Metarule-guided association rule mining for program understanding

O. Maqbool, H.A. Babri, A. Karim and M. Sarwar

Abstract: Software systems are expected to change over their lifetime in order to remain useful. Understanding a software system that has undergone changes is often difficult owing to the unavailability of up-to-date documentation. Under these circumstances, source code is the only reliable means of information regarding the system. In the paper, association rule mining is applied to the problem of software understanding i.e. given the source files of a software system, association rule mining is used to gain an insight into the software. To make association rule mining more effective, constraints are placed on the mining process in the form of metarules. Metarule-guided mining is carried out to find associations which can be used to identify recurring problems within software systems. Metarules are related to re-engineering patterns which present solutions to these problems. Association rule mining is applied to five legacy systems and results presented show how extracted association rules can be helpful in analysing the structure of a software system and modifications to improve the structure are suggested. A comparison of the results obtained for the five systems also reveals legacy system characteristics, which can lead to understanding the nature of open source legacy software and its evolution.

1 Introduction

Legacy systems are old software systems that are crucial to the operation of a business. These systems are expected to have undergone changes in their lifetime owing to changes in requirements, business conditions and technology. It is quite likely that such changes were made without proper regard to software engineering principles. The result is often a deteriorated structure, which is unstable but cannot be discarded because it is costly to do so. Moreover, another reason for retaining these legacy systems is that they have embedded business knowledge which is not documented elsewhere.

Since it is often not feasible to discard a system and develop a new one, techniques must be employed to improve the structure of the existing system. An effective strategy for change must be devised. Strategies for change mentioned in [1] include maintenance, architectural transformation and re-engineering. Re-engineering is a process that re-implements legacy systems to make them more maintainable [1]. According to [2], re-engineering is any activity that improves one's understanding of software or prepares/improves the software itself, usually for increased maintainability, reusability or evolvability.

Given the fact that software maintenance usually accounts for over 50% of project effort [1, 3, 4], making it the single most expensive software engineering activity [1], and perhaps the most important life cycle phase [5], the need for re-engineering to ease the maintenance effort is justified. The re-engineering option should be chosen when system quality has been degraded by regular change, but change is still required, i.e. the system under consideration has low quality but a high business value, and the re-engineering effort is less risky and less costly than system replacement.

The re-engineering effort starts with gaining an understanding of the software system, a process known as reverse engineering. Understanding is critical to many activities including maintenance, enhancement, reuse, design of a similar system and training [6]. Reverse engineering has been heralded as one of the most promising technologies to combat the legacy systems problem [7]. However, gaining system understanding is difficult because documentation for the system is often not available and source code files are the only means of information regarding the system. According to [8], system understanding takes up 47% of the software maintenance effort. Hall [9] places the system understanding effort at 47–62%. Tools and techniques are thus required to make the program understanding task easier. Tools provide automated support for system understanding at the procedural level by extracting the procedural design or at a higher level by extracting the architectural design.

In the past, the application of deductive techniques to different aspects of software engineering, including program understanding, has been more frequent than the use of inductive techniques [10]. Deductive techniques usually employ some knowledge base as their underlying technology which is used to deduce relations within the software system. Great effort is required to build the knowledge base and continuously maintain it. Moreover, algorithms used for deductions may be computationally demanding [10]. Researchers have thus started exploring the use of inductive or data mining techniques in software engineering. Data mining is considered one of the most promising interdisciplinary developments in the information industry [11].

IEE Proceedings online no. 20050012

doi:10.1049/ip-sen:20050012

Paper received 7th March 2005

O. Maqbool and A. Karim are with Lahore University of Management Sciences, DHA, Lahore, Pakistan

H.A. Babri and M. Sarwar are with the University of Management and Technology, 52-L, Gulberg III, Lahore, Pakistan

E-mail: onaiza@lums.edu.pk

In this paper, our focus is on the use of data mining, or more specifically, on association rule mining for discovering interesting relationships within legacy system components which lead to program understanding. These relationships can be used to identify recurring problems within legacy systems, and are thus related to re-engineering patterns. Patterns were first adopted by the software community as a way of documenting recurring solutions to design problems [12]. Since then, they have been used as an effective means to communicate best practice in various aspects of software development including re-engineering. Re-engineering patterns for object-oriented legacy systems were identified based on experiences during the FAMOOS project [13]. Our focus in this paper is on traditional legacy systems developed using the structured approach. Our paper presents an approach to gain insight about the structure of these legacy systems by applying metarule-guided association rule mining. Metarule-guided association rule mining is carried out as a two step process. In the first step, we formulate metarules based on knowledge of typical problems in legacy systems. These metarules place constraints on the form of association rules to be mined in addition to on values of interestingness measures. In the second step, metarule-guided mining is applied to the legacy system source code to mine association rules. By relating the metarules to re-engineering patterns, system understanding is gained, which allows suggestions for making subsequent changes and optimisations to the source code for better maintainability. Thus we make two major contributions in this paper. First, we show how metarule-guided association rule mining can be used to mine legacy system source code to gain insight into its structure and detect re-engineering opportunities. Second, we formulate re-engineering patterns for structured legacy systems and relate each metarule to a re-engineering pattern, thus presenting solutions to problems identified by a metarule.

2 An overview of association rule mining

Association rule mining is a data mining technique that finds interesting association or correlation relationships among a large set of data items [11]. Traditionally, association rule mining has been employed as a useful tool to support business decision making by discovering interesting relationships among business transaction records.

To illustrate the concept of association rule mining, consider a set of items $I = \{i_1, i_2, \ldots, i_n\}$. Let D be a set of transactions, with each transaction T corresponding to a subset of I. An association rule is an implication of the form $A \Rightarrow B$ where $A \subset I$, $B \subset I$ and $A \cap B = \phi$. As an example, consider a set of computer accessories (CDs, memory sticks, microphones, speakers) that are available at a certain store. These accessories form the set of items I of interest to us. Every sale made represents a transaction T. Suppose the sales made are represented in the form of the following set of transactions D in Table 1.

Table 1: A set of transactions representing sales of items

Transaction ID	Items
T1	CD, memory stick
T2	CD
T3	microphone, speaker
T4	CD, speaker, microphone
T5	memory stick, microphone, speaker

Association rules in the above case represent items that tend to be sold together e.g. the association rule *microphone* $\Rightarrow$ *speaker* shows that those who buy microphones also tend to buy speakers. This association rule can also be written as $buys(X, \text{‘}microphone\text{’}) \Rightarrow buys(X, \text{‘}speaker\text{’})$, where the variable X represents customers who bought the items. Since this association rule contains a single predicate (buys), the association rule is referred to as a single-dimensional association rule. When more than one predicate is present, the rule is referred to as multi-dimensional.

A large number of such association rules may exist in a given set of transactions and not all of them may be of interest e.g. the association rule $CD \Rightarrow$ *speaker* can be inferred from the transaction set in Table 1 but it is clear that this association is not as strong (or interesting) as the one between microphones and speakers. An association rule is said to be interesting if it is easily understood, valid, useful, novel or validates a hypothesis that the user sought to confirm [11]. To find interesting rules, support and confidence are commonly used as objective measures of interestingness. Support represents the percentage of transactions in D which contain both A and B. Confidence is the percentage of transactions in D containing A that also contain B. Another measure of interestingness is coverage. The coverage of an association rule is the proportion of transactions in D that contain A. In terms of probability:

$$\text{Support}(A \Rightarrow B) = P(A \cup B)$$
$$\text{Confidence}(A \Rightarrow B) = P(B|A)$$
$$\text{Coverage}(A \Rightarrow B) = P(A)$$

In Table 1, *microphone* $\Rightarrow$ *speaker* is an association rule with support 3/5, confidence 1, and coverage 3/5. This rule is more interesting than $CD \Rightarrow$ *speaker* which has support 1/5, confidence 1/3, and coverage 3/5.

In order to make the data mining process more effective, a user may place various constraints under which mining is to be performed. These include interestingness constraints which involve specifying thresholds on measures of interestingness (such as support, confidence and coverage), and rule constraints which place restrictions on the number of predicates and/or on the relationships that exist among them, based on the analyst's experience, expectations or intuition regarding the data [11]. Rule constraints thus specify the form of the association rule to be mined and are expressed as rule templates or ‘metarules’. As an example, suppose we have customer related data for the sales in Table 1, and we want to associate customer characteristics with sales of speakers. A metarule for the information in which we are interested is of the form $P(X,Y) \Rightarrow buys(X, \text{‘}speaker\text{’})$ where variable X represents a customer and variable Y takes on values of the attribute assigned to the predicate variable P. During the mining process, rules which match this metarule are found. One example of a matching rule is $age(X, \text{‘}young\text{’}) \Rightarrow buys(X, \text{‘}speaker\text{’})$. Since both the predicate variable P and variable Y may vary, rules $gender(X, \text{‘}male\text{’}) \Rightarrow buys(X, \text{‘}speaker\text{’})$ and $age(X, \text{‘}old\text{’}) \Rightarrow buys(X, \text{‘}speaker\text{’})$ also satisfy the metarule.

