Mining Association Rules from Code (MARC) to Support Legacy Software Management

Christos Tjortjis[1]

[1]*International Hellenic University, School of Science & Technology,*

14th km Thessaloniki – Moudania, 57001 Thermi, Greece

c.tjortjis@ihu.edu.gr

Tel: +30 2310807576

Fax: +30-2310474590

Abstract

This paper presents a methodology for Mining Association Rules from Code (MARC), aiming at capturing program structure, facilitating system understanding and supporting software management. MARC groups program entities (paragraphs or statements) based on similarities, such as variable use, data types and procedure calls. It comprises three stages: code parsing/analysis, association rule mining and rule grouping. Code is parsed to populate a database with records and respective attributes. Association rules are then extracted from this database and subsequently processed to abstract programs into groups containing interrelated entities. Entities are then grouped together if their attributes participate to common rules. This abstraction is performed at the program level or even the paragraph level, in contrast to other approaches, that work at the system level. Groups can then be visualised as collections of interrelated entities. The methodology was evaluated using real life COBOL programs. Results showed that the methodology facilitates program comprehension by using source code only, where domain knowledge and documentation are either unavailable or unreliable.

Keywords: Software Management; Software Quality; Program Comprehension; Software Analytics; Data Mining; Association Rules.

1. Introduction

This work introduces a methodology for mining software programs that produces a relatively small number of rules that are easy to manage and understand. It aims at facilitating *program comprehension* during legacy COBOL software management, by producing groups of program entities such as paragraphs, according to their similarities, such as the use of variables, data types and procedure calls.

The novelty of this work is two-fold. Firstly, it uses association rule mining from COBOL source code at the paragraph level, aiming at capturing program structure, where previous association rule mining approaches operated either at a higher level of abstraction and/or on different programming languages and different purposes. Secondly, MARC incorporates the novel idea of grouping entities together based on the strength of association rules connecting their items. In other words, although it uses association rule mining, the final product is a set of clusters.

Although there are many contemporary programming languages to choose from, still many business-critical systems are written in older programming languages such as COBOL. COBOL is predominantly used as “back office” in financial systems to process millions of batch transactions per day, which is crucial to any financial institution. Practitioners perceive legacy systems to be crucial as they are business critical, reliable, and have been running for decades; they have been well tested and practically run without errors to execute business processes [1].

A recent article by Reuters reported that roughly 95% of ATM swipes use COBOL code, a language still powering 80% of in-person transactions. Reuters calculates that there are still 220 billion lines of COBOL code currently used in production, 43% of banking systems are built on COBOL and that every day, COBOL systems handle $3 trillion in commerce [2]. Even when these systems are reaching the end of their useful lives, users are

reluctant to opt for migration to new technologies, given the risk of system change and the possibility that critical business rules got hard-wired within source code over time, without properly being documented.

Software maintenance is accepted to be the most difficult stage in software lifecycle [3], often performed with limited understanding of the design and the overall structure of a system, because of commercial pressures and time limitations. Fast, unplanned modifications based on partial understanding of a system result in increasing complexity and deteriorating modularity [4]. 50%-90% of the maintainers' time was reported to be spent on *program comprehension* particularly when older programming languages are used [5].

Many types of tools are available to help with program comprehension, emphasising different aspects of systems and modules, and usually creating new representations for them [6], [7]. Some tools perform deductive or algorithmic analysis of program properties or structure, e.g. program slicers [8]. Creating a decomposition of a program into a set of subsystems and grouping them according to their interrelationships is of great significance for any maintenance attempt [9], [10], [11].

The remainder of this paper presents data mining methods used for program comprehension, along with key background concepts first; then it describes the proposed methodology, including the data model and mining algorithm, as well as evaluation methods, assumptions and limitations. Section 4 briefly presents the tool implementing MARC; section 5 evaluates results for a case study and section 6 discusses these experimental results and assesses threats to validity. Section 7 concludes the paper with directions for further work.

2. Background

Software systems evolve during their lifecycle to conform to increasing user needs and the ever-changing legal, technological, and business environment. Attaining and retaining high levels of software quality requires continuous maintenance, comprising updates, enhancements and upgrades. Program comprehension is crucial during maintenance, especially in cases where documentation is poor or outdated and program structure is complex. For instance, Brooks [12], Soloway & Ehrlich [13], and Letovsky [14] detail the mental processes behind program comprehension.

Littman et al. put forward the widely applicable to software maintenance theory that a maintainer must accomplish static and causal knowledge, in order to successfully modify a program [15]. *Static knowledge* is concerned with the subsystem structure formed by program components, whilst causal knowledge is concerned with interactions and data flows between program components. This view of grouping program components into subsystems that provide a common service to the overall program is widely accepted as a key stage in program comprehension. Lakhotia [16] argued that such a software system abstraction is very important in maintenance, because it helps maintainers to infer interactions between subsystems. Lakhotia claimed that this knowledge helps the maintainer to understand the full impact of modifications to the source code.

Kunz and Black [17] argued that grouping program components into subsystems reduces the perceived complexity. They claimed that such a view helps maintainers predict the full impact of source code modifications. Tzerpos and Holt [18] conjecture that deriving a decomposition of software systems into a set of meaningful subsystems alleviates much of the effort required to understand a software system.

Data mining techniques can assist program comprehension and maintenance by producing structural views of legacy systems when source code is the only reliable information source. Data mining techniques can discover non-trivial and previously unknown relationships among elements in large databases [19]. This observation highlights the ability of data mining to discover useful knowledge about the design of large legacy systems. According to De Oca and Carver [9], data mining has three features that make it a valuable tool for program comprehension and maintenance tasks. First, data mining can be applied to large volumes of data. This characteristic implies that data mining is capable of analysing large legacy systems with complex structure. Secondly, data mining can expose previously unknown non-trivial patterns and associations between items in databases. Therefore, it can be utilised to reveal hidden relationships between system components. Finally, data mining techniques can extract information regardless of any previous knowledge of the object of study. This feature suits the nature of software maintenance when knowledge about system functionality and implementation details is poor, especially in large undocumented systems.

Data mining can address the problem of program comprehension in an effective way [20]. More specifically, it can provide system decomposition into several cohesive subsystems. This decomposition may be then utilised by software maintainers to speed up the maintenance process. An early industrial survey highlighted the need for automated methods deriving high level abstractions and module interrelationships, in order to accelerate and enhance program comprehension [4]. Various data mining approaches have been proposed. Two early ones are: *Clustering over a module dependency graph* [10] and *Identification of Subsystems based on Associations* (ISA methodology) [9]. Both provide a system abstraction up to the program level.

Mitchell and Mancoridis [10] proposed an automatic method, independent of programming language, for the creation of a hierarchical view of system architecture into subsystems, based on the components and the relationships between components that can be detected in source code. Their method commences with the extraction of a Module Dependency Graph (MDG) from the source code of the system. This graph contains nodes and edges that connect the nodes. Each node represents a module of the system (e.g. a program or a file) and each edge symbolises a dependency between two modules. At the next stage, the graph is partitioned using some special clustering algorithms to produce a high-level structure view of the system, based on the low-level components and their relationships. Based on the previous approach, Mitchell, and Mancoridis [10] developed a clustering tool, called "*Bunch*", which, unlike other software clustering tools, uses search techniques to perform clustering and automatically analyses a software system into a set of clusters containing high-level components (programs or files) of the system. In this respect this method is different from MARC in that clusters contain C modules (programs or files) based on calls to other modules rather than COBOL procedures containing the same variable names and procedure calls. Bunch has the advantage of offering alternative clustering algorithms such as a hill-climbing, an exhaustive clustering, and a genetic algorithm.

The ISA methodology, introduced by De Oca and Carver [9], produces a decomposition of a system into data cohesive subsystems. The authors consider a system as a collection of programs and subsequently a data cohesive subsystem consists of programs that use the same persistent data files. The extraction of these subsystems is based on the *Apriori* algorithm for mining association rules [21]. The ISA methodology consists of three major steps:

a) *Create the database view of the system*. ISA constructs a database view of the examined system that includes a set of tuples in an ASCII file. Each tuple corresponds to a different data file and contains all the programs that use this file.

b) *Perform data mining*. The previous database is mined by applying the Apriori algorithm up to the stage of creation of large itemsets. An itemset consists of several programs that use the same data files. In order to be large, the number of shared files must be equal or greater a user specified *support s*. Each large itemset corresponds to an association.

c) *Consolidate and interpret results*. The produced associations are used to build a group two-dimensional table of programs and files. The group table is constructed in such a way that programs that use the same files are represented by adjacent rows and files that are used by the same programs are represented by adjacent columns.

The ISA methodology has been tested in several COBOL systems that incorporate several programs and data files. The group table that is produced in each test gives a visual representation of the possible data cohesive subsystems inside the COBOL system. The authors applied, as a case study, ISA to TRS, a small system of 25 KLOC distributed in 28 COBOL programs, using 36 data files. The authors do not specify how they selected the required value of minimum support, which was 3 nor do they discuss minimum confidence selection. used their experience to verify that ISA can identify data cohesive subsystems. ISA is different from MARC in that groups contain programs that use the same persistent data files rather than COBOL procedures containing the same variable names and procedure calls.

Kanellopoulos et al. [20] applied K-means clustering [19] to C# source code, producing system overviews and deductions, which support further employment of an improved version of MMS Apriori association rule mining [22] that identifies hidden relationships between classes, methods and member data. It should be noted that MMS Apriori is not applied directly to code but to the output of the clustering step, i.e. clusters and related information such as Parameter Name and ID, Method ID, MIS, Cluster, Record Score and Relevance. This method is different from MARC in that it clusters C# classes that use methods and member data rather than COBOL procedures containing the same variable names and procedure calls.

