Bayesian Learning for Software Architecture Recovery

O. Maqbool[1] and H.A. Babri[2],
[1]Quaid-i-Azam University, Islamabad, Pakistan
[2]University of Engineering and Technology, Lahore, Pakistan
onaiza@qau.edu.pk, babri@uet.edu.pk

Abstract-**Understanding a software system at the architectural level is especially important when the system is to be adapted to meet changing requirements. However, architectural documentation is often unavailable or out-of-date. In this paper, we explore the use of Bayesian learning methods for automatic recovery of a software system's architecture, given incomplete or out-of-date documentation. We employ software modules with known classifications to train the Naïve Bayes Classifier. We then use the classifier to place new instances, i.e. new software modules, into appropriate sub-systems. We evaluate the performance of the classifier by conducting experiments on a software system, and compare the results obtained with a manually prepared architecture. We present an analysis of the results, and also discuss the assumptions under which the results are expected to be meaningful.**

Keywords-**Bayesian Learning, Naïve Bayes Classifier, Software Architecture Recovery**

I. INTRODUCTION

Machine learning is a subfield of Artificial Intelligence (AI) which deals with development of techniques that allow computers to learn. Learning refers to adaptive changes made, so that there is an improvement in performance with experience. Machine learning techniques offer certain advantages over traditional AI techniques such as knowledge-based systems, e.g. they can adapt their behaviour according to the data set that is encountered, and are particularly useful when knowledge about the problem cannot be transformed into an algorithmic form. Machine learning techniques have been successfully applied to a number of problems in various domains e.g. pattern recognition, prediction and detection. In recent years, machine learning techniques have been applied by researchers to support a variety of software engineering activities, including prediction of software development and maintenance effort, program transformation, reuse library construction and maintenance, and requirement acquisition [1].

A software engineering problem to which machine learning techniques have been applied recently is the architecture recovery of legacy software systems. The architecture of a software system refers to the major components or sub-systems within the system, and their interaction with one another. An architectural level understanding of a software system is important especially when the system is to be adapted to meet changing requirements. In case the architectural documentation is not available, as is often the case for old or legacy software systems, an attempt must be made to recover the architecture from the source code. A number of machine learning techniques have been employed for this purpose including clustering [2] - [7], genetic algorithms [8], and association rule mining [9] - [11].

In certain cases, the architectural documentation for a legacy software system may be available, but may be incomplete or out-of-date. This is because legacy software systems are expected to have undergone a number of changes during their lifetime. Even if architectural documentation was developed for the original system, it is likely that the documentation wasn't updated to reflect subsequent changes. In such a case, a partial view of the architecture may be available. It will be useful to utilize the available documentation to determine the appropriate sub-systems to which any newly developed modules (not present in the original documentation) belong, and to place the new software modules into appropriate sub-systems.

In this paper, we present a Bayesian learning based approach to automatically recover a software system's architecture given incomplete or out-of-date documentation. We use software modules present in the existing architectural documentation to train the Naïve Bayes classifier. We then use the classifier to predict an appropriate sub-system for the new software modules. We evaluate the performance of our approach by conducting experiments on a software system, and comparing the results with a manually prepared decomposition.

The organization of this paper is as follows. In section II we present an overview of Concept learning and Bayesian learning. In section III, we describe the Naive Bayes classifier. In section IV, we detail our Bayesian learning based approach. Section V presents experimental results. Finally, in section VI we present the conclusions.

II. AN OVERVIEW OF CONCEPT AND BAYESIAN LEARNING

In this section, we provide a brief introduction to Concept learning and Bayesian learning. For details, the reader may refer to [12], from where this material has been abstracted.

A fundamental task in learning is to be able to generalize from the specific, i.e. acquire general concepts from specific training examples, a task referred to as concept learning. The concept to be learned is called the target concept *c*. Learning takes place through a set of training examples of the form <*x*, *c(x)*>, where *x* is an instance represented by attributes, and *c(x)* is the target concept value. As an example, consider a set of instances *X* representing luggage, described by attributes size (having values big, small, normal), and weight (having values high, normal). Let the target concept values be acceptable or unacceptable. A possible set of training examples, denoted by *D*, is presented in Table 1.