3 Re-engineering patterns for structured legacy systems

Patterns have been used in various disciplines to enable collective experiences to be documented and shared. In the software engineering domain, they have been used effectively to communicate best practices in various aspects of software development including the development process, design, testing and re-engineering. As pointed out in Section 1, re-engineering is carried out to improve the structure of legacy software systems to make them more maintainable. Although there are different reasons for re-engineering, e.g. improving performance, porting the system to a new platform, exploiting new technology, the actual technical problems within legacy systems are often similar and hence some general techniques or patterns can be utilised to aid in the re-engineering task [12]. Re-engineering patterns record experiences about understanding and modifying legacy software systems and thus present solutions to recurring re-engineering problems. Mens and Tourwe [14] point out that trying to understand what a legacy system does, what parts of the legacy system to change and the potential impact of these changes is a significant problem. Unlike forward engineering, which is supported by processes, established processes are not available for re-engineering. Re-engineering patterns fill this gap by providing expertise that can be consulted.

The re-engineering cycle typically involves some form of reverse engineering to understand the system and to identify problems areas, followed by forward engineering to restructure a system. Re-engineering patterns in [12] focus on the re-engineering cycle of object-oriented legacy systems, although some of the identified patterns may apply to software systems which are not object-oriented. Examples of typical problems in object-oriented legacy systems include lack of cohesion within classes, misuse of inheritance and violation of encapsulation [12].

In this paper, we present re-engineering patterns for legacy systems developed using the structured approach. These patterns help in identifying recurring problems within structured code and offer suggestions for improvement. Similar to other software patterns, they capture structural level problems that may not be evident from any one part of a legacy system. It is relevant to note that the re-engineering patterns in [12] capture the entire re-engineering cycle, starting from 'setting the right direction' and 'first contact' with the system, to actually restructuring (in the context of object-oriented systems, refactoring) the system to remove some of the problems which appear in object-oriented code as it evolves. Most of the patterns that deal with initial phases of re-engineering apply to both object-oriented and structured systems. However, most of the patterns dealing with restructuring take into account problems within object-oriented code. In this paper, we do not attempt to re-capture all re-engineering patterns, since as already pointed out, many patterns especially those involving the initial re-engineering phases are equally valid for structured systems. Our focus is on identifying problems which arise in structured systems as they evolve. Thus the patterns identified in this paper complement those identified in [12] by addressing some typical problems within legacy structured systems.

4 Metarule-guided association rule mining for program understanding

In this paper, we employ association rule mining to aid the design discovery process of structured legacy systems. To make our mining process more effective, we use constraint-based mining. Constraints are placed on the form of association rules to be mined and on interestingness measure values. These constraints are specified as metarules. For simplicity, in the following Sections a metarule is used to refer to a combination of association rule form and interestingness measure thresholds rather than a constraint on association rule form only. This is due to the fact that we employ such a combination to reveal system characteristics. The metarules we develop are based on knowledge of some typical problems in legacy systems and serve as templates to mine meaningful association rules. Each metarule is related to a re-engineering pattern. Re-engineering patterns focus on the process of discovery i.e. they discover an existing design, identify problems and then present appropriate solutions [12]. A metarule related to a re-engineering pattern aids in this discovery process by capturing symptoms of typical problems that arise in legacy software systems.

Figure 1 illustrates our rule mining approach as a two-step process. In the first step, we use knowledge of generic problems within structured legacy systems to arrive at metarules. In the next step, metarule-guided mining is carried out on the test legacy systems to find association rules which serve as indicators of problems that are present in the system. We relate metarules to re-engineering patterns by describing how a re-engineering pattern presents a solution to problems within a legacy system, as identified by the mined association rules.

In this Section, we discuss various factors that must be taken into account during formulation of metarules. Results of carrying out metarule-guided mining on test systems are described in Section 5.

4.1 Selecting items and transactions

A metarule can be thought of as a hypothesis regarding relationships that a user is interested in confirming [11]. As a first step, it is necessary to identify items between which relationships are to be mined. In our case, the guiding principle is to choose items which facilitate understanding of the source code and identification of problems, and also allow suggestions for restructuring the code for greater maintainability. Most of the legacy software systems that exist have been developed using the structured approach, with functions or routines forming basic components. Moreover, in legacy software, the use of global variables is often widespread leading to difficulty in understanding the code. In view of these facts, we decided to use functions and global variables as items. Moreover, we also decided to use user defined types. The reason is that user

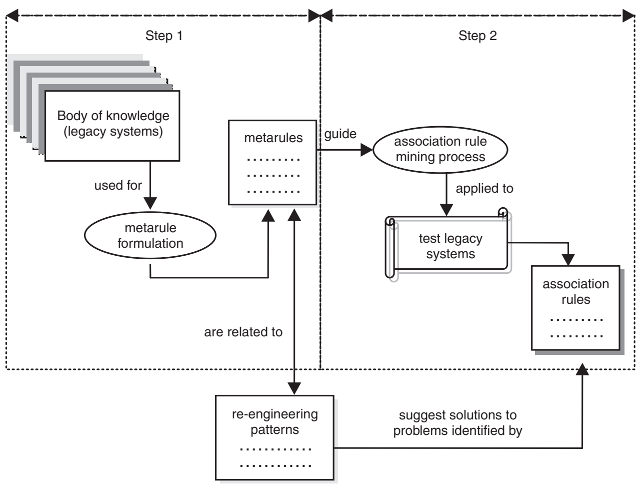

Fig. 1 *Our rule mining approach*

defined types become potential data objects when code is to be restructured as object-oriented code.

To illustrate the item and transaction sets, let $F = \{f_1, f_2, \ldots, f_n\}$ be the set of all functions, $G = \{g_1, g_2, \ldots, g_k\}$ be the set of all global variables, and $U = \{u_1, u_2, \ldots, u_m\}$ be the set of all user defined types present in a software system. The item set I is the union of these sets i.e. $I = F \cup G \cup U$. Each transaction T corresponds to a subset of I and denotes the functions called, global variables and user defined types accessed by a function within the software system (by an access, we mean a reference to a variable by using or setting its value, or taking its address). As an example, consider a software system consisting of two functions (f_1 and f_2) only. Suppose f_1 accesses global variables g_1 and g_2, user defined types u_1, u_3 and u_5, and calls the function f_2. Suppose f_2 accesses global variables g_2 and g_3, user defined types u_1, u_2 and u_3, and does not call any function. The transaction set for this system is presented in Table 2.

Thus in the transaction set that we use, each transaction consists of items in the form of global variables and user defined types accessed by a function, and function calls made by the function. Therefore, there are as many transactions as the number of functions in the software system. To obtain results that are meaningful in the re-engineering context, the function which accesses global variables, user defined types and makes function calls is also treated as an item.

4.2 Setting appropriate thresholds for association rules

Association rules that are mined using rule-based constraints which restrict the rule form may not all be interesting. For finding interesting association rules which indicate problems or re-engineering opportunities, we require interestingness constraints. Traditionally high thresholds of support, confidence and coverage have been used for finding interesting association rules [11]. We use both low and high thresholds of the three measures: coverage, support and confidence. Low thresholds can reveal interesting facts about the software and provide insight into its structure, as illustrated by patterns that we describe in Section 4.4. In general, assigning fixed values to thresholds such as 0.7 for high and 0.3 for low is not desirable owing to the following reasons [12]. First, threshold values should be selected based on coding standards used by the development team, and these may not be available. Secondly, tools that employ thresholds often display only abnormal entities, so the user is not aware of the number of normal entities which may result in a distorted view of the system. A study on software metrics, undertaken with various threshold values [15] revealed that observations were independent of threshold value. Owing to the reasons cited, it may not be meaningful to fix an absolute threshold, since system characteristics such as size, number of global variables, user defined types and functions, can vary widely from system to system. Since the confidence measure does not depend on these system characteristics and is related to the association between two items only, setting thresholds for the confidence measure is not an issue as compared to the other measures.

Oftentimes instead of setting absolute thresholds, researchers advocate other techniques for interpreting results. Demeyer *et al.* [12] suggest that data be presented in a table and sorted according to various measurement values to obtain a meaningful interpretation. To determine a relationship among data points describing one or two variables, Fenton and Pfleeger [16] use box plots and scatter plots. Box plots utilise medians and quartiles to represent the data and suggest outliers. Whereas a box plot hides most of the expected behaviour and shows unusual values, a scatter plot graphs all the data points to reveal trends. Histograms are also used to reveal the distribution of data.

For displaying and interpreting results, we adopt the following approach. Each re-engineering pattern in Section 4.4 has an associated metarule, which specifies the association rule form. For metarules involving low or high thresholds of support and coverage, we mine the system to find all association rules of the form specified by the metarule. As suggested in [12, 16], the entire results are displayed using histograms or other meaningful graphs. High and low thresholds for the support and coverage measures are then determined for the system under consideration interactively by visually inspecting the results of metarule-guided mining. Interactive determination of thresholds may be simplified by indicating certain measures e.g. mean, median, quartiles on the graph.