Sartipi et al. [11] used Apriori association rule mining [21] for architectural design recovery. They proposed a model for the evaluation of the architectural design of a system based on associations among system components and used system modularity measurement as an indication of design quality and its decomposition into subsystems. Three association views of a system were generated: i) control passing, which represents system components correlation based on function invocation, ii) data exchange, which represents system components correlation based on aggregate data types and iii) data sharing, which represents system components correlation based on functions sharing global variables. This approach models software systems as attributed relational graphs with system entities as nodes and data-control-dependencies as edges. Application of association rules mining decomposes such graphs into domains of entities based on the association property. The itemsets used comprises global variables, data types and function calls whilst transactions comprise global variables, data types accessed, and functions called. This approach is based on the concept of the association between the components of a system. The proposed method uses partial matches at the level of module (a collection of functions, data types, and variables) and was evaluated on 2 C programs: i) CLIPS, a medium system (40 KLOC), with 734 functions, 59 data-types, and 163 global variables and

ii) TwentyOne, a small program (1.6 KLOC) consisting of 3 files, 38 functions, and 16 global variables. The authors do not specify how they selected the required value of minimum support, which was 2 nor do they discuss minimum confidence selection. Their method differs from MARC with respect to the language, the purpose of association rule mining as well as the level of granularity.

Other more recent approaches use a variety of data mining methods. For instance, Maqbool et al. [23] extracted association rules to capture associations between functions, global variables and user defined types from C code. They used metarule-guided association rule mining to find associations between items, which can be used to identify potential problems in a system. They used low and high thresholds of coverage, support and confidence. They aimed at understanding programs and indicating problem areas where improvements may be made. By relating their mined association rules to re-engineering patterns, they found solutions to commonly occurring problems. Their method looks for associations not only between functions, but also between functions and other items such as global variables and user defined types. In this sense their work relates association rule mining to re-engineering patterns and operates on C code. Sobernig, and Zdun, addressed the problem of the ad hoc and informal process of distilling architectural design decisions and their relationships using frequent itemsets mining [24]. Dave et al. applied association rule mining on project version history to find files that frequently change together [25].

Classification was used as a means for static code analysis of C for the timely identification of software bugs as well as for locating software defects [26], [27]. Software clustering approaches cluster large software systems based on the static or even dynamic dependencies between software artifacts [28]. Also, we recently combined clustering Java classes, followed by classification of extracted clusters, to assess internal software quality, using Java classes as entities and static metrics as attributes [29]. Finally, software metrics were extracted from C# code and clustered to assess maintainability [30].

Next, we present key background concepts pertaining Association Rule Mining, the backbone of MARC.

2.1. Association Rule Mining

Association rule, or frequent pattern mining is a data mining method for discovering relations between variables in large databases [19]. Given a set of transactions, where each transaction is a set of items, an association rule is an expression $X \rightarrow Y$, where X and Y are sets of items [21]. The actual meaning of this rule is that transactions in the database that contain the items in X also tend to contain the items in Y. Several algorithms have been proposed for the extraction of association rules from large databases [31], [32]. Most of them deal with sales data from transactional databases and provide rules that can be used for marketing purposes.

The aim is to identify strong rules using measures of interest, like *confidence* and *support*. *Frequent pattern* is a pattern (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set. There are exhaustive and heuristic association rule algorithms, like Apriori [21], a prominent algorithm for mining frequent itemsets for Boolean association rules or ARMICA [32].

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n binary attributes called items. Let $D = \{t_1, t_2, \ldots, t_m\}$ be a set of transactions called the dataset. Each transaction in D has a unique transaction ID and contains a subset of the items in I. A *rule* is

defined as an implication of the form X ⇒ Y where X, Y ⊆ I and X ∩ Y = ∅. The sets of items (for short *itemsets*) X and Y are called antecedent (left-hand-side or LHS) and consequent (right-hand-side or RHS) of the rule.

Apriori is simple, yet efficient algorithm, used extensively in the literature [21], [31]. At first, performs a database scan to find large 1-itemsets. These itemsets are actually all initial database items with support above the minimum specified. The subsequent passes consist of two tasks. First, the large itemsets of the previous pass are used to generate candidate itemsets for the current pass. For example, during pass *k*, the set of candidate itemsets C_k is created using the set L_{k-1} of large itemsets, generated during *k-1* pass. Set C_k is produced in two steps. First, L_{k-1} is joined with L_{k-1} so as to create all possible *k*-itemsets. A *k*-itemset has the form $\{item_1^p, item_2^p, ..., item_{k-1}^p, item_{k-1}^q\}$ where *p* and *q* are the joined (*k-1*)-itemsets under the join condition $item_1^p = item_1^q$, $item_2^p = item_2^q, \ldots, item_{k-2}^p = item_{k-2}^q$, $item_{k-1}^p < item_{k-2}^q$ ($item_i^p$ represents the *i*-th item of itemset *p*, i = 1, 2, …, k-1). Then, in the pruning step, all itemsets $c \in C_k$, which have at least one (*k-1*)-subset not in L_{k-1}, are deleted.

Following the generation of C_k, another database scan is performed to compute the support for each itemset $c \in C_k$. For this purpose, each itemset has a corresponding counter for storing its occurrences in the database records. First, for each record, all possible k-itemsets are created. Then, their existence in C_k is checked. If an itemset is member of C_k, its counter is increased. After the scan, the support for each itemset in C_k is calculated using the counter values. Itemsets with support below the minimum value are deleted and the remaining itemsets are kept in memory to be used during the next pass. The phase terminates when the set of large itemsets becomes empty.

To select interesting rules from the set of all possible rules, constraints on various measures of significance and interest can be used. The best-known constraints are minimum thresholds on support and confidence. The *support* supp(X) of an itemset X is defined as the proportion of transactions in the data set which contain the itemset. *Confidence* can be interpreted as an estimate of P(Y|X), the probability of finding the RHS of the rule in transactions under the condition that these transactions also satisfy the LHS, or the measure that indicates how often the rule is true. The confidence of a rule is defined as:

$$conf(X \Rightarrow Y) = supp(X \cup Y)/supp(X) \qquad (1).$$

These concepts are used in the presentation of our methodology, in the following section.

3. Methodology

Mining Association Rules from Code (*MARC*), the methodology proposed in this paper, analyses source code and produces useful rules derived from program entities, facilitating program structure comprehension [34]. The methodology aims at producing a system abstraction at a level lower than previous approaches.

A rule mining engine utilising the widely used *Apriori* algorithm [21] was built to extract association rules from source code elements. Programs can then be decomposed into groups containing entities which participate in common rules. The proposed methodology considers two key issues: modelling the input data and selecting an

appropriate algorithm to be tailored and applied to such data. Both issues are discussed and detailed in the following subsections, along with evaluation methods, assumptions and limitations.

3.1. Data Model

In order to exploit association rule mining algorithms, it is necessary to shift from market basket analysis, where most existing algorithms apply, to source code analysis. The data model to be used should comply with the model already used in market basket analysis to utilise existing algorithms. The <transaction, item> model applies to market basket analysis. Each transaction consists of several items purchased. All transactions are stored in a database table where each item is represented by a column, like the one in Table 1. For each transaction, a record is inserted into the table. If an item is contained in a transaction then the value of the record in the corresponding column is "1", otherwise *null*. Therefore, there is a relation of existence/absence between the transaction and the market items. A unitary value indicates presence and a null value indicates absence.

In the domain of source code analysis, we can use blocks of code, such as paragraphs or statements, instead of transactions, corresponding to the model's entities. These blocks may contain low-level code elements such as variable usage, data types and calls to other blocks. These elements constitute attributes. Thus, an <code entity, attributes for this entity> model is created.

Attributes should contain as much information as possible about the corresponding entity. Information about the presence or absence of an attribute in the block entity, similarly to the transactional model, is not enough since the presence of an attribute may have different meanings. For example, an occurrence of a variable may have different interpretations according to its usage: a variable may be defined, it can be used to store data, to retrieve data or it may be used as a parameter in a function call. All these usages represent additional information about the presence of a variable in the source code. The data model should capture this kind of information. A different attribute must be defined to represent each different usage. Thus, the previous model is transformed to a <block entity, usage of attribute> model.

The proposed data model is not only applicable to COBOL, but also to other programming languages. However, the focus here is COBOL, as motivated in the introduction, so the specific data model used, includes paragraphs (or statements) as entities and variables, data types and procedure calls as attributes. It should be noted that a COBOL program consists of divisions. The PROCEDURE DIVISION is composed of paragraphs, such as procedures. A paragraph is either user-defined or a predefined name followed by a period and consists of zero or more sentences/entries. Sentences are the combination of one or more COBOL statement that performs some processing. In practice, we parse a COBOL file to establish the list of procedures in the PROCEDURE DIVISION along with variables in the WORKING-STORAGE SECTION of the DATA DIVISION and procedure calls inside procedures.

Pre-processing COBOL code produces one database table per program file. Each table has rows corresponding to paragraphs and columns corresponding to variable names and procedure calls (with the prefix pr_). An example of such a table is Table 1, where 1s and 0s respectively signify the presence or absence of the relevant variable or procedure call in the corresponding paragraph. The table was constructed by inserting one line per procedure, and

one column per variable or procedure call, and populated by parsing the PROCEDURE DIVISION to establish the presence or absence of these variables and calls in each procedure. Table 1 corresponds to the code extract from the program 'Register', shown in Fig. 1 and detailed in section 5. For instance, procedures screen-up and screen-down use variables reg-screen-no, ring-bell and procedure calls pr_show-reg-screen and pr_set-cur-pos.

