Facilitating Program Comprehension by Mining Association Rules from Source Code

Christos Tjortjis, Loukas Sinos, Paul Layzell
Department of Computation, UMIST, PO Box 88, Manchester, M60 1QD, UK
Email: christos@co.umist.ac.uk

Abstract

Program comprehension is an important part of software maintenance, especially when program structure is complex and documentation is unavailable or outdated. Data mining can produce structural views of source code thus facilitating legacy systems understanding.

This paper presents a method for mining association rules from code aiming at capturing program structure and achieving better system understanding. A tool was implemented to assess this method. It inputs data extracted from code and derives association rules. Rules are then processed to abstract programs into groups containing interrelated entities. Entities are grouped together if their attributes participate in common rules. The abstraction is performed at the function level, in contrast to other approaches, that work at the program level.

The method was evaluated using real, working programs. Programs are fed into a code analyser which produces the input needed for the mining tool. Results show that the method facilitates program comprehension by only using source code where domain knowledge and reliable documentation are not available or reliable.

1. Introduction

The purpose of this work is to facilitate program comprehension during software maintenance, by producing groupings of program entities according to their similarities, such as the use of variables, procedure calls and so on.

Software maintenance is the last and most difficult stage in the software lifecycle [17], and is 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, give rise to increased code complexity and deteriorated modularity [21], thus resulting in 50%-90% of the maintainers' time to be spent on *program comprehension* [9], [14], [15].

Understanding how a program is implemented and how it functions is a major factor when maintaining a computer program. Syntactic and semantic knowledge can be described in models (chunks) that explicitly represent different dependency relationships between various software artefacts [10]. However it is recognised that there are no explicit guidelines given a program understanding task, nor are there good criteria to decide how to represent knowledge derived by and used for it [4].

There are many types of tools available to help with software comprehension, emphasizing different aspects of systems and modules, and usually creating new representations for them [6], [19]. Some tools perform deductive or algorithmic analysis of program properties or structure, e.g. program slicers [18] or dominance tree analysers [5].

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 [3], [8], [12], [13], [16], [19], [20], [23], [24].

The remainder of this paper presents data mining methods used for program comprehension in section 2; section 3 describes the proposed method, including the data model and the algorithm used. Section 4 introduces the design and implementation details of the tool used as a demonstrator. Section 5 discusses results. Extensive conclusions and further work are respectively presented in sections 6 and 7.

2. Background

Data mining techniques can discover non-trivial and previously unknown relationships among records or

attributes in large databases [11]. This observation highlights the capability of data mining to educe useful knowledge about the design of large legacy systems. Data mining has three fundamental features that make it a valuable tool for program comprehension and related maintenance tasks [8]:

a) It can be applied to large volumes of data. This implies that data mining has the potential to analyse large legacy systems with complex structure.
b) It can be used to expose previously unknown non-trivial patterns and associations between items in databases. Therefore, it can be utilised in order to reveal hidden relationships between system or program components.
c) Its techniques can extract information regardless of any previous domain knowledge. This feature is ideal for maintaining software with poor knowledge about its functionality or implementation details.

Data mining has been previously used for clustering over a Module Dependency Graph (MDG) [13] and for identification of subsystems based on associations (ISA methodology) [8]. Both approaches provide a system abstraction up to the program level. The former creates a hierarchical view of system architecture into subsystems, based on the components and the relationships between components that can be detected in source code. This information is captured through an MDG, which is then analysed in order to extract the required structural view. The latter approach produces a decomposition of a system into data cohesive subsystems by detecting associations between programs sharing the same files.

Both methods provide system decomposition at the program level where each subsystem consists of a number of programs or files. The approach proposed in this paper produces decompositions at a lower level where the subsystem components could be functions or even single statements.

3. Proposed method

Association rules constitute one of the prominent data mining methods. Several algorithms have been proposed for mining association rules from large databases. Most of them deal with sales data stored in transactional databases and produce rules used for marketing purposes [2], [22], [24], [25].

The method proposed here analyses source code and produces useful rules derived from program entities, facilitating program structure comprehension. The method aims at producing a system abstraction at a level lower than previous approaches.