4.3 Selecting interesting association rules in the re-engineering context

It is relevant to note that if we employ user defined types, functions and global variables as items, and use coverage, support and confidence interestingness measures with thresholds low, high and one, the number of possible metarules is quite large. If we consider global variables, user defined types, and functions as items and restrict the maximum number of predicates on the left and right hand sides of association rules to one, the possible number of metarules is 3^2. Differentiating between calling functions and called functions increases this number to 4^2. Additionally, we have three interestingness measures, each of which can have three possible values. Thus the number of metarules is $4^2 \times 3^3 = 432$. This number is even larger if we remove the restriction on the maximum number of predicates. Useful metarules need to be filtered using subjective interestingness measures. These measures are based on user beliefs in data and find rules interesting if they are unexpected or offer information on which the user can act [11].

We reduce the total possible metarules to a small subset by keeping in mind that our purpose is to use these metarules to identify re-engineering opportunities. Thus we select metarules that highlight some typical problems in legacy structured systems. Since we use items based on the set of global variables, user defined types and functions, our metarules are further restricted to problems that are related to this item set.

Amongst the typical problems that occur in legacy code is lack of modularity i.e. strong coupling between modules, which hampers evolution [12]. Although software is expected to evolve over time, the guideline is that evolution should not degrade the quality of software. One way

Table 2: A sample set of transactions illustrating items used in our experiments

Calling function	Global variables	User defined types	Called functions
f_1	g_1, g_2	u_1, u_3, u_5	f_2
f_2	g_2, g_3	u_1, u_2, u_3	

to reduce coupling between modules and thus reduce side effects when changes are made is to reduce the number of global variables or to reduce their scope. Metarules 1 and 2, which are related to patterns 1 and 2 (Table 3), are used to detect global variables that can be localised and thus reduce coupling. Another means to ensure that side effects are detected and do not propagate throughout the system is to encourage functions that are coupled to be identified and placed together. Metarule 3, related to pattern 3 (Table 3), helps in identifying these coupled functions. Duplicated functionality is another typical problem in legacy systems, since quite often various teams re-implement similar functionality [12]. To avoid this problem, functions should share code by making use of utility functions. Use of utilities is facilitated if these utilities are identified and catalogued for reference. Metarule 4, related to pattern 4 (Table 3), deals with identification of utilities. Legacy systems may need to be modularised so that they can be understood more easily and maintained. One modularisation option is to convert a structured system to an object-oriented system, by identifying data items that are potential objects and related functions which act

Table 3: Re-engineering patterns and metarules

Pattern category 1: Enhance modularity and control side effects	
Pattern 1: Reduce global variable usage	*Intent:* Remove global variables that are used by very few functions.
Metarule 1	
Form: Global (X, Y) ⇒ Accessed by (X)	*Coverage: Low*
Implication: Only a small proportion of functions in the system access the global variable(s) on the LHS.	
Pattern 2: Localise variables	*Intent:* Reduce the scope of variables that have larger scope than necessary.
Metarule 2	
Form: Global(X, Y) ⇒ Accessed by (X)	*Confidence: One*
Implication: The global variables are used by one function only.	
Pattern 3: Increase locality of reference	*Intent:* Group related functions together to improve understandability and maintainability.
Metarule 3	
Form: Function(X, Y) ⇒ Called by (X)	*Confidence: One*
Implication: The functions are called by one function only.	
Pattern category 2: Avoid duplicated functionality	
Pattern 4: Identify utilities	*Intent:* Identify utility routines so that they can be shared by functions.
Metarule 4	
Form: Function(X, Y) ⇒ Called by (X)	*Coverage: High*
Implication: Function(s) on LHS are called by most of the functions in the system.	
Pattern category 3: Re-modularise for maintainability	
Pattern 5: Increase data modularity	*Intent:* Place related data items into a structure
Metarule 5-1	
Form: Global(X, Y) ⇒ Global(X, Y)	*Confidence: High Support: High*
Implication: Whenever one global variable is accessed, there is a high probability that the other global variable is also accessed.	
Metarule 5-2	
Form: Type(X, Y) ⇒ Type (X, Y)	*Confidence: High Support: High*
Implication: Whenever one type variable is accessed, there is a high probability that the other type is also accessed.	
Pattern 6: Strengthen encapsulation	*Intent:* Identify potential classes.
Metarule 6	
Form: Accessed by(X) ⇒ Type(X, Y)	*Confidence: One*
Implication: Function(s) access the type(s) on the RHS.	

upon these objects. Metarules 5 and 6 address this issue (Table 3). Patterns and related metarules are briefly described in Section 4.4. Details are presented in the appendix.

4.4 Metarules and re-engineering patterns

According to our description of metarules in the introduction to Section 4, a metarule places a restriction on the form of association rules to be mined in addition to on interestingness measure values. Rule-based constraints are restrictions that may be placed, for example, on the relationships between predicates, or on the values that these predicates may assume. As an example, consider a metarule of the form

$$Global(X, Y) \Rightarrow Accessed\ by\ (X) \qquad \text{(M1)}$$

A more general form of this metarule is $P_1(X, Y) \Rightarrow P_2(X)$ where P_1 and P_2 are predicate variables that are instantiated to the attributes *Global* and *Accessed by* in (M1). Thus in (M1), a restriction is placed on the attribute to which a predicate is instantiated, in addition to on the relationship between the predicates. Variable X represents functions, and variable Y represents global variables used by the function. When association rules are mined using this template, global variables accessed by various functions in the software system are returned. To find association rules that are meaningful in the re-engineering context, an additional constraint is placed on interestingness measures. Thus the complete metarule (for details, refer to Table 3 and the Appendix) is represented as

$$Global\,(X, Y) \Rightarrow Accessed\ by\ (X) \quad Coverage : Low \quad \text{(M2)}$$

When association rules are mined using this template, we are able to identify a re-engineering opportunity i.e. global variables which are used by few functions within the system. In this paper, we use the convention as in (M2) to describe metarules. Predicate names (attributes) are self-explanatory.

Some metarules which are interesting in the re-engineering context, their implications, and related re-engineering patterns are shown in Table 3. In some of the metarules identified, one out of the three interestingness measures has been used. This indicates that the value of the other two measures does not influence the result. However, it is possible to identify metarules where a combination of measures gives interesting results, as is evident from Pattern 5.

We discuss benefits of employing the re-engineering patterns related to metarules, in addition to issues and problems in the Appendix. There are various popular forms for expressing patterns e.g. the Alexandrian form, the Gang of Four (GOF) form and the Coplien form [17]. To describe our re-engineering patterns, we adapt the pattern form used by Demeyer *et al.* [12]. Related patterns are grouped into categories to aid comprehension.

5 Experiments and results

For conducting metarule-guided association rule mining, source code of five open source software systems written in C was used. A brief description of these systems is provided in Table 4.

CVS, Aero, Mosaic and Bash have been used for architecture recovery experiments in [18] and Xfig has been used for architecture recovery experiments in [19]. Our data mining experiments are helpful in gaining understanding of the test systems by providing a more detailed view of their structure.

The source files for the five systems have been parsed using the Rigi tool and relevant ‘facts’ have been stored in an exchange format called the ‘Rigi Standard Format’ (RSF) [20, 21]. The transaction set discussed in Section 4.1 was developed from this fact set. In the following Sections, we present results of applying association rule mining to our test systems and analyse these results.

5.1 Pattern 1: Reduce global variable usage (Metarule 1 form: Global (X, Y) ⇒ Accessed by (X) Coverage: Low)

To evaluate the applicability of Pattern 1, the source code of the five systems was mined using Metarule 1. Table 5 summarises statistics obtained as a result of mining association rules using this metarule.

The metarule associated with pattern 1 can be used to derive useful information from the system when coverage is low. It can be seen from Table 5 that coverage values for the accessed global variables in all five systems remain below 0.2. Even in Mosaic, which has the least coverage value, the most frequently accessed global variable is accessed by 47 functions. The number of functions is higher in the other systems. If a global variable is accessed by 47 functions, there is reason to define it as global. It is obvious that a coverage value of 0.2 cannot be considered low in the above systems. This result illustrates why we do not recommend setting absolute thresholds. It also illustrates the difficulty in setting a threshold which can be used for all systems. Instead of trying to set an absolute threshold, we depict our results using a graph which can be inspected visually to identify unusual values that represent restructuring opportunities. To enable the graph to be interpreted interactively, median, quartiles, and upper and lower control limits (defined as mean $\pm$ 2 $\times$ standard deviation) may be depicted on the graph. As an example, the global variable usage graph for Bash, as mined by metarule 1 is depicted in Fig. 2.

