, to another, B. The MoJoFM [55], calculated by Eq. 10, is a more accurate metric than the rest of the metrics.

$$MoJoFM(A, B) = 1 - \frac{mno(A, B)}{max(mno(\forall A, B))} \times 100\% \tag{10}$$

Table 3 The parameter setting for experiments

Parameter	Values
Population size	$10n$ (where n denotes the number of artifacts)
Generation	$200n$
Crossover rate	0.8
Mutation rate	0.05
Selection	Roulette wheel selection
Crossover operation	Multi-point
Mutation operation	randomly changed a gene

Table 4 Abbreviations used in the genetic algorithm with different dependency graphs

Abbreviation	Algorithm	Used graph
SGA	Genetic	Semantic similarity graph
SNGA	Genetic	Semantic similarity + nominal similarity graph
SDGA	Genetic	Semantic similarity + call dependency graph
SNDGA	Genetic	Semantic similarity + nominal similarity + call dependency graph

Note that the value produced by Precision, Recall, FM, and MoJoFM are always in the range from 0 to 100, whereas the value produced by MoJo is not within a specific interval. Also, smaller value of MoJo, and larger values of other metrics indicate that the resultant modularization is similar to the authoritative decomposition.

4.5 Parameter setting

In the modularization process, the execution conditions of the algorithms were considered the same in the number of runs to calculate the mean and the best result. Bunch, ECA, MCA, SGA, SNGA, SDGA, and SNDGA were executed ten times. The rates of crossover and mutation, the repetitions of algorithms and termination conditions were considered the same in all algorithms. Let n denote the number of artifacts; the parameter settings of each algorithm are given in Table 3.

5 Results and discussions

In this paper, we use the genetic algorithm with different graphs for software modularization. Table 4 shows the name of the algorithms and their explanation used in this paper. Bunch (single-objective), ECA (multi-objective), and MCA (multi-objective) are three state-of-the-art modularization algorithms. These algorithms use CDG, i.e., a structural dependency, extracted from the source code. Therefore, they

can not cover the entire source code. These algorithms can extract high-quality modules for applications written in a single language or homogeneous multi-languages; therefore, they are selected for comparison. It is important to note that only homogeneous languages in Mozilla Firefox have been used to construct the CDG.

To answer the research question RQ1, the algorithms Bunch, ECA, MCA, SGA, SNGA, SDGA, and SNDGA are executed on each folder of Mozilla Firefox by ten independent runs in terms of MoJo, Precision, Recall, FM and MojoFM. Tables 5, 6, 7, 8 and 9 show the results. The CDG depends on the syntax of the programming language, whereas the semantic graph and the nominal similarity graph do not depend on the programming language and any assumptions of the programming language structure. The best results in these tables are highlighted in bold. In the following, we examine the results in detail.

In Table 5, for the MoJo, the proposed algorithms have better performance in eight folders, but in the Browser folder and Db, the MCA algorithm is better than other algorithms. In Table 6, for the Precision, in seven folders, the proposed algorithms are better than structural-based algorithms, but in the Accessible and the Browser folders, the MCA algorithm is better. In the Db folder, the average precision of the MCA algorithm is better than all algorithms, and the best result in the Gfx folder belongs to the MCA algorithm. In Table 7, in the Recall, the proposed algorithms perform better in seven folders, but Bunch in the Browser folder, and ECA in the Build folder, and MCA in the Db folder performs better (with a small difference). In Table 8, in terms of FM, which shows the harmonic mean of Precision and Recall, the proposed algorithms are better in eight folders, but in the Browser and Db folders the MCA algorithm is better, and the best FM result in the Gfx folder belongs to the MCA algorithm, but in the average, the proposed methods performed better. In Table 9, for the MoJoFM, in the Browser and Db folders, the MCA algorithm performs better (with a small difference); in the other eight folders, the proposed algorithms perform better.

Tables 5, 6, 7, 8 and 9 confirm that the semantic and nominal graphs are better than the graphs which depend on the syntax in terms of quality of modularization. Also, it is confirmed that the proposed algorithm resulted in better improvements on the combination of semantic and nominal similarity graphs with CDG in folders with more artifacts in comparison with smaller artifacts. Analyzing these tables by considering the size of folders show that improvements in the simultaneous use of the proposed graphs and CDG are 89.79%, 85.88%, 86.63%, and 59.23%, respectively, in terms of Precision, Recall, FM, and of MoJoFM. The comparison of MoJo, Precision, Recall, FM, and MoJoFM demonstrate the high importance of the semantic graph and the nominal similarity graph. It is also very effective to use them independently or in combination.

To answer the research question RQ2, Fig. 13 depicts the improvement in various criteria when different dependency graphs, i.e., semantic graph (SG), combination of semantic and nominal graphs (SNG), combination of semantic and call dependency graph (SDG), and combination of semantic, nominal similarity, call dependency graph (SNDG) are replaced instead of the call dependency graph (CDG) in the modularization algorithm. The means of modularization quality of different criteria are calculated in all folders in different executions (positive or negative growth).