TABLE 1
A SET OF TRAINING EXAMPLES

Example	Size	Weight	Acceptable
E1	Big	High	No
E2	Normal	Normal	Yes
E3	Small	Normal	No

Concept learning can also be formulated as a search problem, where the hypothesis space *H* is searched for a hypothesis that best fits the training examples.

Bayesian learning takes a probability-based approach to reasoning and inferring results. Each training example that is encountered can change the probability that a hypothesis is correct. Moreover, rather than using knowledge from the current data set or training examples only, prior knowledge can be combined with the observed data to arrive at more meaningful results. Bayesian learning methods are based on the Bayes theorem. To understand the Bayes theorem, consider the problem of determining the best hypothesis from some hypothesis space *H*, given a set of training examples *D*. Let *P(h)* denote the prior probability that hypothesis *h* holds, and let *P(D)* denote the prior probability that data *D* will be observed. Moreover, let *P(D|h)* denote the probability that data *D* will be observed given that *h* holds. Then, according to the Bayes theorem, the posterior probability *P(h|D)* is given by (1):

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)} \tag{1}$$

In learning problems, we are interested in finding the most probable hypothesis *h* given data *D*. Such a hypothesis is called a *maximum a posteriori* or *MAP* hypothesis, and is given by:

$$h_{MAP} \equiv \underset{h \in H}{\operatorname{argmax}} P(h \mid D) \tag{2}$$

It can be seen from (2) that in general, Bayesian learning techniques are computationally expensive, since the computational cost to determine the best hypothesis is linear in the number of hypotheses in the hypothesis space *H*.

III. THE NAÏVE BAYES CLASSIFIER

The Naïve Bayes classifier is a Bayesian learning method that has been successfully applied to solve many practical problems e.g. text classification, speech/image recognition. The problem setting in which the Naïve Bayes classifier is applied can be described as follows:

Consider a learning task where each instance *x* is described by a conjunction of *k* attributes values $a_1, a_2, \ldots, a_k$. Let the target function be denoted by *f(x)*, where *f(x)* can take on any value $v_1, v_2, \ldots, v_j$ from the set *V*. The set of training examples provided to the learner thus consists of a set of instances, along with the target classifications. Given a new instance, the learning task is to predict the target value for this instance.

According to the Bayesian approach, the most probable target value v_{map}, given the attribute values $a_1, a_2, \ldots, a_k$. is given by:

$$v_{MAP} \equiv \underset{vj \in V}{\operatorname{argmax}} P(v_j \mid a_1, a_2, .., a_k)$$

$$= \underset{vj \in V}{\operatorname{argmax}} P(a_1, a_2, .., a_k \mid v_j) P(v_j)$$

The Naïve Bayes classifier makes the simplifying assumption that the attribute values are conditionally independent given the target value, i.e. $P(a_1, a_2, .., a_k \mid v_j) = \prod_i P(a_i \mid v_j)$. Thus according to the Naïve Bayes classifier, the most probable target value is given by:

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i \mid v_j)$$

To illustrate the working of the Naïve Bayes classifier, consider the training examples in Table 1. Suppose we have to classify the new instance <Big, Normal>. From Table 1, v_1=No, and $P(v_1)$=2/3, v_2=Yes, and $P(v_2)$=1/3. *P(Big|No)*=1/2, *P(Big|Yes)*=0, *P(Normal|No)*=1/2, *P(Normal|Yes)*=1. Thus for v_1, *P(No)P(Big|No)P(Normal|No)*=2/3.1/2.1/2=1/6, and for v_2, *P(Yes)P(Big|Yes)P(Normal|Yes)* = 1/3.0.1=0. Thus $v_{NB} = v_1$.

The conditional independence assumption made by the Naïve Bayes classifier allows the classifier to use fewer training examples than would be required otherwise. Moreover, the classifier forms a hypothesis without searching through the hypothesis space. One difficulty in applying the classifier is illustrated in the example discussed above, where the probability of "Yes" is reduced to 0, because *P(Big|Yes)* =0 and this term must be multiplied with all other probability terms. Thus if we estimate probabilities as in the example, it is quite likely that for a practical problem, probabilities are reduced to 0 for some target values, and the results obtained may not be meaningful.