A tool utilising the *Apriori* algorithm [1] was built in order to extract association rules from source code elements. Programs can then be decomposed into groups containing entities which participate in common rules. The proposed method considers two key issues: modelling the input data and selecting an appropriate algorithm to be tailored and applied to these data. Both issues are discussed in the following subsections.

3.1 Data Model

Algorithms for mining association rules are traditionally used for market basket analysis. This work addresses a different application domain: that of source code analysis, thus a suitable data model is needed. This model should comply with formatting requirements imposed by existing algorithms.

Market basket analysis employs the *<transaction, items>* model, where each transaction consists of a number of items purchased. For source code analysis, we use blocks of code instead of transactions; thus blocks correspond to the *entities* of the model. The blocks of code may contain low-level code elements such as variables, data types and calls to other blocks. These elements correspond to *attributes* of the model. Table 1 summarises the proposed data model, which is of the form *<code entity, attributes of this entity>*.

Attributes need to be qualitative, that is boolean, so as to apply existing algorithms without major changes in their reasoning. A table where each row stands for a block of code and each column stands for an attribute represents the model. The value of the attribute is "1" when the corresponding code element is used in the relevant block otherwise it has a null value.

Table 1: Data models in Market Basket Analysis and Source Code Analysis

DATA MODEL	*MARKET BASKET ANALYSIS*	*SOURCE CODE ANALYSIS*
ENTITIES	Transactions	Blocks of code
ATTRIBUTES	Market Items	Variables, data types, calls to blocks of code

It is important to define blocks of code in such a way that the majority of the source code is contained within a block. When defining entities, as blocks of code, in most programming languages there are two alternatives: using *individual statements* or *modules*. The former would result in analysing a program to its entirety. However, this analysis would be very detailed and the results extremely complex, difficult and time consuming to understand. Furthermore, many statements cannot be considered in isolation, especially if they participate in a compound statement such as *<if ... else>*, *<switch>* and so on.

Representing these statements as single entities may produce misleading information about the program.

This is the reason why we chose to use modules (functions, classes, procedures) for defining blocks. Modules are defined in several programming languages and can be easily identified within a program as their start and end follow specific conventions. This facilitates automated input model extraction from the source code. Modules have the advantage of including a significant amount of code that consists of several statements such as variable declarations and assignments. As a result, modules may contain a large number of attributes. Furthermore this choice facilitates grouping of related modules, as it is much easier to identify commonalties on the large range of attributes. More attributes in common imply better quality of association rules and of consequent groups.

In summary, a data motel with program modules for entities is more suitable for the purposes of the proposed method. Nevertheless, adopting a single statement model can always be considered as a complementary one, in case a more detailed decomposition of a program is required.

3.2 Algorithm description

As discussed previously, the data model used as input consists of blocks of code and code elements corresponding to entities and attributes respectively. Each entity is represented by a record in a database table. The algorithm inputs a table and produces a set of block groups. The blocks in each group have attributes in common, which occur in the same association rules. The number of common association rules indicates how strong the relationship among the blocks is. Analysing programs into such block groups results in a view of the program structure.

The algorithm we used is decomposed into three phases:

1. *Creation of large itemsets*.
   By means of the user-defined minimum *support*, find all the sets of items with support equal to or greater than this minimum, where support is the percentage of block entities that contain the itemset.
2. *Creation of association rules*.
   Use the large itemsets to generate the required association rules. Only rules with *confidence* equal to or greater than a user-specified minimum must be produced. For a rule of the form $A \Rightarrow B$, confidence is defined as support($A \cup B$) / support(A).
3. *Generation of block groups*.
   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.

The first two phases are based on the Apriori algorithm [2]. The last one involves producing groups containing block entities with attributes that belong to the same association rules. Compared to the ISA methodology which uses the same algorithm, the proposed method does not terminate when large itemsets are created. It also extracts important association rules between source code components and, based on these associations, reveals related blocks of code. Thus, it provides two-level information about the program under examination: related blocks of code and associated program attributes.