Table 5 Comparing the different algorithms in terms of MoJo

Folder name	Bunch		ECA		MCA		SGA		SNGA		SDGA		SNDGA	
	Best MoJo	Average MoJo	Best MoJo	Average MoJo	Best MoJo	Average MoJo	Best MoJo	Average MoJo	Best MoJo	Average MoJo	Best MoJo	Average MoJo	Best MoJo	Average MoJo
MoJo metric														
Accessible	99	105.875	107	109.4	103	109.2	108	107.400	**58**	69.000	70	86.333	60	**67.571**
Browser	12	14.375	16	19.38	**11**	**13**	17	19.750	14	14.200	14	17.428	13	14.285
Build	4	4.286	4	4	4	4	4	4.000	4	4.286	**3**	4	**3**	4
Content	598	608.170	630	634.17	540	543.8	651	658.670	351	406.000	474	498.8	**189**	**245.6**
Db	5	6.500	**3**	4	**3**	**3**	5	7.000	5	7.000	4	6.285	5	6.714
Dom	66	72.330	75	82.67	73	84.67	86	93.330	42	63.330	56	70.142	**36**	**39.714**
Extensions	84	93.170	69	89.83	78	93	107	112.500	88	96.670	**55**	**65.428**	58	**64.428**
Gfx	152	167.830	131	152.83	109	143.5	143	153.330	76	90.660	89	103.666	**52**	**65.714**
Intl	116	135.5	143	143	93	109.83	124	132.830	136	139.670	**73**	**85.4**	82	111.5
Ipc	73	75.000	75	75	75	75	73	75.000	48	60.830	37	42.571	**35**	**44**

Table 6 Comparing the different algorithms in terms of Precision

Folder name	Bunch		ECA		MCA		SGA		SNGA		SDGA		SNDGA	
	Best precision	Average precision	Best precision	Average precision	Best precision	Average precision	Best precision	Average precision	Best precision	Average precision	Best precision	Average precision	Best precision	Average precision
Precision metric														
Accessible	0.2709	0.232	0.2306	0.1433	**0.8522**	**0.6205**	0.4601	0.4551	0.3928	0.3567	0.3858	0.3048	0.4377	0.3576
Browser	0.8329	0.6177	0.3343	0.3021	**0.9972**	**0.9498**	0.5552	0.5012	0.5071	0.4651	0.5637	0.3699	0.5637	0.4083
Build	0.4744	0.4146	0.5064	0.4853	0.5705	0.5275	0.4872	0.4289	0.641	0.541	0.5833	0.5077	**0.7115**	**0.6112**
Content	0.1215	0.118	0.0983	0.0941	0.2337	0.2282	0.208	0.129	0.392	0.2993	0.4596	0.4131	**0.5247**	**0.4219**
Db	0.5312	0.3768	0.9161	0.8858	0.9161	**0.9161**	0.4961	0.389	0.495	0.4609	**0.9575**	0.847	0.5046	0.3071
Dom	0.5029	0.4107	0.2652	0.2424	0.4357	0.4032	0.3883	0.3541	0.5549	0.477	0.6268	0.4731	**0.7503**	**0.5641**
Extensions	0.2635	0.2642	0.2556	0.1856	0.3453	0.2826	0.3696	0.3628	0.2299	0.216	0.3728	0.3159	**0.389**	**0.3339**
Gfx	0.2534	0.2334	0.2581	0.2110	**0.6520**	0.4275	0.482	0.3821	0.4198	0.3821	0.5184	**0.466**	0.5132	0.3963
Intl	0.4483	0.3203	0.1472	0.1467	0.4862	0.4672	0.732	0.6821	0.2289	0.2111	**0.7505**	**0.6868**	0.262	0.2149
Ipc	0.5014	0.2877	0.2545	0.2516	0.2599	0.2571	0.6303	0.5468	0.6701	0.589	**0.9716**	**0.7919**	0.5192	0.4143

Table 7 Comparing the different algorithms in terms of Recall

Folder name	Bunch		ECA		MCA		SGA		SNGA		SDGA		SNDGA	
	Best recall	Average recall	Best recall	Average recall	Best recall	Average recall	Best recall	Average recall	Best recall	Average recall	Best recall	Average recall	Best recall	Average recall
Recall metric														
Accessible	0.3141	0.2261	0.2748	0.2512	0.2889	0.2380	0.3105	0.2938	0.553	0.4769	0.4262	0.3336	**0.5598**	**0.5082**
Browser	**0.649**	**0.5549**	0.5108	0.4616	0.5592	0.5149	0.3546	0.334	0.5169	0.5097	0.5808	0.4449	0.5597	0.5193
Build	0.74	0.7363	0.79	**0.7571**	**0.8235**	0.7473	0.75	0.7381	0.7054	0.694	0.7429	0.7108	0.7429	0.7064
Content	0.1819	0.1754	0.1681	0.1610	0.1860	0.1855	0.2561	0.2081	0.4371	0.3821	0.1669	0.1654	**0.5757**	**0.5083**
Db	0.9652	0.9376	**0.9776**	0.9769	**0.9776**	**0.9776**	0.9233	0.9182	0.9324	0.9265	0.9691	0.9351	0.9215	0.9166
Dom	0.4345	0.4049	0.4041	0.3694	0.3777	0.3544	0.3131	0.3088	0.7041	0.6213	0.3545	0.343	**0.7136**	**0.6746**
Extensions	0.3462	0.3031	0.5114	0.3711	0.3885	0.2744	0.1815	0.175	0.3138	0.2971	**0.5376**	0.4466	0.48	**0.4519**
Gfx	0.4839	0.3914	0.6755	0.4770	0.6246	0.4293	0.5237	0.4998	0.7101	0.6771	0.6003	0.4992	**0.7953**	**0.7195**
Intl	0.7409	0.7195	0.6164	0.6145	0.7515	0.7244	0.7172	0.7091	0.6528	0.595	**0.8703**	**0.8267**	0.8155	0.7434
Ipc	0.702	0.6919	0.6997	0.6919	0.6907	0.6773	0.7848	0.7262	0.8477	0.7721	**0.8632**	**0.833**	0.8373	0.8139