IV. NAÏVE BAYES LEARNING BASED ARCHITECTURE RECOVERY APPROACH

To apply the Naïve Bayes classifier to recover the architecture of a software system, given incomplete or out-of-date documentation, we adopted the following approach:

1. To formulate our problem (i.e. placing a module/function into an appropriate sub-system) as a learning problem, the first issue we addressed was choosing a representation for the software system. The set of instances *X*, attribute values $a_1, a_2, \ldots, a_k$ describing the set of instances, target values *c(x)*, and training examples *D* for our software system are as follows:

- Each software function in the software system represents an instance *x*. Thus the set of functions within the system represents the set of instances *X*.
- Each instance *x* i.e. software function, is described by attributes in the form of the global variables $g_1, g_2, \ldots, g_s$ accessed by a software function. Other attributes may also be employed e.g the user defined types accessed by a software function, and/or the function calls made by a software function.
- The sub-system to which a software function belongs represents the target values *c(x)* of an instance. If there are *j* sub-systems, then the target values may be represented by $v_1, v_2, \ldots, v_j$.

Thus the set of training examples *D* for our system consists of ordered pairs of the form <*x, c(x)*>, where *x* represents a function described by attributes, and *c(x)* is the sub-system to which the function belongs. As an example, consider a hypothetical software system consisting of four functions (f_1, f_2, f_3, f_4) whose classifications are known. Suppose f_1 accesses global variables g_1 and g_2, and belongs to sub-system S_1. Suppose f_2 accesses global variables g_2 and g_3, and is part of sub-system S_2, f_3 accesses global variables g_1 and g_3, and belongs to sub-system S_1, and f_4 accesses global variable g_1 and belongs to sub-system S_2. The set of training examples for this system are presented in Table 2. 'T' denotes that a global variable is accessed by a function, and 'F' denotes that the variable is not accessed.

TABLE 2
TRAINING EXAMPLES FOR OUR SOFTWARE SYSTEM

Example	g_1	g_2	g_3	Sub-system
f_1	T	T	F	S_1
f_2	F	T	T	S_2
f_3	T	F	T	S_1
f_4	T	F	F	S_2

2. The second issue we addressed was how to estimate the probabilities that the Naïve Bayes classifier requires. We use the approach described in Section III to estimate the probability that a software function belongs to a certain sub-system. To illustrate our approach, consider the training examples in Table 2. According to the Naïve Bayes classifier,

$$v_{NB} = \underset{v_j \in V}{\arg\max} P(v_j) \prod_i P(a_i \mid v_j)$$

Here $v_1=S_1$, and $v_2=S_2$, $P(v_1)$=2/4 and $P(v_2)$=2/4. Suppose we have to classify a new instance f_5 accessing variables g_1, g_2 and g_3. Then the feature vector of f_4={*T,T,T*}. From Table 2, $P(a_1|v_1)= P(T|S_1)$ =1 and $P(a_1|v_2)=P(T|S_2)$=1/2. Similarly, $P(a_2|v_1)= P(T|S_1)$=1/2, $P(a_2|v_2)=P(T|S_2)$=1/2, $P(a_3|v_1)= P(T|S_1)$ =1/2, $P(a_3|v_1)=P(T|S_2)$=1/2. So $P(S_1)P(T|S_1)\ P(T|S_1)\ P(T|S_1)$=1/2*1*1/2*1/2=1/8, and $P(S_2)P(T|S_2)\ P(T|S_2)\ P(T|S_2)$=1/2*1/2*1/2*1/2=1/16. Thus according to the Naïve Bayes classifier, the most probable classification for the function f_4 is S_1, in other words the function f_4 belongs to S_1.

It is relevant to note at this stage that under the assumption made by the Naïve Bayes classifier, the probabilities of occurrence of attributes are independent of each other. This assumption may not be correct, since it may be that two global variables are always accessed together, so that if one is accessed, the probability of occurrence of the second global variable is high. However, the Naïve Bayes classifier performs well for other problems, for example text classification, even when this assumption is clearly incorrect [12]. Thus we perform our architecture recovery experiments in the software domain under the restrictive assumption.

V. EXPERIMENTAL SETUP AND RESULTS

A. *The test system*

To conduct the experiments, we chose Mosaic version 2.6, an open source web browser written in C. Mosaic was developed at the University of Illinois in Urbana-Champaign [13]. The software consists of 818 functions, 348 global variables and 112 user defined types. Relevant 'facts' about the test system have been parsed using the Rigi tool [14], and stored in a 3-tuple format called the Rigi standard format (RSF). Facts of interest to us include functions present in the software system, and the global variables accessed by these functions.