4. Tool Design and Implementation

To illustrate the scope and assess the applicability of the method a tool was designed and implemented. It consists of three subsystems, each one of which performs a specific operation:

- *Database Management Subsystem*
  This is responsible for managing the data either provided by the user (input data) or created during the mining process (tables of rules and groups). It handles all database operations and ensures the integrity of stored data. It also determines the way of communication between the tool and the database.
- *Processing Subsystem*
  It is the heart of the system, responsible for executing the algorithm for mining association rules, as described earlier. It inputs data from the Database Management Subsystem, performs the three processing phases (creation of itemsets, creation of rules, generation of block groups) and sends results to the Input/Output Subsystem for display.
- *Input / Output Subsystem*
  It is responsible for communication with the end-user. It reads values for minimum support and confidence, informs the user of the processing progress, and displays results.

The main application window (Fig. 1) displays information during all processing phases. It contains three grids for displaying the created itemsets, rules and block groups respectively. It also contains two read-only text boxes for displaying the current minimum support and confidence, and buttons for triggering the creation of rules and groups. Through two combo boxes, the user can select groups to display, according to their size and the number of common rules.

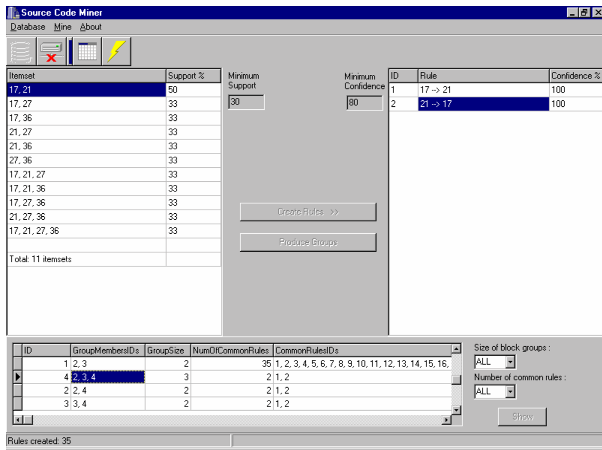

Fig. 1: Main Window

5. Results

The method was tested using real programs which were inputted to the tool. For evaluation we used parts of a software system implemented in COBOL that manages student accommodation. It consists of a number of small programs of average size of 1000 lines of code which implement various functions. We will briefly discuss the results achieved by applying the method to one of these programs, '*Register*', as a case study. Similar results were acquired when other programs were used.

The data model used for COBOL involves procedures as entities, and binary attributes depending on the presence of user-defined and language-defined identifiers. More detailed description of the model can be found at [7]. Minimum support and confidence were set to 5% to 90% respectively in order to produce a considerable amount of rules, enough for allowing their grouping.

Table 2 displays the groups of procedures produced. Figure 2 shows a graphical representation of the results. Each procedure is represented by a box with a number in it corresponding to the relevant procedure. Edges connect procedures with common rules. The number of common rules is shown at the intersection of the edges. Groups of procedures are used to form "collections" indicated by polygons.

As depicted in Figure 2, some procedures (like 19, 24 and 36) participate in many groups. When placing procedures in collections, two criteria were 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 the collection a procedure belongs, a larger set of association rules is produced for disambiguation.

For example, procedure 24 belongs to collection C instead of B, because it has more common rules with procedures in collection C (9 common rules with procedure 15) than with objects of collection B (4 common rules with procedure 36). Procedure 36 has the same number of common rules (2) with procedure 27 in collection A and procedures 17, 21 and 26 in collection B. However, according to the second criterion, it belongs to collection B, because it is connected to more procedures from this collection. Procedure 19 has 9 common rules with two procedures from collection A and two from collection C. Thus an experiment with lower confidence (20%) was needed to realise that it is more related to procedure 27, so it belongs to collection A.

Table 2: Produced Groups

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

Results were assessed by referring to the source code of the program. For example, collection D consists of procedures *do-letter*(23), *screen-down*(45) and *screen-up*(46). These procedures have two common rules: 4 (2 → 119) and 21 (2, 17 → 119). These rules contain the following items: 2 (*ring-bell*), 17 (*reg-screen-no*) and 119 (*pr_show-reg-screen*[1]). As shown in Fig. 3, the three procedures use all these items (shown in bold characters) and actually perform similar operations. Thus, collection D gives a potential subsystem of the initial program. All produced collections were checked against the source code, and it was found that they indeed contain related procedures.