Table 8 Comparing the different algorithms in terms of FM

Folder name	Bunch		ECA		MCA		SGA		SNGA		SDGA		SNDGA	
	Best FM (%)	Average FM (%)	Best FM (%)	Average FM (%)	Best FM (%)	Average FM (%)	Best FM (%)	Average FM (%)	Best FM (%)	Average FM (%)	Best FM (%)	Average FM (%)	Best FM (%)	Average FM (%)
FM metric (%)														
Accessible	27.35	21.34	23.88	19.88	35.89	31.55	33.9	33.31	42.1	38.9	37.83	31.53	**45.31**	**41.67**
Browser	**72.95**	57.48	40.41	36.52	70.90	**66.75**	41.39	39.81	47.15	47.14	46.44	38.44	54.92	44.42
Build	58.96	52.04	61.72	59.15	65.12	61.78	60.87	55.02	67.11	62.04	85.25	63.11	**85.25**	**68.41**
Content	14.95	14.08	12.40	11.88	20.72	20.46	16.91	14.96	36.26	33.99	24.51	23.25	**52.17**	**45.68**
Db	67.46	53.41	94.58	92.91	**94.58**	**94.58**	64.44	54.63	64.29	53.20	93.7	88.32	65.04	45.31
Dom	44.87	40.37	32.03	29.77	40.46	37.60	34.35	32.03	60.54	53.12	41.99	38.94	**68.52**	**65.25**
Extensions	29.93	28.15	34.09	24.74	35.33	27.68	24.34	23.58	25.52	24.98	40.96	35.78	**41.47**	**38.21**
Gfx	33.03	28.99	37.35	29.22	**62.91**	40.85	36.46	32.91	49.93	47.02	54.74	47.17	60.42	**50.92**
Intl	55.65	43.87	23.77	23.70	59.04	56.77	73.42	67.91	33.46	30.16	**79.23**	**74.16**	37.39	33.09
Ipc	57.75	39.48	37.33	36.90	37.48	37.27	62.12	49.21	58.59	49.82	**89.23**	**80.11**	64.02	53.89

Table 9 Comparing the different algorithms in terms of MoJoFM

Folder name	Bunch		ECA		MCA		SGA		SNGA		SDGA		SNDGA	
	Best MoJoFM (%)	Average MoJoFM (%)	Best MoJoFM (%)	Average MoJoFM (%)	Best MoJoFM (%)	Average MoJoFM (%)	Best MoJoFM (%)	Average MoJoFM (%)	Best MoJoFM (%)	Average MoJoFM (%)	Best MoJoFM (%)	Average MoJoFM (%)	Best MoJoFM (%)	Average MoJoFM (%)
MoJoFM metric														
Accessible	42.11	39.4	37.43	36.02	39.77	36.14	57.01	52.81	**67.0**	**64.98**	59.06	50.1	64.91	61.11
Browser	70.0	65.36	60.00	51.88	**72.50**	**67.50**	55.0	52.0	65.0	63.88	65.0	57.5	67.5	64.29
Build	84.21	78.95	78.95	78.95	78.95	78.95	**84.21**	**81.01**	**84.21**	79.45	**84.21**	78.95	**84.21**	78.95
Content	31.11	30.23	27.42	26.94	37.79	37.35	27.12	26.21	60.86	54.93	45.39	43.03	**78.23**	**73.02**
Db	94.68	93.41	**96.81**	95.74	**96.81**	**96.81**	94.68	92.99	94.68	93.38	95.74	93.62	94.68	92.86
Dom	58.23	55.19	52.53	47.68	53.80	46.42	45.57	43.01	73.42	68.96	64.56	56.22	**77.22**	**74.96**
Extensions	49.70	45.51	58.68	46.21	53.29	44.31	35.93	33.68	47.31	44.12	**67.07**	**63.83**	65.27	62.40
Gfx	54.49	50.48	60.78	54.24	67.37	57.04	60.09	57.21	78.29	75.42	73.35	70.30	**84.43**	**80.89**
Intl	79.51	76.4	74.73	74.73	83.57	**80.60**	80.03	78.03	78.17	76.94	78.1	75.65	**85.51**	79.76
Ipc	81.14	80.72	80.62	80.62	80.62	80.62	83.04	81.93	88.62	85.11	90.44	88.16	**90.96**	**89.60**

The positive growth shows the improvement percentage of modularization quality. According to this comparison, the combination of the semantic graph and nominal similarity graph improves the modularization quality compared to the semantic graph. This graph does not depend on the syntax of the programming language. When these graphs are combined with CDG, all evaluation criteria are significantly improved.