We asked an experienced software engineer to develop an architectural description of Mosaic based on the source code. According to the architectural description, the system consists of 11 major sub-systems, with an average of 75 functions per sub-system. The sub-systems, along with the number of functions in each sub-system are given in Table 3. A more detailed description of the expert decomposition can be found at [15].

TABLE 3
MOSAIC SUB-SYSTEMS

Subsystem name	Number of functions
User Interface Manager	219
CCI Manager	144
Mosaic Manager	113
Helpers	85
Image Processor	64
Annotations	52
Hot lists Manager	49
History Manager	45
News group Manager	30
Mail Processor	11

Mosaic Comments	6

Since the Mail Processor and Mosaic Comments sub-systems contain a relatively small number of functions as compared to other sub-systems, the functions within these two sub-systems were removed before conducting our experiments. Thus the total number of functions in our experiments was 801.

B. *The test set*

To evaluate the performance of the Naïve Bayes classifier, we require two sets of examples. A training set, in which the classification of the instances is known, is required to train the classifier. In our experiments, the training set consisted of functions along with sub-system names to which the functions belong. To test the performance of the Naive Bayes classifier after training, a test set must be available. To obtain the training and test sets, we used a k-fold validation technique [12]. Functions within the software system were divided into k=3 equally sized disjoint sub-sets. Thus the 801 functions available within the 9 sub-systems were divided into 3 subsets containing 267 functions each. K-1 of these sets were used for training, and the remaining set was used for testing. Thus our experiment was repeated 3 times, each time with a training set of 534 functions with known classifications, and a test set of 267 functions in which the classifications were not provided to the classifier.

C. *Experimental results*

Table 4 summarizes the results obtained by training the Naïve Bayes classifier on the three training sets and then testing its performance on test sets. The second column of Table 4 represents the percentage of correctly classified functions within the test set, i.e. the functions where the sub-system selected by the Naïve Bayes classifier matches the classification by the human expert.

TABLE 4
EXPERIMENTAL RESULTS FOR NAÏVE BAYES CLASSIFIER USING K-FOLD VALIDATION, K=3

Data Set	Correctly classified
1	49%
2	39%
3	43%
Average	**43.67%**

It can be seen from Table 4 that the percentage of correctly classified examples is highest for data set 1, where almost 50% of the examples are correctly classified. It is relevant to note that only 45% functions within Mosaic access global variables [16], and that on an average, a function accesses only one global variable. Keeping in mind the sparse nature of the function feature vector provided to the Naïve Bayes classifier, correct classification of 50% of the functions shows that the probabilistic approach taken by the classifier can be effectively used for predicting the sub-system to which a software function belongs.

For further analysis, in Table 6 – Table 8 we present the results in more detail for each of the three test data sets. The percentage of correctly classified functions is listed for each sub-system. It can be seen that the percentage of correctly classified functions is high for some sub-systems. Figure 1 summarizes the results presented in Table 6 - Table 8, by taking the average of correctly classified functions for each sub-system for the three data sets.

TABLE 5
EXPERIMENTAL RESULTS FOR NAÏVE BAYES CLASSIFIER FOR TEST DATA SET 1

Sub-systems	Functions	Correctly classified	%age
User Interface Manager	72	57	79%
CCI Manager	45	20	44%
Mosaic Manager	38	18	47%
Helpers	27	8	30%
Image Processor	22	7	32%
Annotations	18	6	33%
Hot lists Manager	18	3	17%
History Manager	15	5	33%
News group Manager	12	6	50%
Total	**267**	**130**	**49%**