Table 1: Example of data extracted from the COBOL program 'Register'

Variable/ Proc. call / Paragraph	*reg-screen-no*	*ring-bell*	*classify-status-table*	*ptr*	*classify-status*	*function-key-letter*	*dummy*	*set-new-classification*	*pr_show-reg-screen*	*pr_set-cur-pos*	…
do-letter	1	1	0	0	0	0	0	1	1	0	…
set-classify-status	0	0	1	1	1	0	0	0	0	0	…
set-new-classification	0	0	1	0	0	1	1	0	0	0	…
screen-up	1	1	0	0	0	0	0	0	1	1	…
screen-down	1	1	0	0	0	0	0	0	1	1	…
show-reg-screen	…	…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…	…

```
    do-letter.
      IF reg-screen-no = 4
        PERFORM set-new-classification
        PERFORM show-reg-screen
      ELSE
        CALL ring-bell
      END-IF.
*****************************************************
    set-classify-status.
      MOVE ALL "NO " TO classify-status-table
      MOVE 1 TO ptr
      PERFORM 10 TIMES
        IF occupancy-index-set (ptr)
          MOVE "YES" TO classify-status (ptr)
        END-IF
        ADD 1 TO ptr
      END-PERFORM.
*****************************************************
    set-new-classification.
      MOVE function-key-letter TO dummy
      CALL "setclass" USING system-parameter-block,
                            master-record,
                            classify-status-table,
                            dummy
      CANCEL "setclass".
*****************************************************
    screen-up.
      IF reg-screen-no > 1
        SUBTRACT 1 FROM reg-screen-no
        CALL "clearx"
        CANCEL "clearx"
        PERFORM show-reg-screen
        PERFORM set-cur-pos
```

```
        ELSE
          CALL ring-bell
        END-IF.
*****************************************************
    screen-down.
        IF reg-screen-no < 5
          ADD 1 TO reg-screen-no
          CALL "clearx"
          CANCEL "clearx"
          PERFORM show-reg-screen
          PERFORM set-cur-pos
        ELSE
          CALL ring-bell
        END-IF.
```

Figure 1. Code extract from program 'Register'

Relationships among variables, as well as among paragraphs, can be found in such a table by analyzing variable co-occurrences. Several variables, co-occurring in the same group of paragraphs, can indicate that these paragraphs are related and may demonstrate associated functionality in the form of functional, communicational or logical cohesion [3]. For instance, as stated above, procedures screen-up and screen-down use variables reg-screen-no, ring-bell and procedure calls pr_show-reg-screen and pr_set-cur-pos, so they may be related.

Attributes should be qualitative to allow for application of existing mining algorithms without major changes in their reasoning. Thus, similarly to market basket analysis, the attributes are Boolean. Each block of code is represented by a table row and each attribute by a column. The value of the attribute is "1" if the corresponding code element is used in the block of code the record refers to. Otherwise, the attribute has a null value, as shown in Table 1.

The blocks of code should be defined in such a way that ensures that a high proportion of the source code is included for analysis. There are two possible solutions concerning the definition of entities that can be applied in procedural programming languages: *individual statements* or *modules* (paragraphs). Using statements as block entities enables analysing a program in its entirety. However, this analysis is performed at a highly detailed level and the results may be extremely complex. A decomposition of a program into several subsystems, where each subsystem contains related statements is difficult to understand and cannot offer important help in the comprehension of the general program structure.

Furthermore, many statements may not be considered in isolation, especially if they participate in a complex statement (e.g. *IF* and *EVALUATE* statements). Representing these statements as single entities will produce misinformation about the program. Modules apply to any programming language thus, using this model, the proposed method can be applicable to different cases, where programs written in different languages must be analysed. A module encloses a non-trivial amount of code that consists of several statements (variable declarations, assignments, calls to other modules, references etc.). As a result, it can be described by many attributes, larger than the one in the individual statement method. This facilitates the creation of groups of related modules, since it is much

easier to find important commonalities on the large range of attributes. The more the common attributes, the better the quality of the produced association rules and the consequent groups of modules.

Finally, program modules can be easily identified inside a program since, for every programming language, their start and end follow specific conventions. This facilitates automating data extraction from the source code, using the input model without user intervention. In summary, a data model with program modules as entities is the most suitable for the purposes of the proposed method. Nevertheless, the first method (single statement model) can always be considered as a complementary one, in cases that a more detailed decomposition of a program at the level of the statement is required.

3.2. Mining Algorithm

As indicated previously, the input data consist of block entities (paragraphs or even statements) having specific attributes that correspond to code elements inside blocks (variables, data types and procedure calls). Each entity is represented by a record in a database table, like Table 1. For each record, the value for columns that represent attributes of the entity is "1". The algorithm uses the table as input to extract association rules, that associate variables and procedure calls within paragraphs, and use these associations to produce groups of blocks (paragraphs). The blocks in each group should have common attributes participating to the same association rules. The number of common association rules indicates how strong the relationship between the blocks is.

For instance, extending the example we used in 3.1, procedures screen-up and screen-down use variables reg-screen-no, ring-bell and procedure calls pr_show-reg-screen and pr_set-cur-pos. That means that an itemset which occurs at least twice in the database is {reg-screen-no, ring-bell, pr_show-reg-screen, pr_set-cur-pos}. Various strong rules might be extracted from these itemsets. For example, if we set minimum support to 6% and minimum confidence to 100% then the rule ring-bell => pr_show-reg-screen emerges. This means that whenever the parameter ring-bell is used in a procedure in program 'Register', then a call to procedure show-reg-screen is always made, and that occurs in at least 6% (i.e. 3) of the program's procedures. Indeed, this is the case for procedures do-letter, screen-up and screen down, as it can be verified by examining Fig. 1.

The main aim of the algorithm is to help the user to gain a view of the program structure by analysing the program into these groups of blocks. The algorithm can be decomposed in three basic phases:

- *Large itemset identification*. Using the min. support value provided by the user, find all sets of items with support above this threshold.
- *Association rule generation*. Use large itemsets to generate association rules. Only rules with confidence above the user-specified minimum value must be produced..
- *Block group generation*. Find all block entities that contain common attributes participating in the same association rules. Create groups of blocks according to the number of common association rules.

Each of these phases is described separately in the following sub-sections, Along with details on the running example for program 'Register'.

3.2.1. Large itemset identification

The first two phases are based on *Apriori* [21], [31]. At first, a database scan is performed to find the large 1-itemsets. These itemsets are actually all initial database items with support above the minimum specified. This min. support can be established empirically following a trial and error process until one gets a "sensible" number of rules or alternatively using an algorithm that does not require this value to be specified by the user, like ARMICA [32]. The subsequent passes consist of the two tasks detailed in 2.1.

Following up the example we have been using from 3.1 onwards, given that 'Register' is a program with 53 procedures, potential values for min. support could be 4%, 6% or above, given that 2% would require variables to be present in only one procedure, giving no ground for grouping procedures. It turns out that values of support above 6% resulted in no frequent itemset, whilst min. support=6% produced two frequent 3-itemsets ({master-record, cur-pos, pr_show-reg-screen} and {master-record, last-master-record, pr_call-error}) and 3 frequent 2-itemsets ({reg-screen-no, pr_show-reg-screen}, {ring-bell, pr_show-reg-screen} and {master-record, cur-pos}). Setting min. support to 4% produces numerous more frequent itemset such as: ({master-record, last-master-record, pr_show-reg-screen}, {master-record, reg-screen-no, pr_show-reg-screen}, {master-record, input-room-number, input-address-lines}, {master-record, pr_call-error, pr_show-reg-screen}, {ring-bell, reg-screen-no, pr_show-reg-screen}, {recorded-flag, last-surname, last-reg-number}).

3.2.2. Association rule generation

The next phase involves the generation of association rules from large itemsets. Initially, a database table is created to store the rules, like the one shown in Tables 2 and 2a. For each itemset, all possible subsets are created and for each subset the corresponding association rule is constructed. If α is a subset of the large itemset *l*, then a rule of the form $a \Rightarrow l - a$ is created. Then, the confidence of the rule must be calculated. If the confidence is equal to or above the user-specified minimum, then the rule is stored in the database, otherwise, it is deleted. As soon as all itemsets have been processed, they are deleted from the main memory and the current phase terminates. For the previous example, if min. confidence is set to 100% then the rules extracted for min, support=6% are:

1. last-master-record => master-record
2. ring-bell => pr_show-reg-screen
3. last-master-record, pr_call-error =>master-record
4. cur-pos, pr_show-reg-screen => master-record

These rules are depicted in Table 2 below, where IDs correspond to the number on the previous list, and each rule is followed by a list of some of the procedures containing its items, as Table 2 only presents a snapshot of the full database table. For instance, 2 --> 119 stands for the rule ring-bell => pr_show-reg-screen and apparently variable ring-bell and procedure call pr_show-reg-screen appear in procedures do-letter and main-processing, along with others (like screen-down and screen-up) which cannot be shown. More rules are extracted if min. confidence is set to less than 100% or if min. support is less than 6%.

Table 2. Association rules from 'Register' (min. support=6%, min. confidence =100%)

Association rules		Blocks (Procedures)										
Rule ID	Rule	call-helplist	call-res-in	check-job-end	conditional-register-clear	do-letter	do-print-register	end-master-update	execute-register-clear	execute-save-register	main-processing	save-master-change
1	6 --> 1	0	0	1	1	0	1	1	1	0	1	1
2	2 --> 119	0	0	0	0	1	0	0	0	0	1	0
3	6, 104 --> 1	0	0	1	1	0	1	0	0	0	1	0
4	34,119 --> 1	1	1	0	0	0	0	0	0	1	1	0

3.2.3. Block group generation

The last phase of the algorithm involves the creation of groups containing block entities (procedures) with attributes (variables or procedure calls) that belong to the same association rules. First, a database table is created to store the produced groups, like the one shown in Tables 3 and 3a. Then, the input table is scanned to create groups of size 1. A new group is created for each block entity with attributes that participate at least in one association rule. After this initialisation, an iterative process begins. During the *k*-th iteration, groups of size *k* are generated, using groups of size k-1. For each group of size *k-1*, all blocks in the input table stored after the (*k-1*)-th block are identified. If such a block satisfies the same association rules with the current group, then a new group of size *k* is created containing the block. Keeping the order of blocks as stored in the input table, it is ensured that no duplicate groups are created (containing the same blocks). Each new group is added to the database table. As soon as all groups of size k-1 have been processed, a new iteration begins. The process terminates when the latest iteration did not produce any new groups.