Figure 14 depicts a significant decrease in MoJo. 2871 artifacts have been used. The total number of MoJo is 1209 for all artifacts in Bunch. This value is reduced to 533 in the proposed algorithm on the SNDG. Therefore, modularization quality improves from 57.8% to 81.4%. This improvement is also confirmed in Precision, Recall, and FM (Tables 5, 6, 7, 8, 9).

Figures 15, 16, and 17 display the internal state of the fourth module of the Content folder. The Content folder has 881 artifacts and 13 modules. These figures show the number of artifacts and their relationships in the call dependency graph, semantic dependency graph, and nominal similarity dependency graph. We have two reasons for displaying these graphs: (1) we intend to show the magnitude of the dependency graphs. These figures show that the semantic dependency graph is much larger than the nominal dependency graph and the nominal dependency graph is much larger than the call dependency graph; and (2) these graphs are only for the fourth module of Content folder, while this folder has 13 modules. Therefore, the dependency graph generated for all modules of this folder will be too large and cannot be modularized with deterministic algorithms and requires high-performance algorithms.

The time required for the convergence of the semantic dependency graph is 1.47 times the amount required in the call dependency graph. Given a large number of edges, it seems logical. The time required in the combination graph is 1.07 times in comparison with the semantic graph. However, convergence is achieved quicker in the combination of semantic and structural graphs.

To answer the research question RQ3, having different criteria and algorithms, analyzing the algorithms and determining which algorithms perform well on all criteria, is difficult. In such cases, Multi-Criteria Decision Making (MCDM) can be used [56]. MCDM analyzes and evaluates the performance of different algorithms

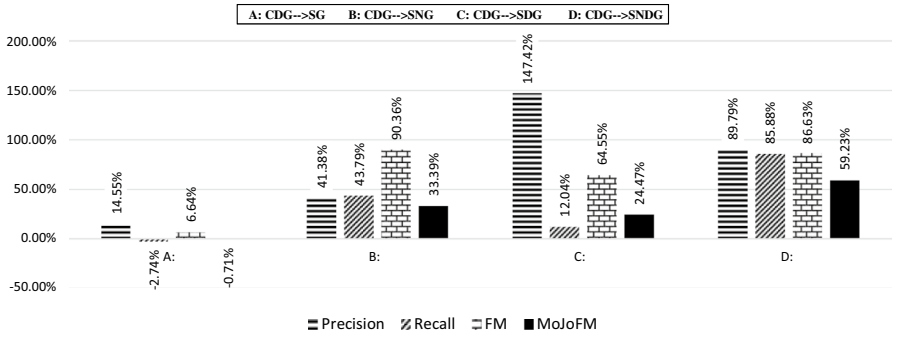


Fig. 13 Comparing the average improvement in four criteria by replacing different graphs in modularization algorithms

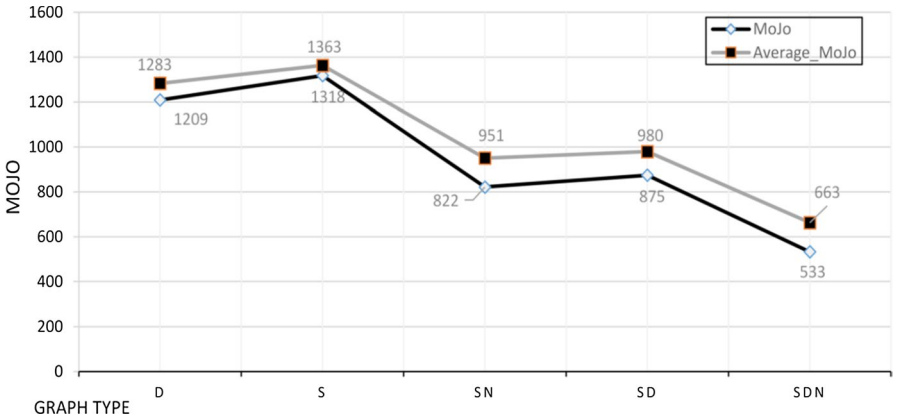


Fig. 14 Comparing MoJo criteria for different graphs considering all folders

and determines how well each algorithm performs against other algorithms based on different criteria. In the MCDM, the behavior of different algorithms is evaluated based on different criteria, and the algorithm's efficiency is in the range of zero and one. To do this, first, a matrix, named X, is created in n×m dimension, where n is the number of algorithms and m is the number of criteria; then the matrix X is normalized, and the matrix P is created, where the values of the criteria are placed in the interval 0 and 1, then, based on information theory (entropy) each criterion are calculated. Based on entropy, the weight of each criterion is calculated, and the importance of the criteria is determined; then the maximum and minimum of each criterion are calculated, and then the distance between each algorithm is calculated from the positive and negative ideal, and finally, the efficiency of each algorithm is obtained. The efficiency value is close to zero, indicating poor performance and close to one, shows the decisive performance of the algorithm in comparison with other algorithms.

Figure 18 demonstrates the efficiency and effectiveness of the algorithms. This figure shows that in Accessible, Build, Content, Dom, Extensions, Gfx, Intl and Ipc folders, the SDGA, and SNDGA algorithms have impressive performance based on the four criteria: Precision, Recall, FM, and MojoFM; and the difference in performance for folders with the largest number of artifacts in all cases is very impressive (Content with 881 files, Intl with 573 files, Ipc with 391 files, Gfx with 342 files and Extensions with 179 files). The total number of files used is 3013 files which placed in 10 folders, so that 2781 files of them placed into 8 folders. The results of the proposed algorithm are significantly better than the rest of the algorithms, which is more than 95% of the files. In the Db folder with 97 files, the performance of all algorithms and their effectiveness are close, due to the low structural and semantic dependency of the contents of the files to each other. In the Browser folder, the call dependency between files is high, which is why the MCA performs well with other algorithms.