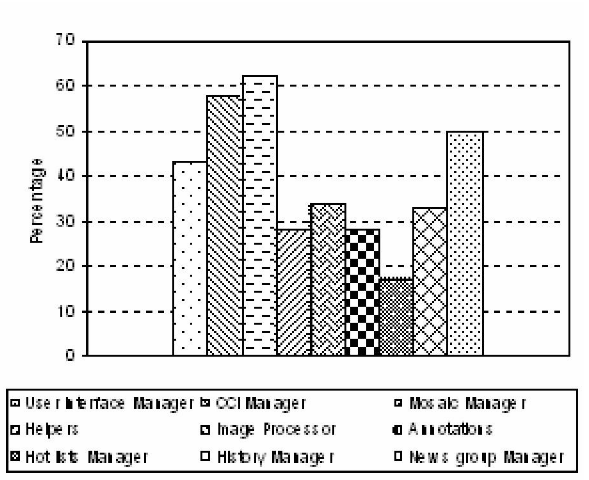
TABLE 6
EXPERIMENTAL RESULTS FOR NAÏVE BAYES CLASSIFIER FOR TEST DATA SET 2

Sub-systems	Functions	Correctly classified	%age
User Interface Manager	72	13	18%
CCI Manager	48	20	42%
Mosaic Manager	39	39	100%
Helpers	28	9	32%
Image Processor	21	7	33%
Annotations	19	2	11%
Hot lists Manager	16	2	13%
History Manager	15	5	33%
News group Manager	9	6	67%
Total	**267**	**103**	**39%**

TABLE 7
EXPERIMENTAL RESULTS FOR NAÏVE BAYES CLASSIFIER FOR TEST DATA SET 3

Sub-systems	Functions	Correctly classified	%age
User Interface Manager	75	24	32%
CCI Manager	51	45	88%
Mosaic Manager	39	15	38%
Helpers	27	6	22%
Image Processor	21	8	38%
Annotations	15	6	40%
History Manager	15	5	33%
Hot lists Manager	15	3	20%
News group Manager	9	3	33%
Total	**267**	**115**	**43%**

Fig 1. Mosaic – Average percentage of functions classified correctly by the Naïve Bayes Classifier in various sub-systems



From Fig. 1, it can be seen that the maximum percentage of correctly classified functions is 62% for the Mosaic manager sub-system, followed by 58% correctly classified functions for the CCI Manager. The Hotlists Manager has the least percentage of correctly classified functions (17%). In general, it is expected that sub-systems having a larger number of functions will have a higher percentage of correctly classified functions, since a larger number of examples in the training set generally produces better results. However, the number of training examples is not the only criterion for good results. If this was the case, the User Interface sub-system, which contains the largest number of functions would have produced best results, and results for sub-systems having a relatively lower number of functions e.g. Newsgroup manager would not have been as good as they are (50% correct classifications).

In addition to the number of training examples, the data within a training example is also a very important factor in determining the quality of results. For the Naïve Bayes classifier, we need training examples where a global variable is both accessed and not accessed by functions within a sub-system, in order to enable the classifier to calculate the probabilities appropriately. In case these examples are not present, the results may not be meaningful. For example, results may be adversely affected if a system contains a large number of functions that access no global variable. Figure 2 shows the percentage of correctly classified functions for each sub-system, along with the total percentage of functions that access no variables within each sub-system.

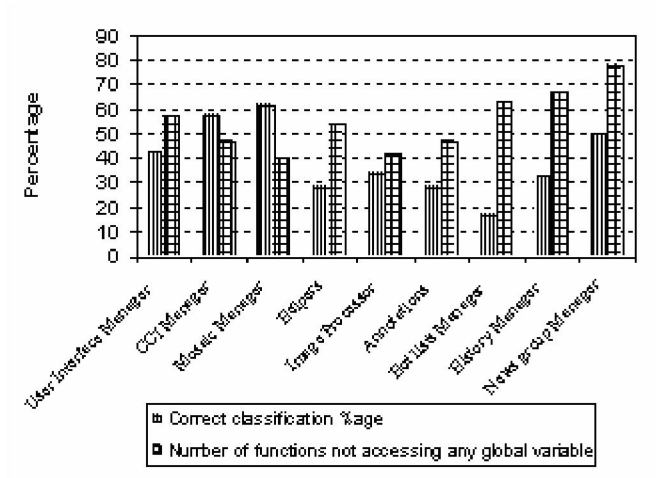


Fig. 2. Mosaic – Percentage of functions not accessing any global variable

It can be seen from Figure 2 that the quality of results obtained by the Naïve Bayes classifier depends both on the number of training examples, as well as the data contained within the training examples. In general, sub-systems with a higher percentage of functions accessing no global variables show worse results than sub-systems that contain a lower percentage of such functions. Exceptions arise in the case of Mosaic Manager, CCi Manager and User Interface Manager, but it is relevant to note that all three of these sub-systems contain a larger number of functions as compared to other sub-systems. Thus inspite of a relatively larger number of functions accessing no global variables, these sub-systems are still able to provide a sufficient number of training examples to the Naïve Bayes classifier in order for it to produce good results.