For example, a rule of the form: "if SALARY exists in a paragraph P, then NAME exists in paragraph P" with confidence 90% and support 5%, implies that 90% of the paragraphs which contain SALARY also contain NAME, and 5% of the total number of paragraphs contain SALARY and NAME. In other words, the higher the support the more often the rule applies and the higher the confidence the more likely it is for the rule to be correct.

All such rules are produced, and entities are grouped together when they share a maximal number of common rules. Entities sharing many common rules, thus sharing many user defined words (variables and procedure calls), are deemed to be similar. Furthermore, rules can be examined to get a better understanding of collocating patterns which govern the use of user defined words. For instance, variables A, B, C may always appear as a group within a paragraph.

Following up our running example on 'Register', Table 3 depicts some of the 142 groups of procedures generated when min. support=6% and min. confidence =100%. For instance, the first group (Group ID=1) comprises procedures with IDs 8 and 15, that is call-helplist and call-res-in respectively, which contain only one common rule, that with ID=4, which is found to be cur-pos, pr_show-reg-screen => master-record, after looking up Table 2. In a similar fashion, the last group (ID=142) comprises procedures with IDs 17, 21, 24, 25, 26, 36, and 43, that is check-job-end, conditional-register-clear, do-print-register, end-master-update, execute-register-clear, main-processing and save-master-change respectively, which contain only one common rule, that with ID=1, which is found to be last-master-record => master-record.

Table 3. Groups from 'Register' (min. support=6%, min. confidence =100%)

Group ID	Procedure IDs	Common Rules IDs
1	8, 15	4
2	8, 27	4
…	….	…
135	17, 21, 24, 25, 26, 36	1
136	17, 21, 24, 25, 26, 43	1
137	17, 21, 24, 25, 36, 43	1
138	17, 21, 24, 26, 36, 43	1
139	17, 21, 25, 26, 36, 43	1
140	17, 24, 25, 26, 36, 43	1
141	21, 24, 25, 26, 36, 43	1
142	17, 21, 24, 25, 26, 36, 43	1

3.3. Evaluation Methods

Given that the end product of our methodology is a set of groups rather than mere association rules, we opt to evaluate results by means more suitable to cluster analysis instead of just support, confidence etc. used for association rules [33]. Clustering is an unsupervised learning method, meaning that a "ground truth is absent", but its evaluation can be supervised if there is expert opinion provided as a substitute to the ground truth [19].

We chose to evaluate final results, i.e. groups of procedures, by comparing them to a mental model that human experts formulate about the structure of a program. Precision and recall were introduced as a quantitative element in judging the accuracy of the results of the approach [9]. In this case, the *precision p* for a subsystem is the percentage of entities in the subsystem that belong to the subsystem according to the expert's mental model. Therefore, precision is high if a group (subsystem) contains only a few entities which belong to other subsystems. On the contrary, precision is low if a group contains many entities which belong to other subsystems. The *recall r* for a subsystem is the percentage of entities belonging to the subsystem's mental model which are present in the subsystem. Therefore, recall is high if the subsystem contains most of the entities suggested by the mental model; it is low if the subsystem contains only a few of the entities suggested by the mental model. Consequently, a subsystem of 'good quality' should present both high precision and high recall.

However, it is known that normally there is a trade-off between precision and recall, in other words if one tries to increase recall, precision deteriorates. In order to assess overall accuracy considering both the precision and the recall at the same time, we propose a simple indicator called the *performance P* of a grouping method, which is the sum of the products (precision) x (recall) for every subsystem. An alternative measure, often used in the literature, is the *balanced F-score*, which is the weighted harmonic mean of precision (*p*) and recall (*r*). The balanced F-score is a value from 0 to 1 inclusive, given by formula (2) [19]:

$$F = \frac{2 * p * r}{p + r} . \qquad (2)$$

The total balanced F-score F_t for a system is the sum of the F-scores of all constituent subsystems. However, it should be noted that neither P nor F_t considers the size of these subsystems. This implies that capturing accurately a small subsystem contributes equally to the value of P or F_t, as compared to capturing accurately a large subsystem. For example, correct identification of a subsystem which only contains one procedure contributes equally to correct identification of a subsystem which contains twenty procedures, which by comparison is not such a trivial task.

To account for the size of various subsystems let us introduce another accuracy measure the *Performance Index* (*PI*). *PI* is calculated by summing up the weighted products of precision and recall for each subsystem; weights depend on the number of entities that are present in each subsystem. *PI* for a clustering at a specific level, is given by formula (3):

$$PI = \sum_{k=1}^{m} \frac{p_k * r_k * e_k}{n} . \qquad (3)$$

where m is the number of subsystems, n is the total number of entities, and p_k, r_k and e_k are respectively the precision, recall and number of entities of each subsystem. *PI* is more informative compared to P or F_t as it uses the same principles, but penalises trivial classification success, in favour of more challenging tasks.

3.4. Assumptions and Limitations

MARC is a methodology designed to operate on COBOL systems, aiming at automating the extraction of groups of entities, such as statements or paragraphs, based on the use of attributes, such as user defined words (variables and procedure calls). It processes code within files but can also optionally include code imported from external files which are "copied" by use of the COPY statement. It performs static analysis, not dynamic analysis. It is an automated methodology, and as such, it does not require expert knowledge about the system under examination, nor does it use any domain specific information; it only uses the language's grammar and syntax.

The notion that attributes, such as variables declared at the Working-Storage section of program files, indicate information rich sections of a program is the basic input model related assumption for MARC. In other words, MARC explores grammatical and conceptual similarity based on attribute matches; structural or semantic similarity is outside the scope of this work. It is thus assumed that the overall purpose of a module (single statement or paragraph), can be indicated by examining the set of the data items it operates on. It follows that modules having similar subsets of data items are likely to have similar purpose; hence they may belong to the same conceptual subsystem.

4. A Tool for Code Analysis

A tool analysing source code was developed. It consists of three subsystems (as seen in Fig. 2), each of which performs a well-specified operation.

- *Database Management Subsystem*: responsible for the management of data either provided by the user (input data) or created during the mining process (tables of rules and groups). It handles database operations (insertion, deletion, record update, table creation and deletion and so on) and ensures the integrity of stored data. It also

determines the way of communication between the tool and the database. In practice we used MS Access as a DBMS, but any relational DBMS would work with the tool.

- *Processing Subsystem*: it is the heart of the system, responsible for executing the algorithm for association rules mining, as described in section 3. It receives input data from the Database Management Subsystem, performs the three processing phases (itemset creation, rule creation, block group generation) and sends results to the Database Management Subsystem for storage and the Input/Output Subsystem for display.
- *Input / Output Subsystem*: responsible for the communication with the end-user. Once the user specifies the program to be parsed, a database is populated with procedures for records and variables and procedure calls for attributes. Then the user can specify min. support and confidence, for rules and groups to be extracted, stored in the database and displayed. Display of results can be filtered according to the number of common ruldes and block size. The subsystem also provides information about processing progress.

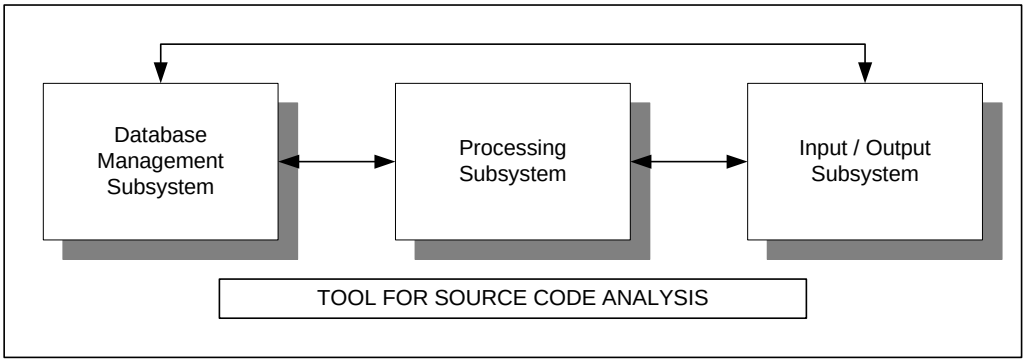

Figure 2. Code Analysis Tool Subsystems

As depicted in Fig. 2, all subsystems communicate with each other, in order to perform their tasks. Information such as min. support and confidence, itemsets, rules and block groups, is displayed during all processing phases. All processed data are stored in three database tables. Table *MinedTable* stores information extracted from source code. Each record contains a field with the name of the current block and one field for every item that can be found in the code blocks. The fields that represent items appearing in the current block have value 1, otherwise null.

All extracted association rules are stored in the *ResultTable*, like Table 2a. A record of this table contains the following fields: *Rule ID* (integer field where the identifier of the created rule is stored), *Rule* (text field that contains the rule itself in the following format 1, 4, 7 → 8, 13 where the IDs of the items that participate in the rule are separated by commas), *Confidence* (the confidence value of the rule), $Block_1$ (the first block in the *MinedTable*), $Block_2$ (the second block in the *MinedTable*), …, $Block_{last}$ (the last block in the *MinedTable*). The block fields that contain all items in the current rule have value 1, otherwise null (see Table 2a).

The *GroupTable* stores all generated block groups. A record of this table contains the following fields (see Table 3a): *Group ID* (the identifier of the group), *Block Members IDs* (the identifiers of the blocks that participate in the group), and *Common Rules IDs* (the identifiers of the common association rules).