It is evident from Fig. 2 that a substantial proportion of global variables are being accessed by very few functions in Bash. The graphs of the other four systems show a very similar global usage trend. It is interesting to note that the median number of functions accessing a global variable is almost the same (2 and 3) for all systems. The

Table 4: Test system characteristics

	CVS	Aero	Mosaic	Bash	Xfig
Description	Version control system	Rigid body simulator	Web browser	Unix shell	Drawing tool
Version	1.8	1.7	2.6	1.14.4	3.2.3
Lines of code (LOC)	30 k	31 k	37 k	38 k	75 k
Functions	810	686	818	892	1661
Global variables	419	535	348	539	1746
User defined types	375	814	112	198	828

Table 5: Global variables usage summary

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	1469	2628	858	1714	8280
Number of functions accessing global variables (% of total functions)	391 (48%)	422 (62%)	370 (45%)	545 (61%)	1329 (80%)
Median number of functions accessing a global variable	2	3	2	2	3
Average number of functions accessing a global variable	3.99	5.29	2.93	4.12	5.97
Standard deviation	5.96	10.31	4.32	5.70	15.73
Most frequently accessed global variable	Noexec	aktuelleWelt	current_win	rl_point	XtStrings
Coverage of the most frequently accessed global variable (number of functions)	0.068 (55)	0.152 (104)	0.057 (47)	0.07 (62)	0.139 (231)

average number of functions accessing a global variable is also similar. Table 6 depicts the breakdown for global variable access in all five test systems, and Fig. 3 shows the breakdown specifically for Bash.

Metarule-guided rule mining has thus successfully identified a re-engineering opportunity. To improve modularity and hence software quality, it will be useful to examine global variables being accessed by very few functions in order to ascertain whether there is a need to define these variables globally. As an example, consider the variable `noninc_history_pos` used in just two functions `noninc_search` and `noninc_dosearch` in Bash. The only valid reason for defining this variable as global is that its value is to be retained throughout the duration of the program. Even in such a case, the variable should be defined as static to restrict its scope. An examination of the code shows that the variable has indeed been defined as static. Alternatively, the `rl_kill_ring_length` variable, also used by just two functions `rl_yank_pop` and `rl_kill_text` within `readline.c` is not a static variable. If the variable value is not required to persist throughout the lifetime of the program, it should be passed as a parameter within the relevant functions instead of being defined globally. Even if a global variable is required, its scope should be restricted according to usage to reduce coupling.

5.2 Pattern 2: Localise variables (Metarule 2 form: Global(X, Y) ⇒ Accessed by(X) Confidence: One)

To evaluate the applicability of Pattern 2, the source code of the five systems was mined using Metarule 2. Table 7 summarises statistics obtained as a result of mining association rules using this metarule.

It is interesting to note that in all systems, quite a large percentage of global variables is being used by only one function. The fact that these variables have been defined as global seems to indicate a poor design or a design that has deteriorated over time. However, the definition of these variables as global is justified if they represent settings that are required to persist throughout a program's lifetime. For example, a detailed look at Bash reveals that global variables being accessed by single functions do indeed represent system settings. The `msg_buf` variable which represents a buffer for messages is used by one function only (`rl_message`). If this buffer is accessed more than once during the execution of the program, there is reason for it to be global. The fact that this variable has been defined as static shows an effort by developers to reduce variable scope. However, opportunities for reduction in variable scope within the system may be identified. For example, the globally defined variable `special_vars`

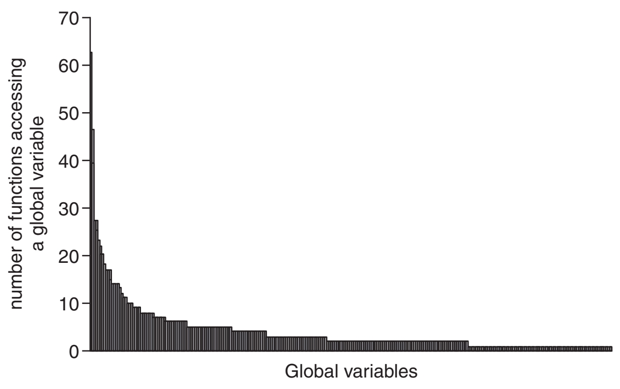

Fig. 2 *Number of functions accessing a global variable*

Table 6: Global variable access breakdown in test systems

	CVS	Aero	Mosaic	Bash	Xfig
Percentage of global variables accessed by more than 5 functions	13%	13%	9%	14%	18%
Percentage of global variables accessed by 5 or less functions	87%	87%	91%	86%	82%

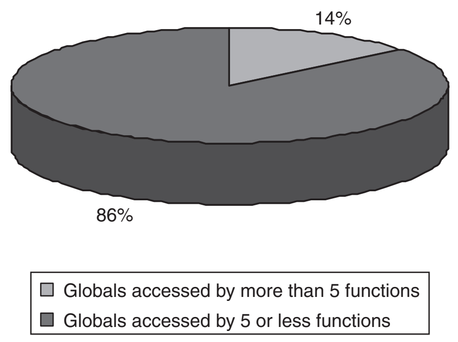

Fig. 3 *Global variable access breakdown*

is also used by only one function but has not been defined as static.

5.3 *Pattern 3: Increase locality of reference (Metarule 3 form: Function(X, Y) ⇒ Called by (X) Confidence: One)*

To evaluate the applicability of Pattern 3, the source code of the five systems was mined using Metarule 3. Table 8 summarises statistics obtained as a result of mining association rules using this metarule.

It can be seen from Table 8 that most of the functions called by a single function reside in the same file as the calling function in all systems. Bash and Xfig have higher percentages of functions residing in same files. Functions residing in a different file from the calling function need to be examined in more detail. A valid reason for placing a called function in a different file from the calling function is that the called function may be a 'utility' function which has been placed together with other utility functions in a separate file. For example, in Bash the function `all_visible_variables` present in the file `variables.c` is called by the function `variable_completion_function` in `bashline.c`. `variables.c` contains functions related to shell variables. Functions within `bashline.c` are related to reading lines of input. In this case, there is valid reason for placing `all_visible_variables` in the file variables.c, along with other functions related to shell variables. If this is not the case i.e. the called function appears to be logically related to the calling function, it should be placed in the same file as the calling function to increase efficiency and ease maintenance by increasing locality of reference.

Table 7: Global variables accessed by one function

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	77	72	104	115	310
Percentage of global variables accessed by one function	18%	13%	30%	21%	18%

5.4 *Pattern 4: Identify utilities (Metarule 4 form: Function(X, Y) ⇒ Called by(X) Coverage: High)*

To evaluate the applicability of Pattern 4, the source code of the five systems was mined using Metarule 4. Table 9 summarises statistics obtained as a result of mining association rules using this metarule.

Table 8: Functions called by one function

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	236	250	321	298	368
Percentage of functions called by one function	29%	36%	39%	33%	22%
Percentage of functions residing in same file as calling function	54% (128)	65% (163)	56% (181)	74% (220)	73% (270)
Percentage of functions residing in a different file	46% (108)	35% (87)	44% (140)	26% (78)	27% (98)

Table 9: Function call summary

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	3899	1708	1472	2075	4798
Number of functions making function calls (% of total functions)	631 (78%)	469 (68%)	518 (63%)	694 (78%)	1302 (78%)
Median number of calls made	2	1	1	2	2
Average number of calls made	6.73	3.80	2.57	3.18	4.28
Standard deviation	17.64	7.79	4.77	8.46	8.45
Number of functions to which 5 or more calls are made (% of total functions)	148 (18%)	75 (11%)	55 (7%)	98 (11%)	239 (14%)
Number of functions to which 10 or more calls are made (% of total functions)	84 (10%)	39 (6%)	22 (3%)	29 (3%)	101 (6%)
Most frequently called function	error	fprintf	mo_fetch_window_by_id	xmalloc	Put_msg
Coverage of the most frequently called function (number of functions)	0.259 (210)	0.138 (95)	0.072 (59)	0.219 (195)	0.08 (133)

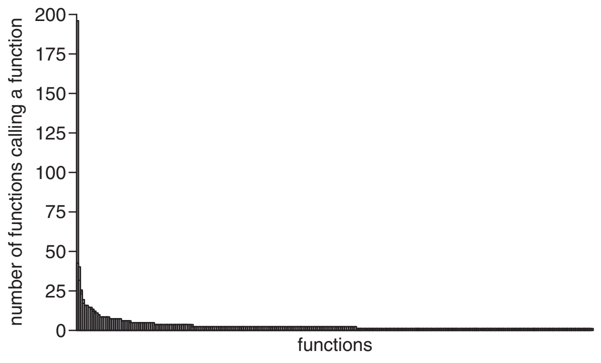

Fig. 4 *Number of function calls made to functions*

The metarule associated with Pattern 4 can be used to derive useful information from the system when coverage is high. It can be seen from Table 9 that coverage values for the called functions in the five systems remain below 0.3. In CVS, which has the highest coverage, the most frequently called function is called by 210 functions. If a function is called by 210 functions, there is reason to assume that the function is some kind of a utility function. Similar to Pattern 1, setting a threshold is difficult because with a high absolute threshold of say 0.7, none of the functions would be considered as frequently accessed.

Instead of trying to set an absolute threshold, we depict the results using a graph which can be inspected interactively to identify frequently called functions. To make the interpretation of the graph easier, median, quartiles, and upper and lower control limits may be depicted on the graph. As an example, the function call graph for Bash is depicted in Fig. 4. It is easy to identify functions called by a large number of functions from Fig. 4. The graphs of the other four systems show a very similar function call trend. It is interesting to note that the median number of functions calling a function is almost the same (1 and 2) for all systems. The average number of functions calling a function is also similar.

An examination of the Bash code shows that 75% of the functions to which 20 or more calls are made, reside in the files `general.c`, `error.c`, or `variable.c`. This indicates that utility functions are placed in separate files in Bash, making it easier to reference such functions.