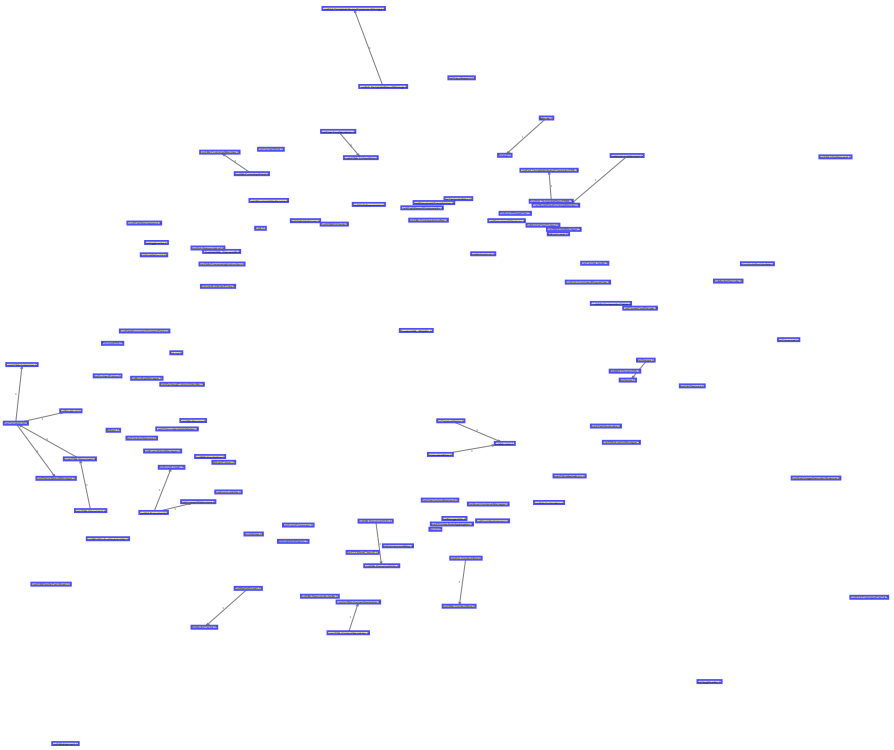


Fig. 15 The Call Dependency Graph generated for the fourth module of the Content folder

6 Threats to validity

To explain the shortcomings and strengths of the proposed modularization approach, we explore the outside circumstances that could influence the validity of the achieved results. Threats to validity are classified into external validity and internal validity. External validity is concerned with extent to which the design of the study allows us to generalize the results to populations other than that from which the sample was drawn, or to similar populations in different settings or at different times.

- In search-based software engineering, generalizing an approach to the broad perspective of problems is a very important threat to the validity of results because of the large number of diverse software systems available to any study of software modularization. To handle this threat, we chose Mozilla Firefox application for two reasons: first, the largest number of programming languages used to write it compared to other software systems; second, the number of artifacts in its folders varies greatly (from 21 to 881 artifacts). It is important to note that there are only a few software that has more than 881 artifacts (here files) in a folder.

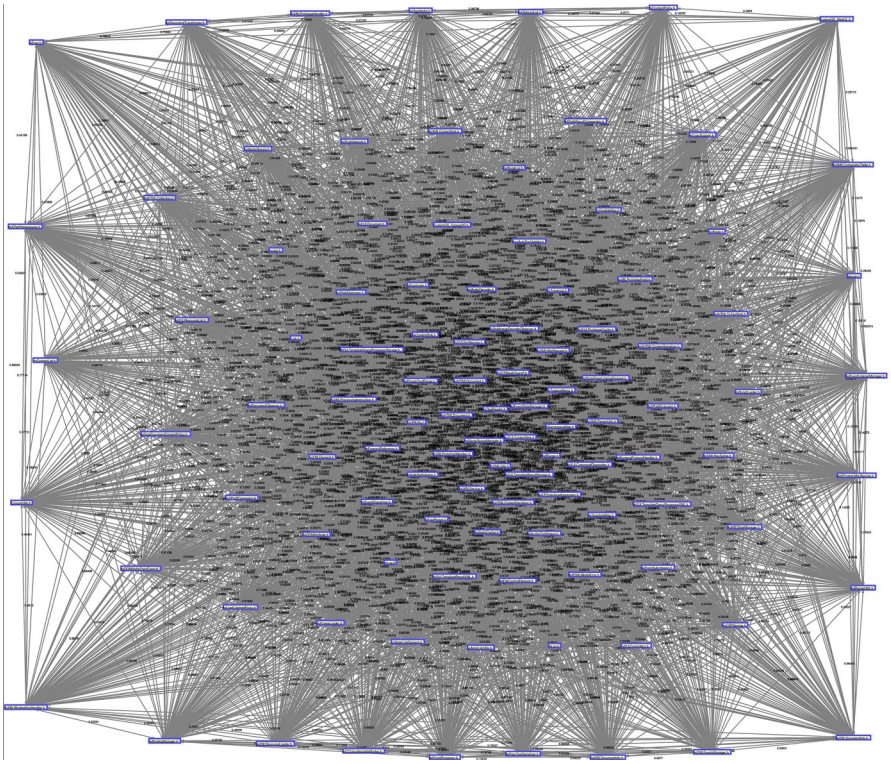


Fig. 16 The semantic dependency graph generated for the fourth module of the Content folder