Table 2a. Example of association rules extracted

Association rules			Blocks					
Rule ID	**Rule**	**Confidence %**	**Block1**	**Block2**	**Block3**	**Block4**	**Block5**	**Block6**
1	17 --> 21	100	0	1	1	1	0	0
2	21 --> 17	100	0	1	1	1	0	0
3	17 --> 27	66	0	1	1	0	0	0
4	17 --> 21, 27	66	0	1	1	0	0	0
5	36 --> 17, 21	100	0	1	1	0	0	0
6	27, 36 --> 17	100	0	1	1	0	0	0

Table 3a. Example of groups of blocks generated

Group ID	Block Members IDs	Common Rules IDs
1	2, 3	1, 2, 3, 4, 5, 6
2	2, 4	1, 2
3	3, 4	1, 2
4	2, 3, 4	1, 2

5. Result Evaluation

The methodology was evaluated using real programs, which were input to the tool. A COBOL proprietary software system that manages student accommodation was used for evaluation purposes. An expert, one of the original developers, responsible also for maintaining the system, guided the evaluation process, in terms of selecting suitable programs, proving mental models and assessing results.

The software system contains a large number of programs. Following advice from the expert, a small subset of the programs was selected for experimentation; these implement various important system functions, and their average size is approx. 1000 Lines of Code (LOC), the largest being 2288 LOC and the smallest 340 LOC. Results attained by applying *MARC* to one of the most significant of these programs called '*Register'* with 700 LOC, will be discussed here as a case study. Similar results were acquired when other programs were used. It must be noted that experiments at the statement level resulted in low level overviews of groups of conceptually "similar" statements. Such overviews were so detailed that, from a system comprehension perspective, were of as little use as the localised knowledge obtained by manually examining code. Using such low-level information in support of other maintenance tasks is possible, but beyond the scope of this work, and will be not analysed any further.

The actual data model used for '*Register'* involves procedures (paragraphs) as entities, and binary attributes depending on the presence or absence of variables and procedure calls in each procedure. The program consists of 53 procedures, 53 procedure calls and 72 variables. Minimum support and confidence were set to 4% and 100% respectively, to produce enough rules, adequate to allow for meaningful groupings. Setting min. confidence to 100% was deemed essential, as we wanted rules to always be true whilst setting min. support to 4% requires itemsets to be

present in at least 2 procedures, the bare minimum for a meaningful pattern. This produced 26 rules, which involve 16 of the procedures, two of the procedure calls and 10 of the variables.

Table 4 has similar format to that of tables 3 and 3a and displays a snapshot of the groups of procedures produced, one row per group. The table consists of 3 columns. The first column is the unique Group ID; the second column displays the IDs of all the procedures which are members of the group in question. The last column shows the IDs of the rules that are common to all procedures in the group. For example, the fourth row shows a group with ID 93; the group consists of 4 procedures whose IDs are 17 (check-job-end), 21 (conditional-register-clear), 24 (do-print-register), and 36 (main-processing) respectively. All four procedures have two rules in common, namely rules with IDs 1 (6 --> 1 or last-master-record=> master-record, which happens to be the same rule already presented in 3.3.2 and 3.3.3) and 11 (6, 104 --> 1 or last-master-record, pr_call-error => master-record, the same as rule 3 in 3.3.2). In order to make sense of what these groups and rules imply, one needs to check the database to see the corresponding procedures and variables/calls which participate to these rules. If on the other hand, one seeks to gain an overview of a system then these groups need to be visualised at high level of abstraction, hiding some of the details, as these can always be retrieved from the database.

Table 4. Groups of procedures produced

Group ID	Procedures IDs	Common Rules IDs
47	15, 19, 24	2, 3, 5, 6, 14, 15, 16, 17, 18
64	19, 27, 44	7, 8, 9, 10, 22, 23, 24, 25, 26
30	24, 36	1, 11, 12, 20
93	17, 21, 24, 36	1, 11
77	23, 45, 46	4, 21
81	24, 26, 36	1, 12
3	8, 15	13, 19
37	27, 36	19, 20
155	17, 21, 24, 25, 26, 36, 43	1
90	8, 15, 27, 36	19
121	23, 36, 45, 46	4
43	4, 8, 15	13
83	24, 27, 36	20

One can notice that each procedure may participate to more than one group, as it can have variables/procedure calls participating to more than one association rule. A flexible approach would be to allow this as a means of mapping the reality that some procedures could conceptually belong to more than one subsystem. However, in order to allow for comparison with results produced by other methods particularly involving clustering [35], finding a way of resolving ambiguity was deemed necessary.

Consequently, two ways of visualising and consolidating results are presented here: one allowing for loose coupling among procedures which are assigned to different subsystems, and another requiring each procedure to be assigned to one and only one subsystem. These alternatives are discussed in the following subsections.

5.1. Loosely Coupled Groups

The first version, which allows for loose coupling among procedures assigned to different subsystems, is graphically represented in Fig. 3. This is one way of visualising the results depicted in Table 4. Each procedure is represented by a box containing its ID. Edges connect procedures with common rules. The number of common rules is shown at the intersection of the edges, unless there is just one edge in which case the number is displayed in the middle of the edge. Groups of procedures form "*collections*" indicated by square polygons with bold lines and named after capital letters, i.e. **A**, **B**, **C**, **D** etc. For instance, group **A**, comprises three procedures (19,27, 44) corresponding to group 64 in Table 4.

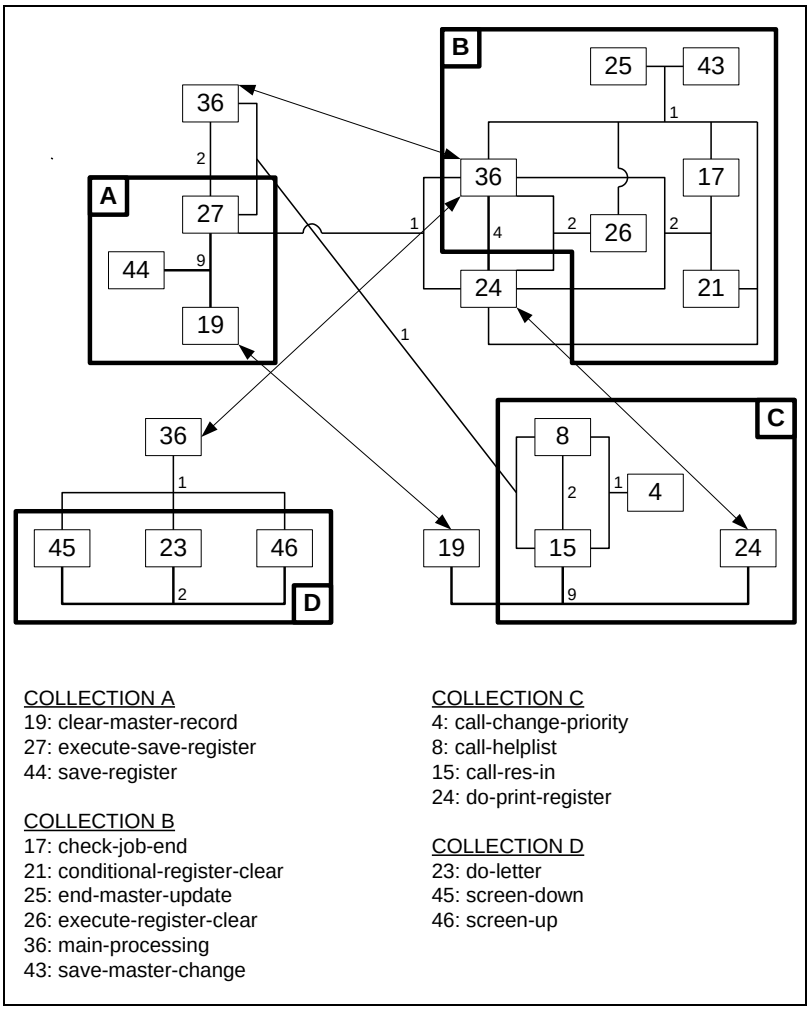

Figure 3. Loosely Coupled Groups Produced

As depicted in Fig. 3, some procedures (like 19, 24 and 36) participate to more than one group, given that they share common rules with more than one subset of procedures. When a procedure partakes to k>1 groups it is depicted by k boxes, linked by double edge arrows. When placing procedures in collections, two criteria can be used: a) groups with more common rules get priority over groups with less common rules and b) the number of common

rules being equal, then the number of procedures the current procedure is connected to, determines the collection it belongs to. If these criteria fail to determine which collection a procedure belongs to, then a larger set of association rules can be produced for disambiguation, by lowering the level of confidence or support.

Results were evaluated by referring to the source code of the program. For example, collection **D** in Fig. 3 consists of procedures do-letter, screen-down and screen-up, with IDs 23, 45 and 46 respectively. These procedures have two common rules with IDs 4 and 21 which were cross-referenced to correspond to the following rules: (2 → 119, or ring-bell => pr_show-reg-screen) and (2, 17 → 119 or ring-bell, reg-screen-no => pr_show-reg-screen), respectively. These rules contain three items with IDs 2, 17 and 119 which were found to correspond to the following variables/procedure calls respectively: ring-bell, reg-screen-no and pr_show-reg-screen.

As shown in Fig. 1, all three procedures in collection **D** use these three items (shown in bold) and actually perform similar operations. Thus, collection **D** indicates a functionally coherent subsystem of the initial program. All the produced collections were similarly checked against the source code, which confirmed that they indeed contain related procedures.

Our expert, one of the original developers of '*Register*', the program under examination, who is also responsible for its maintenance, produced a mental model for it, by distributing its procedures into subsystems according to their functionality. The full model consists of ten subsystems. Table 5 shows a subset of the mental model with 4 of these subsystems; these contain all 16 procedures that are examined in this case study.