5.5 Pattern 5: Increase data modularity (Metarule 5-1 form: Global(X, Y) ⇒ Global(X, Y) Confidence: High Support: High, Metarule 5-2 form: Type(X, Y) ⇒ Type(X, Y) Confidence: High Support: High)

To evaluate the applicability of Pattern 5, the source code of the five systems was mined using Metarule 5-1 and Metarule 5-2. In this case, we use a threshold of 0.7 for confidence. It should be noted that setting absolute thresholds for coverage and support may not be meaningful (see Section 4.2), since they depend on system size. However, confidence denotes the association of two items with each other, and is not affected by size of the system. Thus it is meaningful to set a threshold e.g. 0.7, which denotes that if a global variable (user defined type) is accessed, there is a 70% probability that the other global variable (user defined type) is also accessed. However, the confidence measure alone cannot be used to arrive at a meaningful result. A global variable may be associated with more than one global variable. Similarly a user defined type may be associated with more than one user defined type. It is thus difficult to decide which global variables or user defined types to place in one structure. In such a case, the support of the association rule may be helpful. Support denotes the number of functions which access the two global variables or user defined types together. If two global variables or user defined types are accessed together by many functions, it is more useful to place them in a single structure, rather than global variables or user defined types that are accessed together by few functions.

Table 10 summarises statistics about global variables with high association. The names of the global variables in all five systems reveal that they are related to one another. Through the use of association rule mining, such variables may be identified and placed in a structure after inspection. Similar to Table 10, Table 11 summarises statistics about user defined types with high association. For CVS, Aero, Mosaic and Bash, accesses to user defined types are reported for local variables within functions.

5.6 Pattern 6: Strengthen encapsulation (Metarule 6 form: Accessed by(X) ⇒ Type(X,Y) Confidence: One)

To evaluate the applicability of Pattern 6, the source code of the systems was mined using Metarule 6. Table 12 summarises statistics obtained as a result of mining association rules using this metarule.

Identification of potential objects is facilitated by mining association rules of the described form. As an example, consider some of the user defined types identified within Bash. The `user_info` type is accessed by 10 functions which reside in seven different files. This type can be grouped together with the functions that access it to form a class, which represents user information and member functions to access this information. Some of the types are accessed by functions within one file, making the task of conversion to an object-oriented design easier. For example, the `JOB` type is accessed by 22 functions (`start_job`, `delete_job`, `find_job` etc.), 21 of

Table 10: Global variables with high association

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	1923	5199	2602	1635	14983
Highest support for global variables occurring together with high confidence (number of functions)	0.022 (18)	0.099 (68)	0.007(6)	0.047 (42)	0.127 (211)
Global variables occurring together with highest support	Pending_error, pending_error_text	KoerperKoordAnAus, KoordAnAus	size_of_cached_cd_array, cached_cd_array	rl_end, rl_point	_ArgCount, _ArgCount

Table 11: User defined types with high association

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	41	54	10	14	422
Highest support for user defined types occurring together with high confidence (number of functions)	0.010 (8)	0.017 (12)	0.010 (8)	0.003 (3)	0.128 (213)
User defined types occurring together with highest support	DIR, dirent	TKollision, TReal	mo_hotlist, mo_hot_item	KEYMAP_ENTRY_ARRAY, keymap	Arg, WidgetList

Table 12: Types accessed by functions in the test systems

	CVS	Aero	Mosaic	Bash	Xfig
Number of association rules satisfying the metarule	489	476	294	319	3937
Number of functions accessing user defined types (% of total functions)	299 (37%)	279 (41%)	248 (30%)	249 (28%)	1364 (82%)
Number of types accessed	80	63	47	51	149
Median number of functions accessing a type	2.5	3	3	3	5
Average number of functions accessing a type	6.11	7.56	6.26	6.25	26.42
Standard deviation	10.74	14.56	12.56	9.36	53.92

which reside in the file `job.c`. In a similar manner, types and accessing functions may also be identified easily within other systems.

5.7 Discussion of results

Statistics obtained as a result of carrying out association rule mining on the test legacy systems are summarised in Figs. 5–12. Statistics presented in Figs. 5, 6 are related to the number of functions, global variables and user defined types in the test systems. It can be seen that the number of functions and global variables is similar across all systems except Xfig. The number of functions making function calls, accessing global variables and user defined types are also comparable across these systems. The number of functions accessing user defined types is larger for Xfig as

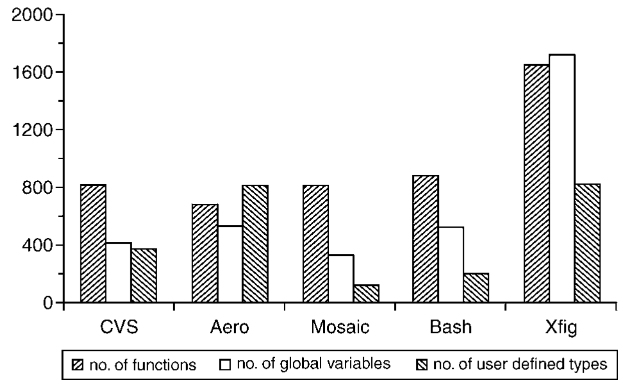

Fig. 5 *Function, global variable and user defined type statistics*

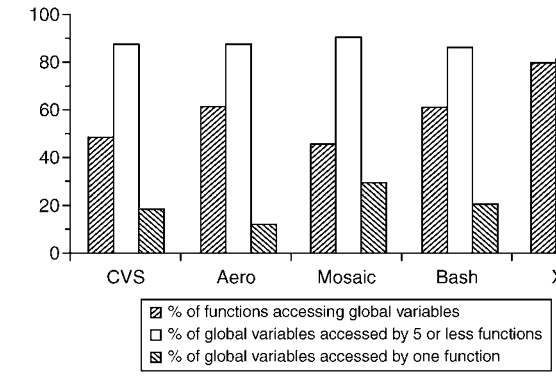

Fig. 7 *Global variable access statistics*

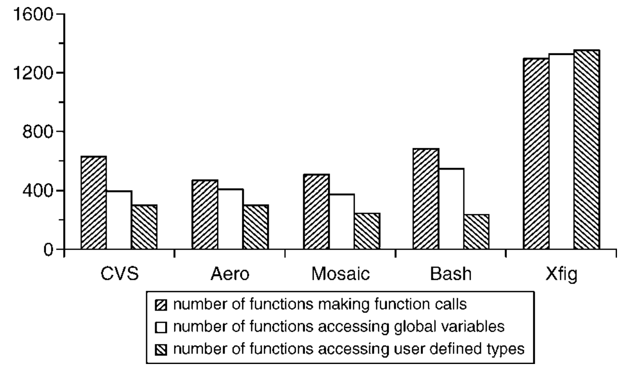

Fig. 6 *Statistics for function calls, global variable accesses and user defined type accesses*

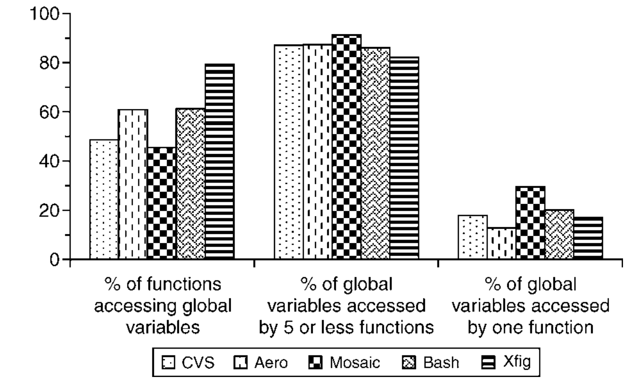

Fig. 8 *Comparison of accesses to global variables*

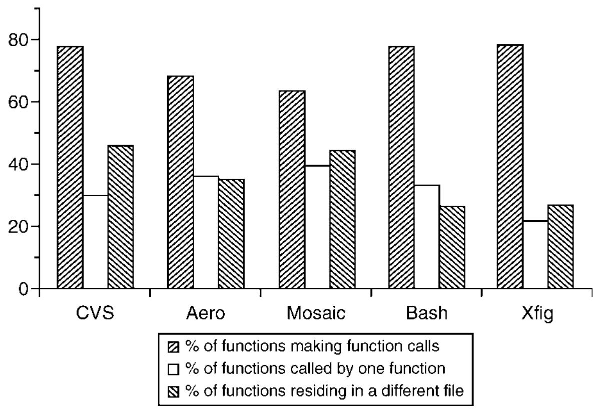

Fig. 9 *Function call statistics*

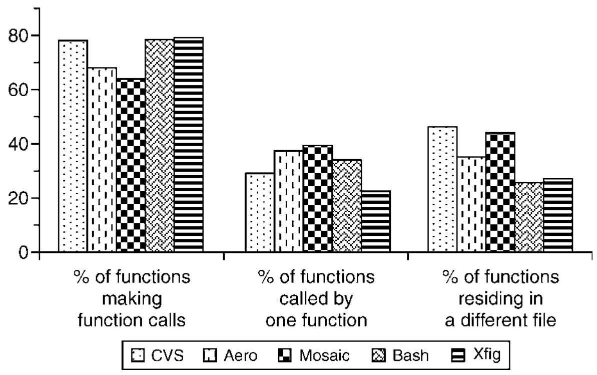

Fig. 10 *Comparison of calls to functions*

compared to other systems, and also larger than the number of functions accessing global variables and making function calls.