Another method for testing the validity of results is by comparing them to a mental model that humans modulate about the program structure. For the tested program, one of the original developers of the system who is responsible for its maintenance was requested to produce a mental

[1] Prefix "pr" indicates a procedure call. Item *pr_show-reg-screen* is a call to procedure *show-reg-screen*.

model for it, by distributing its procedures into subsystems according to their functionality. This model consists of ten subsystems shown in Table 3 (*control*, *interface* and *students* being the most important). Comparing the model to the produced program decomposition, the following results were obtained:

- Collection A: all procedures belong to the *students* subsystem (100% success).
- Collection B: half of the procedures belong to the *control* subsystem and the remaining to the *students* subsystem (50% success). However, all control procedures were placed on the same collection.
- Collection C: three procedures belong to the *students* subsystem and one to the *help* subsystem (75% success).
- Collection D: two procedures belong to the *interface* subsystem and one to the *students* subsystem (success 66%). However, procedure *do-letter* handles printing operation, which can be considered as an interface to the user.

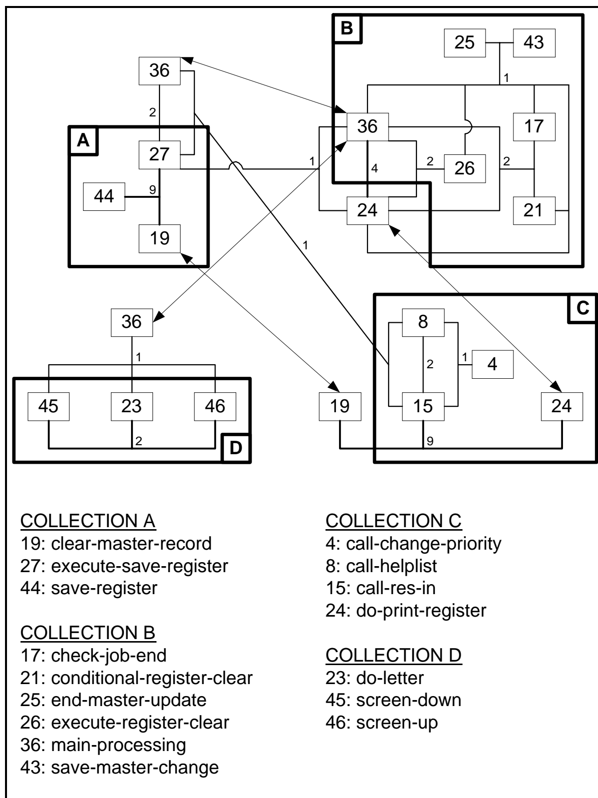

Fig. 2: Visual Representation of Produced Groups

Summarising, 72.75% of the produced abstraction is consistent to the mental model, a relatively good result considering the following facts: a) the mental model does not capture the interconnections between procedures from different subsystems, although these connections play an important role in the system operation, and b) mental models tend to be subjective, depending on the mental processes of the human brain.

6. Conclusions

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

```
        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.
***********************************************
        do-letter.
            IF reg-screen-no = 4
               PERFORM set-new-classification
               PERFORM show-reg-screen
            ELSE
               CALL ring-bell
            END-IF.
```

Fig. 3: Code Extract from 'Register'

Table 3: Mental Model for 'Register'

Subsystem	Procedure
booking type	call-bkdef
Control	check-job-end, conditional-register-clear, conditional-register-exit, goto-reject, goto-transfer, initialisation, main-control, main-processing, termination
dates	call-daterels
errors	call-error
event history	call-res-history, register-event-history
help system	call-helplist
Interface	call-caps, call-left-caps, call-left-upper, call-moveleft, call-upper, clear-error, clear-upper-lower, inc-line-col, screen-down, screen-up, set-cur-pos
log file	put-day
students	execute-save-register, get-next-free-entry, save-register, accept-reg-screen, call-res-in, get-note-indicator, set-night-text, show-reg-screen, do-letter, do-print-register, print-register-conditionally, call-change-priority, call-details, call-i-request, call-request, clear-master-record, end-master-update, execute-register-clear, issue-register-letter, prepare-register-details, prepare-resident-open-room, read-master, save-master-change, set-classify-status, set-new-classification
user validation	valid-user-processing

Input tables produced for source code analysis tend to be sparse. This can be explained by the fact that a program usually contains variables used for specific reasons, by specific procedures. Only some global variables are 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 corresponding to variables used. 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 set in order to produce a number of large itemsets. By decreasing support, more items pass the support threshold, more large itemsets are created, and thus, more information about block groups can be retrieved.