Table 5: Mental Model for '*Register'*

Subsystem	Procedure ID	Procedure
help system	8	call-helplist
Interface	45	screen-down
	46	screen-up
Control	17	check-job-end
	21	conditional-register-clear
	36	main-processing
Students	4	call-change-priority
	15	call-res-in
	19	clear-master-record
	23	do-letter
	24	do-print-register
	25	end-master-update
	26	execute-register-clear
	27	execute-save-register
	43	save-master-change
	44	save-register

In the baseline case, with all 16 entities grouped together, resulting in 100% recall, the precision for each subsystem is respectively 6%, 13%, 19% and 63%. The baseline performance thus is P=1, F_t=0.356 and PI=0.445.

Comparing the model to the produced program decomposition in Fig. 4, the following results were obtained:

- Collection **A**: all 3 procedures belong to the *students* subsystem, resulting in 100% precision, 30% recall, P=0.30 and F=0.462.

- Collection **B**: 3 of the 6 procedures belong to the *control* subsystem, and the remaining to the *students* subsystem. So, with regards to the *control* subsystem precision is 50%, recall 100%, *P*=0.50 and *F*=0.667.
- Collection **C**: one procedure belongs to the *help* subsystem and three to the *students* subsystem, resulting in 25% precision, 100% recall, *P*=0.25 and *F*=0.400, for the *help* subsystem.
- Collection **D**: two procedures belong to the *interface* subsystem and one to the *students* subsystem; this results in 66% precision, 100% recall, *P*=0.67 and *F*=0.800, for the *interface* subsystem.

Summarising, 72.75% of the produced abstraction is consistent to the mental model, a relatively good result since the mental model does not capture the interconnections between procedures from different subsystems, although these connections play an important role in the system operation. One can observe that the highest precision was achieved for collection **A** which has procedures with the maximum number of common rules (9). All in all, for the decomposition in Fig. 1, is *P*=1.72, F_t=0.582 and *PI*=0.380. This is 1.72 and 1.63 times better than the baseline performance in terms of *P* and F_t respectively, but also 1.17 times worse in terms of performance *P*.

5.2. Decoupled Groups

An alternative way of visualising and consolidating the results depicted in Table 4 does not allow for loose coupling among procedures, which are assigned to different subsystems, as the one discussed earlier. On the contrary, it requires each procedure to be assigned to one and only one subsystem.

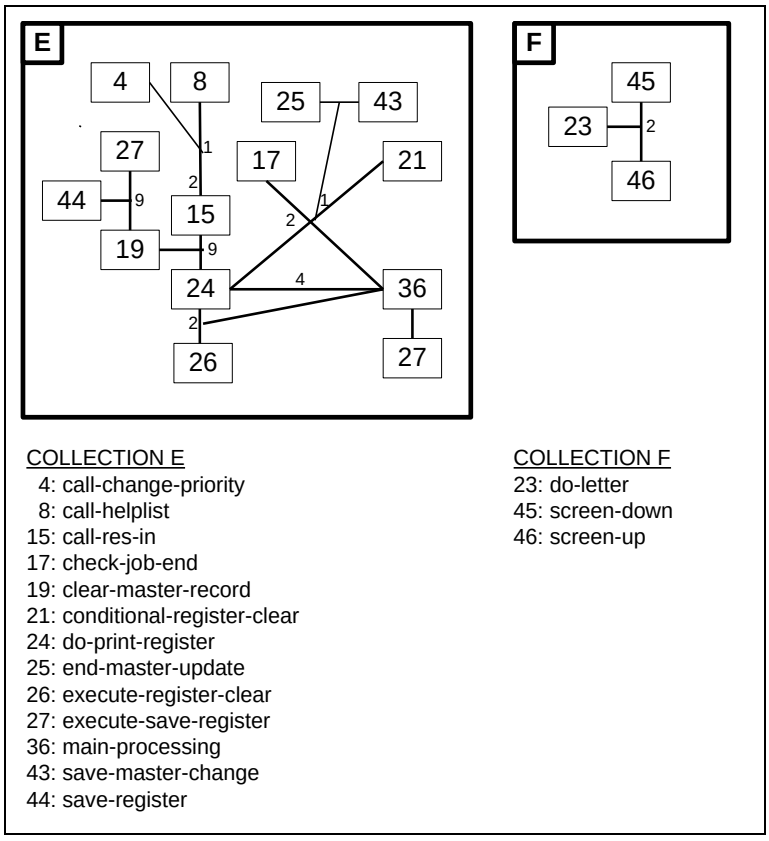

Figure 4: Decoupled groups produced

Applying this principle leads to the groupings depicted in Fig. 4. Here there are only two collections, **E** and **F**. Collection **F** actually represents group 77 as seen in Table 4; it consists of procedures 23, 45 and 46 which have 2 rules in common (with IDs: 4 and 21). By cross-referencing the IDs with the database, it can be shown that these IDs correspond to procedures do-letter, screen-down and screen-up respectively, as detailed in 5.1.

Comparing the mental model to the produced program decomposition in Fig. 4, the following results can be obtained:

- Collection **E**: 9 out of 13 procedures belong to the *students* subsystem, 3 procedures belong to the *control* subsystem and one procedure belongs to the *help* subsystem. This results in 69% precision, 90% recall, P=0.62 and F=0.783, for the *students* subsystem; 23% precision, 100% recall, P=0.23 and F=0.375, for the *control* subsystem and finally 8% precision, 100% recall, P=0.08 and F=0.143, for the *help* subsystem.
- Collection **F**: two procedures belong to the *interface* subsystem and one to the *students* subsystem; this results in 67% precision, 100% recall, P=0.67 and F=0.800, for the *interface* subsystem.

All in all, for the decomposition in Fig. 4, is P=1.60, F_t=0.525 and PI=0.521. This is 1.60, 1.47 and 1.17 times better than the baseline performance in terms of P, F_t and PI respectively.

5.3. Result Comparison

The results achieved by Loosely Coupled Groups (LCG) and Decoupled Groups (DG), along with the baselines values for P, F_t and PI are summarised in Fig. 5. It can be observed here that LCG outperformed DG when applied to *'Register'*, except for PI.

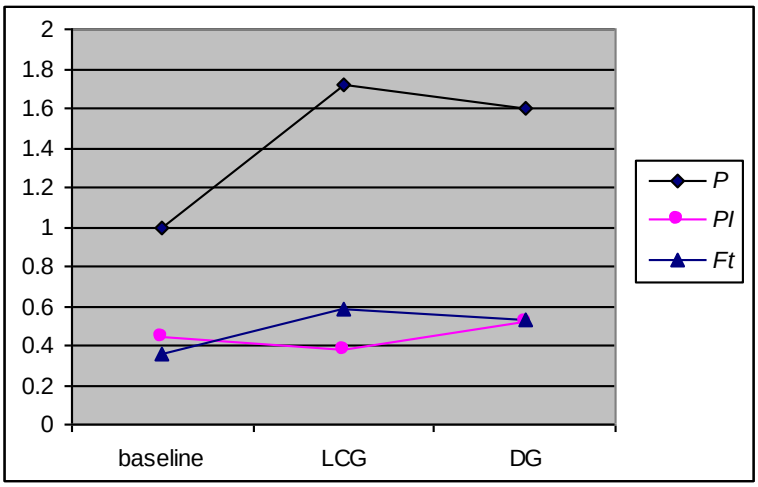

Figure 5: Summary results for *'Register'*

Table 6: Results on *'Register'* and its four subsystems

	Register		
	P	PI	F_t
Baseline	1.00	0.45	0.36
LCG	**1.72**	0.38	**0.58**
DG	1.60	**0.52**	0.53

(a)

	Help				Interface			
	p	*r*	*P*	*F*	*p*	*r*	*P*	*F*
Baseline	0.06	**1.00**	0.06	0.118	0.13	**1.00**	0.13	0.222
LCG	**0.25**	**1.00**	**0.25**	**0.400**	**0.67**	**1.00**	**0.67**	**0.800**
DG	0.08	**1.00**	0.08	0.143	**0.67**	**1.00**	**0.67**	**0.800**

(b)

	Control				Students			
	p	*r*	*P*	*F*	*p*	*r*	*P*	*F*
Baseline	0.19	**1.00**	0.19	0.316	0.63	**1.00**	**0.63**	0.769
LCG	**0.50**	**1.00**	**0.50**	**0.667**	**1.00**	0.30	0.30	0.462
DG	0.23	**1.00**	0.23	0.375	0.69	0.90	0.62	**0.783**

(c)

A more comprehensive summary of the results is given in Table 6, which details how the LCG and DG faired in comparison to the baseline performance. The table shows results for performance *P*, Performance Index *PI* and total balanced F-score F_t for the full system (Table 6a) or precision *p*, recall *r*, performance *P*, and balanced F-score *F* for the four subsystems (Table 6b). Numbers in bold indicate the highest value per column.

Results show that LCG achieves higher Performance *P* both at the system and the subsystem level (in all but the *student* subsystem, where DG is doing better). The same is largely true in terms of the balanced F-score *F*: LCG is the best both at the system and the subsystem level (in all but the *student* subsystem, where DG is again doing better). The only area that DG outperforms LCG globally is in terms of Performance Index *PI*. So, for *'Register'*, LCG was shown to perform better both locally and globally.

In this case study, all produced rules among the 16 procedures in hand were used. Alternatively, one could introduce thresholds by choosing to ignore groups with low number of common rules. In that case, different groupings could have been produced.

Overall, it can be claimed that experimenting with '*Register'*, not only indicated the applicability of the methodology, but also highlighted its strengths in capturing the logical modularity of the system, as it was up to 1.72 times, in terms of Performance *P*, up 1.63 times, in terms of total balanced F-score F_t, and up 1.17 times, in terms of Performance Index *PI*, more accurate than a "wild guess", which would place all the procedures in a single group.