Figures 7 and 8 present statistics related to the access of global variables by functions within test systems, which is an indicator of the global coupling present in these systems. It is interesting to note that although the percentage of functions accessing global variables varies from 45% to 80%, the percentage of global variables accessed by 5 or less functions is very similar across all systems. Mosaic has the smallest percentage of functions accessing global variables, and also the highest percentage of global variables accessed by five or less functions. These results show that all five systems present opportunities for applying Pattern 1 and Pattern 2 to reduce coupling by reducing the number of global variables. In cases where a global variable is required, its scope may be reduced to make the program easier to understand and manage.

Figures 9 and 10 present statistics related to functions called by a single function within the systems. The percentage of functions making function calls is similar across all systems, varying from 63 to 78%. The percentage of functions called by one function only is also similar across all systems, varying from 22 to almost 40% in Mosaic. Thus the opportunity to apply Pattern 3 to increase locality of reference is present in all systems. Functions that are called by a single function and yet reside in a different file need to be examined. CVS and Mosaic have the highest percentages of such functions, which should be examined to determine the feasibility of placing them in the same file as the calling function.

Figures 11 and 12 present statistics of functions within the systems to which five or more calls are made. This percentage varies from 7 to 18%, in general remaining low.

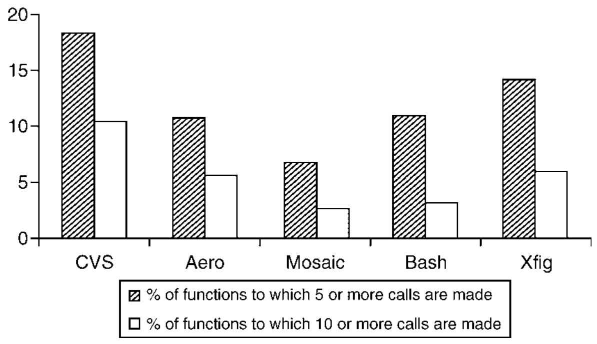

Fig. 11 *Frequently accessed functions*

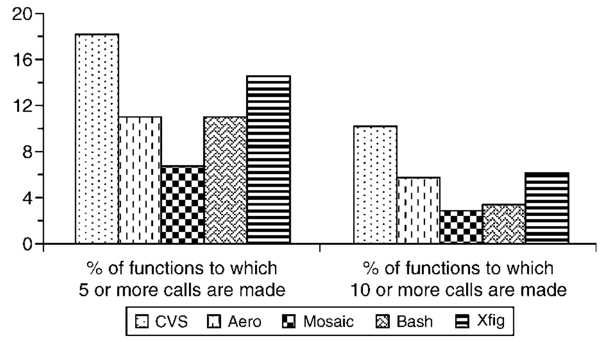

Fig. 12 *Comparison of frequently accessed functions*

CVS has the highest percentage of functions to which five or more calls are made. Thus in all systems, there exist functions that are accessed by many functions. It is useful to examine these functions and identify them as utilities, also placing them in appropriate files.

Tables 10–12 summarise statistics related to system remodularisation. The applicability of Pattern 5 is evident for all the systems, as global variables and user defined types that occur together have been identified. It is suggested that Pattern 5 be applied before Pattern 6, so that related data items are placed together. Once data has been modularised, functions accessing these types may be mined using Pattern 6, thus taking a step towards indicating potential classes in an object-oriented design.

6 Related work

Based on studies of the IBM programming process and the OS/360 operating system, Lehman proposed laws that guide the evolution of E-type software systems [22]. A study of the evolution of the Logica plc Fastwire (FW) financial transaction system, reported in 1997, supports most of the laws formulated earlier [23]. Both these studies focus on system growth in terms of the number of modules within the system in order to determine trends in software evolution. More recently, Godfrey and Tu carried out studies to understand the evolution of open source software [24, 25]. An interesting observation is that the growth rate of most open source software systems is super-linear, violating Lehman's law of software evolution which suggests that the growth of systems slows down as their size increases. Godfrey and Tu also present an approach for studying architectural evolution of software systems [26], which integrates visualisation and software metrics. Evolution status of various entities is modelled by determining which entities are new, have been deleted, or remain unchanged. Unlike these studies by Lehman and Godfrey, which model structural changes at the entity level, our selection of items in the form of functions, global variables and user defined types will enable the study of characteristics of legacy system evolution at a more detailed level.

Visualisation can also be used to gain an understanding of a legacy system's overall structure and its evolution. Lanza *et al.* [27] propose a lightweight approach to understanding object-oriented legacy systems using a combination of software metrics and visualisation. They visualise various aspects of the system e.g. a system hotspot view helps in identifying very large and small classes, a system complexity view is based on inheritance hierarchies. These views are helpful in indicating problematic areas which may require a deeper study. The evolution of legacy object-oriented systems can also be studied using the same lightweight approach [28].

There has been growing interest in the application of data mining techniques to gain better understanding of software systems. In recent years, researchers have applied various data mining techniques including association rule mining, cluster analysis and concept learning in different contexts. The architecture recovery of software systems using data mining techniques is discussed in [29–32]. The data mining technique employed in these papers is primarily association rule mining. The identification of sub-systems based on associations (ISA) was proposed by Oca and Carver [29]. They use association rule mining for extraction of data cohesive sub-systems by grouping together programs that use the same data files. A very similar approach is that of [30], where the use of a representation model (RM) is discussed to represent the sub-systems identified using the ISA methodology in [29]. Sartipi *et al.* [31] discuss a technique for recovering the structural view of a legacy system's architecture based on association rule mining, clustering and matches with architectural plans. Tjortjis *et al.* [32] also employ association rule mining to arrive at decompositions of a system at the function level. Groups of functions, i.e. sub-systems, are created by finding common attributes participating in the same association rules.

Clustering techniques have also been employed for architecture recovery and software re-modularisation [19, 33–37]. Tzerpos and Holt [33] present the case for using clustering techniques to re-modularise software, after the techniques have been adapted to fit the peculiarities of the software domain. In [34], Wiggerts provides a framework to apply cluster analysis for re-modularisation. Experiments with clustering as a re-modularisation method are described in [35] and [36]. Both papers conclude by recommending similarity measures and clustering algorithms which yield good experimental results for software artifacts. A theoretical explanation to some previous experimental results obtained by researchers in the area is provided in [37]. The paper also describes a new clustering algorithm, which gives better results as compared to the currently employed algorithms for clustering software. In [19], the weighted combined algorithm for software clustering is presented. This algorithm shows improvement in clustering results as compared to previously employed clustering algorithms.

The use of concept learning to support software system maintenance is described in [10, 38–39]. Inductive techniques are employed to extract a maintenance relevance relation (MRR) from the source code, maintenance logs and historical maintenance update records. An MRR simply indicates that if a software engineer needs to understand file1, he/she probably also needs to understand file2. The problem has been presented as a concept learning problem, and a decision tree classifier is used for classifying file pairs as relevant, not relevant and potentially relevant. Relevance indicates that two files were modified in the same update and potential relevance indicates that both files were looked at in the same update. Program comprehension can be also aided by source code mining [40, 41].

The present work employs association rule mining to find interesting associations between global variables, user defined types and function calls within a system and relates them to re-engineering patterns. Although association rule mining has been employed by other researchers e.g. Sartipi *et al.* [31] and Tjortjis *et al.* [32], their purpose and technique is different from the one proposed in this paper. The purpose of association rule mining in the case of Sartitipi *et al.* is architectural design recovery i.e. decomposition of the legacy system into modules, where a module is a collection of functions, data types and variables. The item set used by them consists of global variables, data types and function calls within the system. The transaction set consists of global variables, data types accessed and functions called by a function. The Apriori algorithm is used to generate frequent item sets, which indicate interesting association between functions. Functions that are associated are candidates for being placed in the same module. The architectural query language (AQL) is used to describe a system's conceptual architecture. Based on this architecture and the frequent item sets, clustering and a branch and bound algorithm are used to instantiate the concrete architecture. Although the item and transaction sets employed in [31] are similar to the ones we employ in this paper, it is evident that the problem addressed is very different. Whereas Sartipi *et al.* use association rules to indicate association between functions to discover modules, in this paper, our purpose in applying association rule mining is to gain program understanding and indicate problem areas where improvements may be made. To achieve this purpose we find associations not only between functions, but between functions and other items as well e.g. between global variables and user defined types.