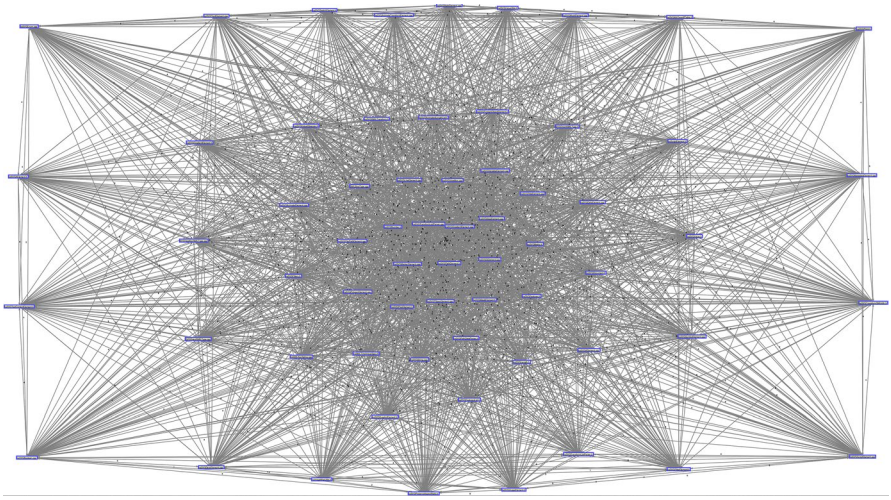


Fig. 17 The nominal similarity dependency graph generated for the fourth module of the Content folder

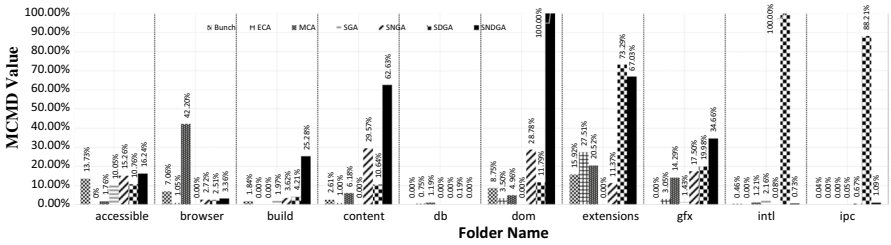


Fig. 18 Multi-Criteria Decision Making for different dependency graphs

- Because the proposed approach considers structural and non-structural features for modularization, it is thus able to modularize the programs written with several languages. However, the use of structural information is more logical than non-structural features in the field of software modularization. It is important to note that, in the absence of structural features, the modularization quality achieved for such software will be low.
- In addition to structural features, the proposed method depends on non-structural features such as identifier names and comments. If in a program, identifier names are not meaningful, and the comments are either not written or well-written, the modularization quality will be low.

Internal validity refers specifically to whether an experimental treatment/condition makes a difference to the outcome or not, and whether there is sufficient evidence to substantiate the claim. Threats internal validity of the paper are:

- In this paper, we used MoJo, Precision, Recall, FM, and MoJoFM criteria to assess the proposed algorithms. These criteria are used to compare the difference between two modularizations. The MoJo and MoJoFM use a greedy method to compute the difference between two modularizations; thus, they may not be able to calculate the exact difference. Precision, Recall, and FM are very sensitive to the artifacts' co-module and are not considered the number of modules in calculating the difference between two modularizations. All these criteria do not consider edges between artifacts in calculating similarity.
- The rate of crossover and mutation operators used in the genetic algorithm, and also the coefficients (i.e., 4 and 9) used in Eq. 5 are obtained using the extensive experiments on the Accessible folder. Then, these rates and coefficients are tested on nine other folders. However, these numbers may not work well on another software system.
- The nominal similarity dependency graph is dependent on a proper naming convention. Well-chosen identifiers by developers will produce the best results, while the bad naming (for example, generic name, arbitrary name, and abbreviated names) is one of the main threats of this algorithm.

7 Conclusion

Source code understanding is an essential role in the software maintenance process. In this paper, we presented an approach to understand multilingual programs. To this end, we presented three dependency graphs named CDG, SDG, and NDG. The CDG shows the invocation relationships between software artifacts. It does not cover most of the features in the multilingual program for modularization. The SDG is created according to all semantics contained in the vocabulary of source code. The NDG is constructed based on the similarity between identifier names in the source code. The CDG covers some structural features of the source code and SDG and NDG cover some non-structural features of the source code. Both SDG and NDG are completely independent of the syntax of a programming language. Then a genetic algorithm has been adopted with a new objective function. It can consider the combination of three types of graphs in the modularization. The experimental results on a multilingual software system, named Mozilla Firefox, demonstrated that the combination of syntax independent graphs with syntactic graphs such as the CDG improves the modularization quality; therefore, a reliably automated modularization can be promised.