6. Discussion

MARC was evaluated using parts of a large legacy COBOL system. Programs were analysed in a "line by line", and "paragraph by paragraph" fashion. The methodology groups paragraphs together if they contain a user-defined number of common, strong rules. The "line by line" mode though, was of very fine level of detail resulting in overviews which arguably could have been obtained by manually examining the code. The real strength of *MARC* was revealed when grouping paragraphs together.

Examples with COBOL programs of up to 2288 lines of code highlighted the potential of *MARC* in identifying groups of variables and/or procedure calls which tend to appear in the same paragraph, thus implying that they are interrelated. The accuracy of the results was evaluated by comparing the produced subsystem abstractions with

expert mental models. The abstractions were shown to be accurate, capturing subsystems consistently with the mental models. Pair wise values of precision and recall ranged between (25%, 100%) and (67%, 100%). The highest precision achieved was 100%, the highest recall 100% and the highest balanced F-score was 0.8. The lowest values for precision, recall and balanced F-score were respectively: 8%, 30/%, and 0.143. All in all, *MARC* performed worse than the baseline accuracy only once, when LCG failed to capture adequate proportion of the *student* subsystem.

An empirical evaluation revealed the ability of the methodology to produce meaningful results that can be utilised to generate a structural view of the examined program. The methodology can reveal important associations at two levels: a) between blocks of code and b) among source code components (variables, procedure calls) inside these blocks. Evaluation of the proposed methodology highlighted certain issues regarding the methodology and interpretation of results, which are discussed next.

Firstly, it was observed that database tables produced as input for *MARC* tended to be sparse. This can be explained by the fact that a program usually contains variables used locally for specific reasons, by specific procedures. Only the usage of some "global" variables is spread across several procedures. As a matter of fact, any procedure can only use a limited number of variables. Thus, records in the input table for each procedure have very few non-null fields, which correspond to variable usage. On the contrary, in transactional databases, each transaction may contain any item with the same probability as any user can potentially buy any product.

Based on the previous observation, it can be deducted that very small support values should be used in order to produce several large itemsets. By decreasing support, more items pass the support threshold, more large itemsets are created, and consequently, more information about block groups can be retrieved.

The groups produced may not contain all the procedures. Procedures which fail to participate in groups normally contain items with very low support, below the specified threshold. It is possible that these procedures perform specialised functions which normally do not appear in any group. However, if minimum support is not low enough, important information may be missed, and several 'interesting' procedures may not be present in the resulting groups. Thus, it is important to choose low support values to ensure that only 'isolated' procedures of limited interest are not present in the groups.

High confidence values produce strong rules between items. For example, a rule of the form <*varA*→*varB*> with 100% confidence implies that variable B is always present when variable A is present. Such rules reveal possible participation of variables in very specific tasks. These rules should be used to produce groups of interconnected procedures that deal with such specific tasks.

Some procedures possibly participate in more than one group. These procedures are likely to contain a larger number of variables than others. For example, as shown in Table 4, procedure 36 (main-processing) participates in eight out of a total of thirteen from the groups produced for '*Register'*. As expected, this procedure is the main procedure of the program that contains 32 items (variables and calls to other procedures), when all other procedures contain less than 10 items. Generally, such procedures can be considered as communication links between different groups.

Generated results need to be summarised and interpreted, to create meaningful collections of procedures. Membership in such collections is based on the resulting groups. Procedures that participate to a group having many common rules should undoubtedly belong to the same collection. However, the number of common rules is not the only criterion for collection membership; occurrence of common rules among groups may also indicate a relation between some procedures. For example, as shown in Table 4, procedures 4 (call-change-priority), 8 (call-helplist) and 15 (call-res-in) share only one common rule: one with ID 13 (1, 17 --> 119 or master-record, reg-screen-no => pr_show-reg-screen). This rule does not appear in any other group. Although it has a low occurrence among the groups, its presence indicates a relationship among the procedures that share it. For procedures that participate in several groups, two criteria were identified, to decide in which collection they should be placed: the number of common rules in each group and the number of related procedures.

The produced results were verified by referring to the source code of the programs. This method can be applied to small programs up to some thousand lines of code, but it is very difficult to be used in much larger programs. In such cases, one could rely on the program's mental model, which represents the program structure from the software engineer's point of view. However, this method is not always reliable, as different people may have different mental models about the same system, according to their mental processes.

In summary, *MARC* was shown to produce results that were accurate when compared to expert mental models, and meaningful in terms of reasonably capturing conceptual similarity. Consequently, *MARC* can be plausibly expected to assist software maintainers to understand the structure of unfamiliar systems. Its precision and recall is comparable the levels achieved in other works like [11], although there is literature not using a ground truth such as [9]. However, there are certain threats to validity discussed in 6.1 and issues, which could be further investigated in the future and are discussed in section 7.

In terms of utility, *MARC* produces systems' overviews, which can aid comprehension and reduce the perceived complexity thus facilitating maintenance. *MARC* forms collections of procedures, using similarity based on collocated sets of variables or procedure calls. Thus, it can easily be seen how this methodology can facilitate maintenance tasks, such as code modifications. For example, if a software maintainer wanted to change the use of variables within the body of a procedure it would be advisable to check whether other "similar" procedures are affected. This would be even more so, if the task involved changing a procedure: all other procedures which call it should be carefully examined. This can facilitate impact analysis and risk assessment.

MARC could also be used in perfective maintenance, when attempting to improve systems cohesion and coherence, in other words when the goal is to increase system modularity. This could happen by relocating procedures into programs where they "naturally" belong. Finally, *MARC* could also provide insights into programming styles, by exposing patterns related to the presence of variables and procedure calls in paragraphs. Interestingly, irregularities in the observed patterns may suggest that parts of a program may be "exceptional", thus requiring further testing or expert examination, interpretation and validation.

6.1. Threats to validity

The proposed methodology offers significant advantages, yet, further to the points made in section 3.4, there are certain issues that need to be carefully considered when applying it to a new context. Firstly, due to the nature of the data model the database produced tends to be sparse, thus producing a limited number of strong rules, which in turn could magnify the contribution of outlier values. This could be addressed mainly by manual inspection. This can be considered a small price to pay for the automation of the remaining process. Another issue is that of determining the "right" values for min. support and confidence. Again, this currently is dealt with empirically in a trial and error manner, but could be resolved using algorithms that automatically determine these thresholds [32].

Another threat underpinning the evaluation of the methodology is that of the need of an expert who provides a mental model of the system in order to use this as a ground truth. This irrevocably introduces bias and subjectivity which could partially be alleviated by engaging several experts and aggregating opinions. This, however, is a scarce resource, thus we are left with a question regarding whether a disparity between the expert model and the one produced by MARC is always down to error attributed to the methodology and not the other way around.

Finally, the system under examination is a reasonably sized one, comprising many small to medium programs, the most representative of which were selected for analysis by its chief programmer and lifelong curator. MARC coped well producing results in fractions of seconds for most cases. We did not test the methodology on any massive industrial scale system, so scalability is not proven, but no major challenges are expected regarding this.

7. Conclusions and Further Work

MARC was shown to produce accurate results when compared to expert mental models, reasonably capturing conceptual similarity. Consequently, it can be plausibly expected to facilitate understanding the structure of unfamiliar systems. The work presented here extends the state of the art in two dimensions. Firstly, it uses association rule mining from COBOL source code at the paragraph level, aiming at capturing program structure. Existing approaches, such as these proposed by De Oca and Carver [9], Kanellopoulos et al. [20], and Maqbool et al. [23], used association rules on programs written in various programming languages, at various levels of abstraction aiming at a variety of purposes.

De Oca and Carver [9] used Apriori association rule mining [21] to establish associations among COBOL programs and files. Their work addresses the issue of program comprehension at a level higher than MARC: that of files and programs. Kanellopoulos et al. [20] applied MMS Apriori association rule mining [22] to the output of a clustering step, i.e. clusters extracted from C# code at the class level, rather than directly to code, as MARC does. Maqbool et al. [23] used metarule-guided association rule mining from C code to gain program understanding and indicate problem areas for improvements to be made. They find associations not only between functions, but also between functions and items such as global variables and user defined types and relate them to re-engineering patterns. MARC mainly differs from this approach in terms of the purpose (capturing program structure vs. identification of problems) as well as its applicability to COBOL rather than C.

Secondly, MARC incorporates a novel idea in the field of software analytics, or code mining: that of grouping entities together based on the strength of association rules connecting their items. In other words, although it uses association rule mining, the final product is a set of clusters. This idea requires further exploration, possibly within the boundaries of mainstream data mining.

Finally, this paper goes beyond traditional evaluation metrics like precision, recall and F-score, and the debate surrounding these, by piloting two novel measures of accuracy, namely Performance (P) and Performance Index (PI). Again, more work needs to be done to validate these measures and assess their quality compared to the ones traditionally used in the literature [36], [37]. However this work indicates their potential, and as it can be seen in Fig. 5, where P and PI appear to behave in line with F_t. Ongoing work further validates this [38], but this is definitely an avenue worth exploring in the context of metrics.

As mentioned earlier, the suggested methodology for mining association rules from source code is new. The remainder of this section provides several possible further enhancements to the methodology, that should be performed in order to achieve a comprehensive solution to the problem of source code analysis. These improvements include:

- *Experimenting with other association rule mining algorithms*.

Apriori was selected as the basis for algorithm of the proposed methodology. However, it would be very interesting, from a research point of view, to use other existing algorithms for mining association rules that perform equally well or even better than Apriori, especially in cases of larger databases [22], [31], [33], [39]. Thus, it would be possible to measure and compare the performance of each algorithm in the domain of source code analysis. Using algorithms which automatically determine the required thresholds for min. support and confidence, such as [32], is deemed to be beneficial to the process as a whole.

- *Creation of a quantitative input model*.