The groups produced may not contain all the procedures. Excluded procedures normally contain items with very low support, below the user specified minimum. These procedures may perform special operations and normally do not appear in any group. However, if minimum support is not low enough, important information may be omitted and several 'interesting' procedures may not be present in the results. Thus it is important to keep low support values to ensure that only 'isolated' procedures of limited interest are excluded from results.

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 in order to produce groups of interconnected procedures that deal with specific tasks.

Some procedures possibly participate in many groups. These procedures are likely to contain a larger number of items than others. For example, in the test presented in the previous section, procedure 36 is present in eight groups out of a total of thirteen. As expected, this procedure is the main procedure of the program that contains 32 items (variables and calls to other procedures), when all the other procedures do not contain more than 10 items. Generally, such procedures can be considered as communication links between different groups.

Generated results need to be summarised and interpreted, in order to create meaningful collections of program entities. The creation of such collections is based on the result groups. Procedures that participate in a group having a large number of common rules undoubtedly belong to the same group. However, the number of common rules is not the only criterion. The occurrence of the common rules among the groups may also indicate a relation between some procedures. For example, in the test presented in the previous section, procedures 4, 8 and 15 share only one common rule (13). 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, so as 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 if view. However, this method is not reliable enough because different people may have a different mental model about the same program, according to their mental processes.

7. Further Work

As mentioned earlier, the suggested approach for mining association rules from source code is new. This section provides a number of possible enhancements to

the approach and the tool in particular, that should be performed in order to achieve a complete solution to the problem of source code analysis. These improvements include:

- *Testing other algorithms for mining association rules.*

Apriori was selected to be the base algorithm for designing the tool. However, it would be challenging to use other algorithms for mining association rules that may perform equally well or even better than Apriori, especially in cases with very large databases. This comparative assessment could result in a toolset for mining rules.

- *Formation of a quantitative input model.*

The data model proposed in this method is qualitative. A possible enhancement would be the formation of a quantitative data model. Such a model would be richer as it could capture information not only for the existence/absence of code items in blocks, but also for the number of item occurrences inside blocks. For example a rule of the form *<itemA: 3 → itemB: [2 … 5]>* would mean: "if itemA occurs 3 times then itemB occurs 2 up to 5 times". We should note that adoption of a quantitative data model requires an algorithm that can produce and process quantitative association rules.

- *Automatic derivation of input data.*

The proposed method assumes the existence of input data from source code, which can be extracted either manually or automatically. However, in real conditions, it is impossible to manually create input data from programs consisting of thousands LOC. Even for smaller programs, this task is time consuming. Thus, in order to provide an integrated solution to source code analysis, the automatic creation of input data should be incorporated in the methodology. The present prototype tool is equipped with a source code analyser which can extract data from source code or even the outcome of a parser.

- *Further testing on larger programs.*

Validation tests were executed on programs containing up to 1000 lines of code. In order to achieve a better evaluation of the produced results, more tests should be performed on larger programs. However, in order to proceed to these tests, the previous enhancement should be implemented first, so as to get the input data from these programs automatically.

- *Source code analysis at the statement level.*

As previously mentioned block entities can be either modules (functions, classes, procedures) or single statements. Up to now, the suggested approach concentrated on a model consisting of modules and the ultimate aim was to create program decomposition in groups containing interconnected modules. On the other hand, the tool can also be used in order to perform a source code analysis in the statement level. To do so, the input data should consist of records representing statements and the final results will produce groups of related statements. This kind of analysis is useful, when local parts of a program need to be understood.

References

[1] R. Agrawal, T. Imielinski and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *Proc. of the ACM SIGMOD Conference on Management of Data*, 1993, pp. 207-216.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proc. 20th Int'l Conf. Very Large DataBases* (VLDB 94), 1994, pp. 487-499.