From the results presented in this section, it can be seen that the Naïve Bayes classifier performs reasonably well even though the training examples do not carry a large amount of information i.e. as discussed, the feature vector associated with each function (representing global variables accessed by it) is sparse. Thus we can conclude that if the training examples provided to the Naïve Bayes classifier contain meaningful information, the classifier can be effectively used for predicting the sub-system to which a software function belongs.

VI. CONCLUSIONS

In this paper, we explored the application of Bayesian learning techniques to the problem of recovering the architecture of a software system when the architectural documentation exists but may be incomplete or out-of-date. Functions with known classifications (i.e. functions which are known to belong to certain sub-systems), are used as training examples. We proposed the use of the Naïve Bayes classifier for learning through these training examples, and predicting the sub-systems to which new functions are expected to belong.

To test our approach, we performed experiments on an open source software system. We used the global variables accessed by a software function as attributes. Our experimental results are promising, and show that the efficacy of the approach depends on the number of training examples and the amount of information present within each training example.

In the future, we would like to use a larger set of attributes. Moreover, we would like to explore the use of other measures to calculate probability for the Naïve Bayes classifier. We would also like to apply other supervised learning techniques to the same problem, and compare the results produced by these techniques.

Acknowledgement

We would like to acknowledge Abdul Quddus Abbasi for developing the Naïve Bayes classifier software to conduct our experiments.

References

[1] D. Zhang and J. Tsai, "Machine learning and software engineering," In *Proc. of the Intl. Conference on Tools with Artificial Intelligence (ICTAI)*, pages 22-32, 2002.

[2] K. Sartipi, K. Kontogiannis, "A user-assisted approach to component clustering," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 4, pp. 265-295, July/August 2003.

[3] N. Anquetil and T.C. Lethbridge, "Experiments with clustering as a software remodularization method," In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pp. 235-255, 1999.

[4] J. Davey and E. Burd, "Evaluating the suitability of data clustering for software remodularization," In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pp. 268-277, 2000.

[5] M. Saeed, O. Maqbool, H.A. Babri, S.M. Sarwar, and S.Z. Hassan, "Software clustering techniques and the use of the combined algorithm," In *Proc. of the Intl. Conference on Software Maintenance and Reengineering (CSMR)*, pages 301-306, 2003.

[6] O. Maqbool and H.A. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," In *Proc. of the Intl. Conference on Software Maintenance and Reengineering (CSMR)*, pages 15-24, 2004.

[7] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, 31(2):150-165, 2005.

[8] B. Mitchell, *A Heuristic Search Approach to solving the Software Clustering Problem*. PhD thesis, Drexel University, 2002.

[9] C. M. D. Oca, and D. Carver, "Identification of data cohesive subsystems using data mining techniques," In *Proc. of the Intl. Conference on Software Maintenance (ICSM)*, pages 16-23, 1998.

[10] K. Sartipi, K. Kontogiannis and F. Mavaddat, "Design recovery using data mining techniques," In *Proc. of the Intl. Conference on Software Maintenance and Reengineering (CSMR)*, pages 129-140, 2000.

[11] C. Tjortjis, L. Sinos and P. Layzell, "Facilitating program comprehension by mining association rules from source code," In *Proc. of the Intl. Workshop on Program Comprehension (IWPC)*, pages 125-133, 2003

[12] T. Mitchell, *Machine Learning*, McGraw Hill, 1997.

[13] Website mosaic http://archive.ncsa.uiuc.edu/sdg/software/mosaic.

[14] Website rigi http://www.rigi.csc.uvic.ca/.

[15] Website reverse engineering http://suraj.lums.edu.pk/ reverseeng.

[16] O.Maqbool, H.A.Babri, A. Karim and S.M. Sarwar, "Metarule-guided association rule mining for program understanding," *IEE Software,* 152(6):281-296, December 2005.