Tjortjis *et al.* also employ association rule mining for grouping together similar entities within a software system. The item set used by them consists of variables, data types and calls to blocks of code (modules), where modules may be functions, procedures or classes. The transaction set thus consists of variables, types accessed and calls made by modules. In their algorithm, large item sets are first generated by finding item sets that have a higher support than a user-defined threshold. From this item set, association rules with confidence greater than a user-defined threshold are generated. Finally groups of modules are created based on the number of common association rules. Thus in their case, similar to the rule mining approach in [31], rules are mined to group modules. We, however, use metarule-guided association rule mining to find associations between items, which can be used to identify potential problems in a system. By relating our mined association rules to re-engineering patterns, we find solutions to commonly occurring problems. To the best of our knowledge, research has not been conducted to relate association rule mining to re-engineering patterns as is done in this paper.

7 Conclusions

In this paper we have presented an association rule mining approach for the problem of understanding a software system given only the source code. To make the association rule mining process more effective, we employ metarule-guided mining. Metarules specify constraints on the mining process. We use two types of constraints: rule constraints and interestingness constraints. Rule-based constraints are used to restrict the form of the association rule mined whereas interestingness constraints are used to restrict the possible values of interestingness measures. Furthermore, we relate each metarule to a re-engineering pattern that presents solutions to the problems identified by a metarule. To illustrate the feasibility and effectiveness of employing metarule-guided mining for program understanding, we restrict ourselves to a small number of metarules and re-engineering patterns that address some typical problems within legacy systems developed using the structured approach.

Metarule-guided mining was used to analyse the structure of five legacy systems and extract meaningful association rules which provided useful insight about the software's overall structure and suggested strategies for its improvement. The extracted association rules capture associations between functions, global variables and user defined types within the system. They highlight potential problems within the code and identify areas which require a more detailed study. As illustrated, these association rules applied along with re-engineering patterns can be used to restructure the code for maintainability, and if required, to re-modularise the code, e.g. by converting a structured design to an object-oriented design. A manual inspection to carry out the same tasks would have taken a much longer time.

Our experiments with five open source systems reveal similar results in terms of the average and percentage values in the patterns discussed. For example, the average and median number of functions accessing a global variable is very similar in all five systems, as is also the average and median number of function calls made. This result is interesting, especially since the test systems have different application domains and the number of functions, global variables and user defined types within these systems is also different. Our experimental results show how rules extracted by the mining process can be used to identify potential problems in, and suggest improvements to, the legacy systems under study. Thus metarule-guided association rule mining is effective in revealing interesting characteristics, trends and nature of open source legacy systems, and can be used as a basis for restructuring or re-modularising them for greater maintainability.

8 Acknowledgments

The authors wish to thank Rainer Koschke of the Bauhaus group at the University of Stuttgart for providing the RSF for Bash, Aero, Mosaic and CVS, and to Johannes Martin of the Rigi group, University of Victoria, for providing the RSF for Xfig.

9 References

1 Sommerville, I.: 'Software engineering' (Addison Wesley, 2000, 5th edn.)
2 Arnold, R.S.: 'Software reengineering' (IEEE Computer Society Press, 1993)
3 Pressman, R.S.: 'Software engineering a practitioner's approach' (McGraw-Hill, 2001, 5th edn.)
4 Pfleeger, S.L.: 'Software engineering theory and practice' (Prentice-Hall, 1998)
5 Glass, R.L.: 'Frequently forgotten fundamental facts about software engineering', *IEEE Softw.*, 2001, **18**, pp. 110–112
6 Biggerstaff, T.J.: 'Design recovery for maintenance and reuse', *Computer*, 1989, **22**, pp. 36–49
7 Muller, H.A., Story, M., Jahnke, J.H., Smith, D.B., Tilley, A.R., and Wong, K.: 'Reverse engineering: a roadmap'. 22nd Int. Conf. on Software Engineering (ICSE), June 2000
8 Parikh, G., and Zvegintzov, N.: 'Tutorial on software maintenance' (IEEE Computer Society Press, 1983)
9 Hall, R.P.: 'Seven ways to cut software maintenance costs', *Datamation*, 1987, **33**, pp. 81–84
10 Shirabad, J.S., Lethbridge, T.C., and Matwin, S.: 'Supporting software maintenance by mining software update records'. Int. Conf. on Software Maintenance (ICSM), 2001
11 Han, J., and Kamber, M.: 'Data mining: concepts and techniques' (Morgan Kaufmann, 2000)
12 Demeyer, S., Ducasse, S., and Nierstrasz, O.: 'Object-oriented reengineering patterns' (Morgan Kaufmann, 2003)
13 http://www.iam.unibe.ch/~scg/Archive/famoos/
14 Mens, T., and Tourwe, T.: 'A survey of software refactoring', *IEEE Trans. Softw. Eng.*, 2004, **30**, pp. 126–139
15 Demeyer, S., and Ducasse, S.: 'Metrics, do they really help?'. Proc. LMO'99, Paris, 1999
16 Fenton, N., and Pfleeger, S.L.: 'Software metrics: a rigorous and practical approach' (PWS Publishing Company, 1997, 2nd edn.)
17 Rising, L.: 'The patterns handbook techniques, strategies and applications' (Cambridge University Press, 1998)
18 Koschke, R.: 'Atomic architectural component recovery for program understanding and evolution', PhD Thesis, University of Stuttgart, 2000
19 Maqbool, O., and Babri, H.A.: 'The weighted combined algorithm: a linkage algorithm for software clustering'. Conf. on Software Maintenance and Re-engineering (CSMR), March 2004
20 http://www.bauhaus-stuttgart.de/bauhaus
21 Martin, J., Wong, K., Winter, B., and Müller, H.A.: 'Analyzing xfig using the Rigi tool suite'. Seventh Working Conf. on Reverse Engineering (WCRE), Brisbane, Australia, 2000
22 Lehman, M.M.: 'Program, life cycles and the laws of software evolution', *Proc. IEEE*, 1980, **68**, pp. 1060–1076
23 Lehman, M.M., Ramil, J.F., Wernik, P.D., Perry, D.E., and Turski, W.M.: 'Metrics and laws of software evolution – the Ninetiesview'. Int. Symp. on Software Metrics, 1997
24 Godfrey, M.W., and Tu, Q.: 'Evolution in open source software: a case study'. Int. Conf. on Software Maintenance (ICSM), 2000
25 Godfrey, M.W., and Tu, Q.: 'Growth, evolution and structural change in open source software'. IWPSE 2001
26 Tu, Q., and Godfrey, M.W.: 'An integrated approach for studying architectural evolution'. Int. Workshop on Program Comprehension (IWPC), 2002
27 Lanza, M., and Ducasse, S.: 'Polymetric views – a lightweight visual approach to reverse engineering', *IEEE Trans. Softw. Eng.*, 2003, **29**, pp. 782–795
28 Lanza, M., and Ducasse, S.: 'Understanding software evolution using a combination of software visualization and software metrics'. Proc. LMO'02, 2002
29 Montes de Oca, C., and Carver, D.L.: 'Identification of data cohesive subsystems using data mining techniques'. Int. Conf. on Software Maintenance (ICSM), November 1998
30 Montes de Oca, C., and Carver, D.L.: 'A visual representation model for software subsystem decomposition'. Working Conf. on Reverse Engineering (WCRE), October 1998
31 Sartipi, K., Kontogiannis, K., and Mavaddat, F.: 'Architectural design recovery using data mining techniques'. Conf. on Software Maintenance and Reengineering (CSMR), February 2000
32 Tjortjis, C., Sinos, L., and Layzell, P.: 'Facilitating program comprehension by mining association rules from source code', 11th IEEE Int. Workshop on Program Comprehension (IWPC), May 2003
33 Tzerpos, V., and Holt, R.C.: 'Software botryology: automatic clustering of software systems'. Ninth Int. Workshop on Database and Expert Systems Applications (DEXA), August 1998
34 Wiggerts, T.A.: 'Using clustering algorithms in legacy systems remodularization'. Fourth Working Conf. on Reverse Engineering (WCRE), October 1997
35 Anquetil, N., and Lethbridge, T.C.: 'Experiments with clustering as a software remodularization method'. Sixth Working Conf. on Reverse Engineering (WCRE), 1999
36 Davey, J., and Burd, E.: 'Evaluating the suitability of data clustering for software remodularization'. Seventh Working Conf. on Reverse Engineering (WCRE), Brisbane, Australia, 2000
37 Saeed, M., Maqbool, O., Babri, H.A., Sarwar, S.M., and Hassan, S.Z.: 'Software clustering techniques and the use of the combined algorithm'. Conf. on Software Maintenance and Re-engineering (CSMR), March 2003
38 Shirabad, J.S., Lethbridge, T.C., and Matwin, S.: 'Mining the maintenance history of a legacy software system'. Int. Conf. on Software Maintenance (ICSM), 2003

39 Shirabad, J.S., Lethbridge, T.C., and Matwin, S.: 'Mining the software change repository of a legacy telephony system'. Proceedings of the 1st International Workshop on Mining Software Repositories, 2004
40 Balanyi, Z., and Ferenc, R.: 'Mining design patterns from C++ source code'. Int. Conf. on Software Maintenance (ICSM), 2003
41 Kanellopoulos, Y., and Tjortjis, C.: 'Data mining source code to facilitate program comprehension: experiments on clustering data retrieved from C++ programs'. Int. Workshop on Program Comprehension (IWPC), 2004
42 Murray, R.B.: 'C++ strategies and tactics' (Addison Wesley, 1993)
43 McConnell, S.: 'Code complete a practical handbook of software construction' (Microsoft Press, 1993)

10 Appendix

10.1 Pattern category 1: Enhance modularity and control side effects

10.1.1 Pattern 1: Reduce global variable usage:

Intent: Remove global variables that are used by very few functions.
Metarule 1: *Global* $(X, Y) \Rightarrow$ *Accessed by* (X)
Coverage: *Low*

Implication: Only a small proportion of functions in the system access the global variable(s) on the LHS.
Problem: As software evolves, its structure becomes complex and its internal quality degrades. Owing to high coupling between system components in such a deteriorated structure, a single change often results in a number of side effects. Functions which access the same global variables are highly coupled (common coupling). How can such coupling among functions be reduced?
Forces: This problem is difficult because

1. To improve system structure, the structure must first be understood. In the absence of documentation, gaining a structural view and identifying dependencies may be difficult and time consuming.
2. Some form of coupling may be necessary within the system. Highly coupled functions whose coupling levels may be reduced need to be identified.
3. A change e.g. changing a shared global variable to a parameter that is passed between the related functions may require changes at a number of places within the code.