In order to exploit existing mining algorithms for association rules extraction, the suggested data model was qualitative. A future enhancement could be the adoption of a quantitative data model. This model is richer as it can store information not only for the existence/absence of code items in blocks (as the present model does), but also for the number of item occurrences inside the blocks (e.g. itemA: 3 → itemB: [2 … 5] means: if itemA occurs 3 times then itemB occurs from 2 to 5 times). Furthermore, the usage of this model requires a new algorithm that can produce and process quantitative association rules.

- *Further evaluation on larger programs and/or involving end users*.

Experiments were executed on programs containing up to 2288 lines of code. In order to achieve a better empirical evaluation of the proposed methodology, experiments could be performed on larger programs. The barrier here is the availability of the original programmers or at least a knowledgeable group of maintainers who may be able to provide the “ground truth”, i.e. the mental models necessary for result evaluation. Alternatively, we could run a user-study/ survey to evaluate the tool, involving project managers and proposing the output of the tool with the aim of gathering their feedback.

- *Obtainment of not just insightful but also actionable information for maintenance tasks*

Our methodology produces overviews and interrelationships. This can be time saving when it comes to acquiring adequate comprehension prior to maintenance. However, it would be desirable if these insights were also truly actionable in the sense described by Zhang et al. [40].

8. References

1. R. Khadka, B.V. Batlajery, A. Saeidi, S. Jansen, J. Hage, "How do professionals perceive legacy systems and software modernization?", *Proc. 36th Int'l Conf. Software Engineering* (ICSE 14), 2014, pp. 36-47.
2. "COBOL blues", Reuters 2017. http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html Last access 30/11/18.
3. I. Sommerville, *Software Engineering*, 10h Ed., Harlow, Addison-Wesley, 2016.
4. C. Tjortjis and P.J. Layzell, "Expert Maintainers' Strategies and Needs when Understanding Software: A Qualitative Empirical Study", *Proc. IEEE 8th Asia-Pacific Software Engineering Conf.* (APSEC 2001), IEEE Comp. Soc. Press, 2001, pp. 281-287.
5. B. Eddy, "Structured source retrieval for improving software search during program comprehension tasks", *Proc. ACM SIGPLAN Conf. Systems, Programming, and Applications: Software for Humanity* (SPLASH '14), 2014, pp. 13-15.
6. G. Canfora, A. Cimitile, A. De Lucia and G.A. Di Lucca, "Decomposing legacy systems into objects: an eclectic approach", *Information and Software Technology*, Vol. 43, 2001, pp 401-412.
7. C. Tjortjis, N. Gold, P.J. Layzell and K. Bennett, "From System Comprehension to Program Comprehension", *Proc. IEEE 26th Int'l Computer Software Applications Conf.* (COMPSAC 02), IEEE Comp. Soc. Press, 2002, pp. 427-432.
8. J. Silva, "A vocabulary of program slicing-based techniques", *ACM Computing Surveys* (CSUR), Vol. 44, No. 3, June 2012, Article No. 12.
9. C.M. De Oca and D.L. Carver, "Identification of Data Cohesive Subsystems Using Data Mining Techniques", *Proc. Int'l Conf. Software Maintenance* (ICSM 98), IEEE Comp. Soc. Press, 1998, pp.16-23.
10. B.S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool", *IEEE Trans. Software Eng.*, Vol. 32, no. 3, 2006, pp. 193-208.
11. K. Sartipi, K. Kontogiannis and F. Mavaddat, "Architectural Design Recovery Using Data Mining Techniques", *Proc. 2nd European Working Conf. Software Maintenance Reengineering* (CSMR 2000), IEEE Comp. Soc. Press, 2000, pp. 129-140.
12. R. Brooks, "Towards a theory of the Comprehension of Computer Programs", *Int'l. Journal of Man-Machine Studies*, Vol. 18, no. 6, pp. 543-554, 1983.
13. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge", *IEEE Transactions Software Engineering*, Vol. 10, no. 5, pp. 595-609, 1984.
14. S. Letovsky, "Cognitive Processes in Program Comprehension", *1st Workshop Empirical Studies of Programmers*, pp 58-79, 1986.
15. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance", *1st Workshop Empirical Studies of Programmers*, pp. 80-98, 1986.
16. A. Lakhotia, "A Unified System for expressing Software Subsystem classification techniques", *Journal of Systems and Software*, Vol. 36, no 3, pp. 211-231, 1997.
17. T. Kunz and J. P. Black, "Using Automatic Process Clustering for Design Recovery and Distributed Debugging", *IEEE Transactions Software Engineering*, Vol. 21, no. 6, pp. 515-527, 1995.
18. V. Tzerpos and R. Holt, "Software Botryology: Automatic Clustering of Software Systems", *Proc.* IEEE *9th Int'l Workshop Database Expert Systems Applications* (DEXA98), pp. 811, 1998.
19. I. Witten, E. Frank, M. Hall and C. Pal, "*Data Mining: Practical Machine Learning Tools and Techniques*", 4th Ed., Morgan Kaufmann, 2016.
20. Y. Kanellopoulos, C. Makris and C. Tjortjis, "An Improved Methodology on Information Distillation by Mining Program Source Code", *Data & Knowledge Engineering*, Vol. 61, No. 2, 2007, pp. 359-383.
21. R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proc. 20th Int'l Conf. Very Large Data Bases* (VLDB 94), 1994, pp. 487-499.

22. I. N. Kouris, C. Makris, and A. Tsakalidis, "An Improved Algorithm for Mining Association Rules Using Multiple Support Values", *Proc. 16th Int'l Florida Artificial Intelligence Research Society Conf.*, (FLAIRS 03), 2003, pp. 309-314.
23. O. Maqbool, H.A. Babri, A. Karim, M. Sarwar, "Metarule-guided association rule mining for program understanding", *IEE Proc. Software*, Vol. 152, No. 6, 2005, pp. 281-296.
24. S. Sobernig, and U. Zdun, "Distilling Architectural Design Decisions and Their Relationships Using Frequent Item-Sets", *13th Working IEEE/IFIP Conf. on Software Architecture*, 2016, pp. 61-70.
25. N. Dave, K. Potts, V. Dinh, and H.U. Asuncion, "Combining Association Mining with Topic Modeling to Discover More File Relationships", *Int'l Journal on Advances in Software*, vol. 7 no. 3 & 4, pp. 539-550, 2014.
26. A.T. Misirli, A.B. Bener, B. Turhan, "An industrial case study of classifier ensembles for locating software defects", *Software Quality Journal*, Vol. 19, No. 3, 2011, pp. 515-536.
27. H. Tribus, I. Morrigl, S. Axelsson, "Using Data Mining for Static Code Analysis of C", *Proc. 8th Int'l Conf. Advanced Data Mining and Applications* (ADMA 2012), LNAI 7713, 2012, pp. 603-614.
28. M. Shtern, V. Tzerpos, "Methods for selecting and improving software clustering algorithms*". Software Practice and Experience*, Vol. 44, No. 1, 2014, pp. 33-46.
29. D. Papas and C. Tjortjis, "Combining Clustering and Classification for Software Quality Evaluation", *Lecture Notes Computer Science*, Springer-Verlag, Vol. 8445, 2014, pp. 273-286.
30. S. Arshad, C. Tjortjis, "Clustering Software Metric Values Extracted from C# Code for Maintainability Assessment", SETN 16, Article No. 24, *ACM Int'l Conf. Proc. Series*, 2016.
31. Z.H. Deng and S.L. Lv, "Fast mining frequent itemsets using Nodesets", *Expert Systems with Applications*, Vol. 41, No. 10, 2014, pp. 4505-4512.
32. 1.S. Yakhchi, S.M. Ghafari, C. Tjortjis, M. Fazeli, "ARMICA-Improved: A New Approach for Association Rule Mining", Proc. 10th Int'l Conf. on Knowledge Science, Engineering and Management (KSEM 17), *Lecture Notes in Artificial Indigence*, Springer-Verlag, vol. 10412, 2017, pp. 296-306.
33. S.M. Ghafari, C. Tjortjis, "A Survey on Association Rules Mining Using Heuristics", *WIREs Data Mining and Knowledge Discovery*, Wiley, Vol. 9, no. 4, July/August 2019.
34. C. Tjortjis, L. Sinos and P.J. Layzell, "Facilitating Program Comprehension by Mining Association Rules from Source Code", *Proc. IEEE 11th Int'l Conf. Program Comprehension* (ICPC 03), IEEE Comp. Soc. Press, 2003, pp. 125-132.
35. D. Rousidis, C. Tjortjis, "Clustering Data Retrieved from Java Source Code to Support Software Maintenance: A Case Study", *Proc. IEEE 9th European Conf. Software Maintenance Reengineering* (CSMR 05), IEEE Comp. Soc. Press, 2005, pp. 276-279.
36. H. Zhang and X. Zhang, "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Trans. Software Eng.*, Sept. 2007, pp. 635-636.
37. T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Trans. Software Eng*., Vol. 33, No. 9, Sept. 2007, pp. 1-4.
38. C. Tjortjis, "Data Mining Code Clustering (DMCC): An Approach Supporting Software Maintenance and Comprehension", *Technical report*, School of Science & Technology, International Hellenic University, 2018, available at https://www.ihu.edu.gr/tjortjis/publications.htm
39. S.M. Ghafari and C. Tjortjis, "Association Rules Mining by improving the Imperialism Competitive Algorithm (ARMICA)", IFIP Advances in Information and Communication Technology, *Proc. 12th Int'l Conf. on Artificial Intelligence Applications and Innovations* (AIAI 2016), Vol. 475, Springer, 2016, pp 242-254.
40. D. Zhang, Y. Dang, J.-G. Lou, S. Han, H. Zhang, and T. Xie, "Software Analytics as a Learning Case in Practice: Approaches and Experiences", *Proc. Int'l Workshop on Machine Learning Technologies in Software Engineering* (MALETS 2011), 2011, pp. 55-58.