7.1 Future work

The following are recommendations and suggestions for future works:

- Building a thesaurus database for programming languages involves creating conceptual records to support concepts and their relationships
- Proposing a new objective function which is based on the information theory instead of HMQ to modularize software systems,
- Addressing McCabe's cyclomatic complexity on modularization quality in multilingual programs,
- Proposing a parallel genetic algorithm to modularize the proposed dependency graphs,
- Using other heuristic algorithms such as [57, 58] instead of genetic algorithm.

References

1. Isazadeh A, Izadkhah H, Elgedawy I (2017) Source code modularization: theory and techniques. Springer, Berlin
2. Praditwong K, Harman M, Yao X (2011) Software module clustering as a multi-objective search problem. IEEE Trans Softw Eng 37(2):264–282
3. Mitchell BS, Mancoridis S (2006) On the automatic modularization of software systems using the bunch tool. IEEE Trans Softw Eng 32(3):193–208
4. Andritsos P, Tzerpos V (2005) Information-theoretic software clustering. IEEE Trans Softw Eng 31(2):150–165

5. Mayer P, Bauer A (2015) An empirical analysis of the utilization of multiple programming languages in open source projects. In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, ACM, p 4
6. Mayer P, Kirsch M, Le MA (2017) On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. J Softw Eng Res Dev 5(1):1
7. Chen Y-F, Gansner ER, Koutsofios E (1998) A c++ data model supporting reachability analysis and dead code detection. IEEE Trans Softw Eng 24(9):682–694
8. Mancoridis S, Souder TS, Chen YF, Gansner ER, Korn JL (2001) Reportal: a web-based portal site for reverse engineering. In: 2001 Proceedings of Eighth Working Conference on Reverse Engineering, IEEE, pp 221–230
9. Chen YF, Fowler GS, Koutsofios E, Wallach RS (1995) Ciao: a graphical navigator for software and document repositories. In: 1995 Proceedings of International Conference on Software Maintenance, IEEE, pp 66–75
10. Kapdan M, Aktas M, Yigit M (2014) On the structural code clone detection problem: a survey and software metric based approach. In: International Conference on Computational Science and Its Applications, Springer, pp 492–507
11. Raza A, Vogel G, Plödereder E (2006) Bauhaus-a tool suite for program analysis and reverse engineering. In: Ada-Europe, vol 4006, Springer, pp 71–82
12. Müller HA, Tilley SR, Wong K (1993) Understanding software systems using reverse engineering technology perspectives from the rigi project. In: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, vol 1, IBM Press pp 217–226
13. Bischofberger WR (1992) Sniff: A pragmatic approach to a C++ programming environment. In: C++ Conference. Chicago, pp 67–82
14. T. C. Group. http://thechiselgroup.org/shrimp/
15. Yadav R, Patel R, Kothari A (2014) Reverse engineering tool based on unified mapping method (retum): class diagram visualizations. J Comput Commun 2(12):39
16. Reniers D, Voinea L, Ersoy O, Telea A (2014) The solid* toolset for software visual analytics of program structure and metrics comprehension: from research prototype to product. Sci Comput Program 79:224–240
17. Ferenc R, Beszédes Á, Tarkiainen M, Gyimóthy T (2002) Columbus-reverse engineering tool and schema for c++. In: Proceedings of International Conference on Software Maintenance, IEEE, pp 172–181
18. Telea A, Byelas H, Voinea L (2009) A framework for reverse engineering large c++ code bases. Electron Not Theor Comput Sci 233:143–159
19. Schwanke RW (1991) An intelligent tool for re-engineering software modularity. In: Proceedings of 13th International Conference on Software Engineering, IEEE, pp 83–92
20. Terceiro A, Costa J, Miranda J, Meirelles P, Rios LR, Almeida L, Chavez C, Kon F (2010) Analizo: an extensible multi-language source code analysis and visualization toolkit. In: Brazilian Conference on Software: Theory and Practice (Tools Session)
21. Rosenblum DS, Wolf AL (1991) Representing semantically analyzed c++ code with reprise. In: C++ Conference, pp 119–134
22. Matzko S, Clarke PJ, Gibbs TH, Malloy BA, Power JF, Monahan R(2002) Reveal: a tool to reverse engineer class diagrams. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, Australian Computer Society, Inc., pp 13–21
23. Izadkhah H, Elgedawy I, Isazadeh A (2016) E-cdgm: an evolutionary call-dependency graph modularization approach for software systems. Cybern Inf Technol 16(3):70–90
24. Clark M. Jdepend. http://www.clarkware.com/software/JDepend.html. Accessed in March
25. Lanza M (2003) Codecrawler-lessons learned in building a software visualization tool. In: Proceedings of Seventh European Conference on Software Maintenance and Reengineering, IEEE, pp 409–418
26. Auber D, Melancon G, Munzner T, Weiskopf D, et al. (2010) Solidsx: a visual analysis tool for software maintenance. In: Poster Abstracts at Eurographics/IEEE-VGTC Symposium on Visualization
27. Engberts A, Kozaczynski W, Liongosari E, Ning JQ, (1993) Cobol/sre: a cobol system renovation environment. In: Proceeding of the Sixth International Workshop on Computer-aided Software Engineering, CASE'93, IEEE, pp 199–210