[3] N. Anquetil and T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization method", *Proc. 6th Working Conf. Reverse Engineering* (WCRE 99), IEEE Comp. Soc. Press, Oct. 1999, pp. 235-255.

[4] F. Balmas, H. Wertz and J. Singer, "Understanding Program Understanding", Proc. 8th Int'l Workshop Program Comprehension (IWPC 00), IEEE Comp. Soc. Press, 2000, pp. 256.

[5] E. Burd, M. Munro, "Evaluating the Use of Dominance Trees for C and COBOL", Proceedings of the International Conference on Software Maintenance, Oxford, England, August 30-September 3, 1999, IEEE Computer Society Press, 1999, ISBN 0769500161, pp. 401-410.

[6] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "Decomposing legacy systems into objects: an eclectic approach", Information and Software Technology, Vol. 43, 2001, pp 401-412.

[7] K. Chen, C. Tjortjis and P.J. Layzell, "A Method for Legacy Systems Maintenance by Mining Data Extracted from Source Code", *Case studies of IEEE 6th European Conf. Software Maintenance and Reengineering* (CSMR 2002), IEEE Comp. Soc. Press, 2002, pp. 54-60.

[8] 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.

[9] K. Erdös and H.M. Sneed, "Partial Comprehension of Complex Programs (enough to perform maintenance)", *Proc 6th Int'l Workshop Program Comprehension* (IWPC 98), IEEE Comp. Soc. Press, 1998, pp. 98-105.

[10] A.R. Fasolino and G. Visaggio, "Improving Software Comprehension through an Automated Dependency Tracer", *Proc. 7th Int'l Workshop Program Understanding* (IWPC 99), IEEE Comp. Soc. Press, 1999.

[11] U. M. Fayyad, G. Piatetsky-Shapiro and P. Smyth. "From Data Mining to Knowledge Discovery: An Overview", in *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996.

[12] A. Lakhotia, “A Unified Framework For Expressing Software Subsystem Classification Techniques”, *Journal of Systems and Software,* Vol. 36, No 3, pages 211--231 1997.

[13] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen and E. R. Gansner, “Using Automatic Clustering to Produce High-Level System Organisations of Source Code”, *Proc. 6th Int'l Workshop Program Understanding* (IWPC 98), IEEE Comp. Soc. Press, 1998, pp. 45-53.

[14] A. Von Mayrhauser and A.M. Vans, *Program Understanding – A Survey*, Technical Report CS-94-120, Dept. of Computer Science, Collorado State University, August 1994.

[15] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, Wiley Computer Publishing, 1996.

[16] 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.

[17] I. Sommerville, *Software Engineering*, 6th edition, Harlow, Addison-Wesley, 2001.

[18] F. Tip, "A Survey of Program Slicing Techniques", Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[19] 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.

[20] C. Tjortjis and P.J. Layzell, “Using Data Mining to Assess Software Reliability”, *Suppl. Proc. IEEE 12th Int’l Symposium Software Reliability Engineering* (ISSRE2001), IEEE Comp. Soc. Press, 2001, pp. 221-223.

[21] 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.

[22] H. Toivonen, “Sampling Large Databases for Association Rules”, *Proc. 22nd Int’l Conf. Very Large Databases* (VLDB 96), 1996, pp. 134-145.

[23] V. Tzerpos and R. Holt, “Software Botryology: Automatic Clustering of Software Systems”, *Proc. 9th Int'l Workshop Database Expert Systems Applications* (DEXA 98), IEEE Comp. Soc. Press, 1998, pp. 811-818.

[24] T. A. Wiggerts, “Using Clustering Algorithms in Legacy Systems Remodularization”, *Proc. 4th Working Conf. Reverse Engineering* (WCRE 97), IEEE Comp. Soc. Press, 1997, pp. 33-43.

[25] M.J. Zaki, S. Parthasarathy, M. Ogihara and W. Li, “New Algorithms for Fast Discovery of Association Rules”, *Proc. of the 3rd Int’l Conf. Knowledge Discovery Databases and Data Mining*, 1997, pp. 283-286.