Solving this problem is feasible because

1. With the proper tools, some forms of coupling between functions can be identified.

Solution: Identify global variables which are used by few functions within the system. Reduce common coupling by reducing the use of such global variables. Rather than defining such variables as global, pass them as parameters within the relevant functions. If passing a global variable as a parameter is not convenient, its scope may be restricted to a single file by defining it as static.
Trade-offs:
Pros: When a large number of variables are to be shared amongst functions, global variables are convenient. Moreover, global variables have a longer lifetime than automatic variables, making it simpler to share information between functions that do not call each other. However, unless the global data is read only, the use of global variables results in undesirable coupling between functions, leading to difficulties in program understanding and maintenance. By removing unnecessary global variables and restricting their usage, coupling among components is reduced, making it easier to trace faults and avoid unintentional changes to data.
Cons and difficulties: Careful evaluation of each global variable is required to decide whether it should be passed as a parameter, declared to be static or left as it is. Even if very few functions in the system access the global variable, the designers of the system may have valid reasons for defining it as global.

10.1.2 Pattern 2: Localise variables:

Intent: Reduce the scope of variables that have larger scope than necessary.
Metarule 2: *Global*$(X, Y) \Rightarrow$ *Accessed by* (X)
Confidence: *One*

Implication: Each global variable on LHS is used by one function only.
Problem: Legacy systems are likely to contain a large number of global variables which reduces the manageability of the systems. How can a function accessing global variables be made easier to read and manage?
Forces: This problem is difficult because

1. A function may access a large number of global variables.
2. Global variables are not the only factor that make a function more difficult to read and understand.

Solving this problem is feasible because

1. You have an idea of the global variable usage within the system (Pattern 1)

Solution: Identify global variables that are accessed by one function only. Make the variable a local variable for that function.
Trade-offs:
Pros: If a variable is used by one function, there may be no reason for defining it as global. Localising a global variable reduces chances of error and makes functions easier to read and maintain.
Cons and difficulties: Careful evaluation of each identified global variable is required before a decision to reduce its scope is taken. The rationale for a decision by a developer to define a variable as global is rarely documented.

10.1.3 Pattern 3: Increase locality of reference:

Intent: Group related functions together to improve understandability and maintainability.
Metarule 3: *Function*$(X, Y) \Rightarrow$ *Called by* (X)
Confidence: *One*

Implication: Each function on LHS is called by one function only.
Problem: A legacy system is often large and may consist of a large number of functions. Changes made within a software system may involve changes to various related functions. How can changes be localised?
Forces: This problem is difficult because

1. It is necessary to identify functions that may be affected by a change.
2. A function may be related to a number of other functions, so it is difficult to decide the set of functions with which its relation is strongest.

Solving this problem is feasible because

1. Localised changes ease the maintenance task.

Solution: Identify functions between which there appears to be a strong relation. Place such functions in close proximity,

perhaps in the same file. It may be possible to inline functions being called by only one function.
Trade-offs:
Pros: Grouping together related functions promotes modular continuity [3] because changes in requirements are localised instead of resulting in system wise changes. Localised changes ease the maintenance task. Also, in case a function is inlined, making a function inline results in performance improvement and is clearly beneficial if the function expansion is shorter than the code for the calling sequence.
Cons and difficulties: Placing related functions in the same file may be useful for legacy applications that have been developed using the structured approach. It may not be a feasible option in some cases, e.g. when a system has a layered architecture and function calls are made across different layers, or in object-oriented architectures where the functional separation is related to data.

In case the inlining option is chosen, it is important to remember that large inline functions will save a small percentage of run time but will have a higher space penalty. Also functions with loops should almost never be inlined [42] because the run time of a loop is likely to swamp the function call overhead. Large inlined functions may also make it difficult to understand the functionality of the calling function.

10.2 Pattern category 2: Avoid duplicated functionality

10.2.1 Pattern 4: Identify utilities:

Intent: Identify utility routines so that they can be shared by functions.
Metarule 4: *Function*$(X, Y) \Rightarrow$ *Called by* (X)
Coverage: *High*

Implication: Function(s) on LHS are called by most of the functions in the system.
Problem: In legacy systems, functionality is often duplicated due to different teams re-implementing similar functionality. To avoid this problem, functions should share code by making use of utility routines. How can the use of utility routines be facilitated?
Forces: This problem is difficult because

1. To facilitate the use of utilities, the utilities must be identified.

2. Many such utility routines may be present, all of which need to be catalogued for reference.

Solving this problem is feasible because

1. Utility routines may have certain characteristics which facilitate identification.

Solution: Identify functions that are called by many functions within the system. Treat these functions as utility functions. It may be useful to place groups of related utility functions in separate files.
Trade-offs:
Pros: Utility functions represent re-usable components of a structured system. Re-usable components can lead to measurable benefits in terms of reduction in development cycle time and project cost, and increase in productivity.
Cons and difficulties: In order for functions to be re-used effectively, they must be properly catalogued for easy reference, standardised for easy application and validated for easy integration [3]. In case the number of such functions is large, related functions need to be identified and grouped together, otherwise searching for the appropriate function may be time consuming.

10.3 Pattern category 3: Re-modularise for maintainability

10.3.1 Pattern 5: Increase data modularity:

Intent: Place related data items into a structure.
Metarule 5-1: *Global*$(X, Y) \Rightarrow$ *Global*(X, Y)
Confidence: *High*
Support: *High*

Implication: Whenever one global variable is accessed, there is a high probability that the other global variable is also accessed.

Metarule 5-2: *Type*$(X, Y) \Rightarrow$ *Type*(X, Y)
Confidence: *High*
Support: *High*

Implication: Whenever one type is accessed, there is a high probability that the other type is also accessed.
Problem: Legacy systems are likely to contain a number of global variables and user defined types. How can relations between global variables (user defined types) be clarified? How can the global variables (user defined types) be more easily managed?
Forces: This problem is difficult because

1. A large number of global variables and user defined types may exist in a system, and examining them may be time consuming.

2. A global variable (user defined type) may be related to many other global variables (user defined types). Examining such complex relationships and arriving at meaningful conclusions may be very difficult

Solving this problem is feasible because

1. The purpose of a function is much easier to understand if relationships between data are clear and data is well managed.

Solution: Identify related global variables (user defined types). Examine them to see which of them form coherent entities. If they are logically related, combine them into a structure.
Trade-offs:
Pros: Combining variables/types into structures leads to code that is easier to understand and change. If at some stage, a shift is to be made to an object-oriented design paradigm, the structures become data attributes of potential classes.
Cons and difficulties: It may be the case that one global variable/type is associated with a high degree of confidence with a number of other global variables/types i.e. when the global variable/type is accessed, a number of other global variables/types are accessed. In this case, to avoid a large structure that hinders rather than promotes understandability, the software engineer needs to study the code and analyze which global variables/types are to be combined into a structure.

10.3.2 Pattern 6: Strengthen encapsulation:

Intent: Identify potential classes.
Metarule 6: *Accessed by* $(X) \Rightarrow$ *Type*(X, Y)
Confidence: *One*

Implication: The function(s) access the type(s) on the RHS.

Problem: Many legacy systems were developed using the structured approach. To facilitate re-use and ease of maintenance, how can these systems may be modularised as object-oriented systems?
Forces: This problem is difficult because

1. Identifying potential classes accurately in such large systems is not easy.

Solving this problem is feasible because

1. Classes contain related data attributes for which operations are defined. Related data attributes can be identified using pattern 5 (increase data modularity).

Solution: Identify collections of functions and the common data set they access. These represent potential classes and should be packaged together to strengthen encapsulation and provide information hiding.
Trade-offs:
Pros: Large programs that use information hiding have been found easier to modify by a factor of 4 than programs that do not [43]. Information hiding forms a foundation for both structured and object-oriented design.
Cons and difficulties: Functions may access more than one type, in which case a careful study of the code is required to decide the type with which the function should be associated. It may be the case that the types accessed by a function form a coherent entity which can be transformed into a structure (See Pattern 5).