28. Ning JQ, Engberts A, Kozaczynski WV (1994) Automated support for legacy code understanding. Commun ACM 37(5):50–58
29. Grant C (1999) Software visualization in prolog. Technical report University of Cambridge, Computer Laboratory
30. Maqbool O, Babri H (2007) Hierarchical clustering for software architecture recovery. IEEE Trans Softw Eng 33(11):759–780
31. Brian SM (2002) A heuristic search approach to solving the software clustering problem. PhD thesis, Drexel University, Thesis
32. Mitchell BS, Mancoridis S (2008) On the evaluation of the bunch search-based software modularization algorithm. Soft Comput Fusion Found Methodol Appl 12(1):77–93
33. Parsa S, Bushehrian O (2005) A new encoding scheme and a framework to investigate genetic clustering algorithms. J Res Pract Inf Technol 37(1):127
34. Mahdavi K, Harman M, Hierons RM (2003) A multiple hill climbing approach to software module clustering. In: Proceedings of International Conference on Software Maintenance, ICSM 2003, IEEE, pp 315–324
35. Mamaghani AS, Meybodi MR (2009) Clustering of software systems using new hybrid algorithms. In: Ninth IEEE International Conference on Computer and Information Technology, CIT'09, vol 1, IEEE, pp 20–25
36. Chhabra JK et al (2017) Harmony search based remodularization for object-oriented software systems. Comput Lang Syst Struct 47:153–169
37. Tajgardan M, Izadkhah H, Lotfi S (2016) Software systems clustering using estimation of distribution approach. J Appl Comput Sci Methods 8(2):99–113
38. Huang J, Liu J (2016) A similarity-based modularization quality measure for software module clustering problems. Inf Sci 342:96–110
39. Chhabra JK et al (2017) Improving modular structure of software system using structural and lexical dependency. Inf Softw Technol 82:96–120
40. Prajapati A, Chhabra JK (2017) A particle swarm optimization-based heuristic for software module clustering problem. Arab J Sci Eng 2017:1–12
41. Huang J, Liu J, Yao X (2017) A multi-agent evolutionary algorithm for software module clustering problems. Soft Comput 21(12):3415–3428
42. Mohammadi S, Izadkhah H (2019) A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code. Inf Softw Technol 105:252–256
43. Van Deursen A, Kuipers T (1999) Identifying objects using cluster and concept analysis. In: Proceedings of the 1999 International Conference on Software Engineering, IEEE, pp 246–255
44. Corazza A, Di Martino S, Maggio V, Scanniello G (2016) Weighing lexical information for software clustering in the context of architecture recovery. Empir Softw Eng 21(1):72–103
45. Maletic JI, Valluri N (1999) Automatic software clustering via latent semantic analysis. In: 14th IEEE International Conference on Automated Software Engineering, IEEE, pp 251–254
46. Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49(3):230–243
47. Maletic JI, Marcus A (2000) Using latent semantic analysis to identify similarities in source code to support program understanding. In: Proceedings of 12th IEEE International Conference on tools with artificial intelligence, ICTAI 2000, IEEE, pp 46–53
48. Maletic JI, Marcus A (2001) Supporting program comprehension using semantic and structural information. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society, pp 103–112
49. Bavota G, De Lucia A, Marcus A, Oliveto R (2010) Software re-modularization based on structural and semantic metrics. In: 7th Working Conference on Reverse Engineering (WCRE), IEEE, pp 195–204
50. Bavota G, Gethers M, Oliveto R, Poshyvanyk D, Lucia Ad (2014) Improving software modularization via automated analysis of latent topics and dependencies. ACM Trans Softw Eng Methodol (TOSEM) 23(1):4
51. Misra J, Annervaz K, Kaulgud V, Sengupta S, Titus G (2012) Software clustering: unifying syntactic and semantic features. In: 19th Working Conference on Reverse Engineering (WCRE), IEEE, pp 113–122
52. Ballatore A, Bertolotto M, Wilson DC (2014) An evaluative baseline for geo-semantic relatedness and similarity. GeoInformatica 18(4):747–767

53. Zenkert J, Klahold A, Fathi M (2018) Knowledge discovery in multidimensional knowledge representation framework. Iran J Comput Sci 1(4):199–216
54. Tzerpos V, Holt RC (1999) Mojo: a distance metric for software clusterings. In: Proceedings of Sixth Working Conference on Reverse Engineering, IEEE, pp 187–193
55. Wen Z, Tzerpos V (2004) An effectiveness measure for software clustering algorithms. In: Proceedings of 12th IEEE International Workshop on Program Comprehension, 2004, IEEE, pp 194–203
56. Tian Z-P, Zhang H-Y, Wang J, Wang J-Q, Chen X-H (2016) Multi-criteria decision-making method based on a cross-entropy with interval neutrosophic sets. Int J Syst Sci 47(15):3598–3608
57. Bodaghi M, Samieefar K (2019) Meta-heuristic bus transportation algorithm. Iran J Comput Sci 2(1):23–32
58. Gandomi AH, Alavi AH (2012) Krill herd: a new bio-inspired optimization algorithm. Commun Nonlinear Sci Numer Simul 17(12):4831–4845

Affiliations

Masoud Kargar[1] · Ayaz Isazadeh[2] · Habib Izadkhah[2]

Masoud Kargar
kargar@qiau.ac.ir

[1] Faculty of Computer and Information Technology Engineering, Qazvin Branch, Islamic Azad University, Qazvin, Iran

[2] Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran