Software Specifications Mining using Transaction Mapping Algorithm

R. Jeevarathinam
Assistant Professor
Department of Computer Science SNR Sons College,
Coimbatore, India

Dr. Antony Selvadoss Thanamani
Professor ad Head
Department of Computer Science
NGM College, Pollachi, India

ABSTRACT
Specification mining is a dynamic analysis process aimed at automatically inferring suggested specifications of a program from its execution traces. In software development it would be preferable if all programs and software projects are developed with clear, precise and documented specifications. But due to hard deadlines and `short-time-to-market' requirement, software products often come with project oriented, incomplete and even without any documented specifications. This situation is further motivated by a phenomenon termed as software evolution. As software evolves the documented specification is often not updated. This might render the original documented specification of little use after several cycles of program evolution. The above factors have contributed to high software maintenance costs. In this paper a novel technique to efficiently mine software specifications, called TM_TraceMiner is proposed which mines software specifications from program execution traces. To address the limitations of Apriori-like methods and FP-growth methods, a mining paradigm has been proposed, which uses Transaction Mapping algorithm.

Keywords
Algorithms, Apriori, FP-growth, mining specifications, program execution traces, transaction mapping.

1. INTRODUCTION
Specifications mining first proposed by Ammons et al [1] has received intensive research due to its wide range of applicability in many real life domains. They mine specifications from program execution traces. Let a program execution trace be a sequence of method calls to an application interface API [2,3]. Given a set of program execution traces, a small portion might be erroneous. The specification miner infers sequencing of constraints among the method calls. A trace might only be different to another due to different numbers of loop iterations during program execution. Given a database containing specifications, specification mining is a task of identifying specifications that satisfy a minimum support [4].

Mining specifications can be done by using Association rule mining. Association rules mining is a very popular data mining techniques and it finds relationships among the different entities of records (for example specifications records). Since the introduction in 1993 by Agrawal et al.[5], it has received a great deal of attention in the field of knowledge discovery and data mining. The problem of association rules mining was introduced in [5] and was improved to obtain the Apriori algorithm in [6]. The Apriori algorithm employs the downward closure property- if an itemset is not frequent, any superset of it cannot be frequent either. The Apriori algorithm performs a breadth-first search in the search space. TraceMiner is an Apriori based method which mines software specifications from program execution traces by employing a search lattice and search tree to store the execution trace data sets [7].

FP-growth is a well known algorithm that uses FP-tree data structure to achieve a condensed representation of the data base transactions and employs a divide-and-conquer approach to decompose the mining problem into a set of smaller problems [8]. FP-TraceMiner [9] is a FP-growth based method which mines all the frequent execution traces by recursively finding all frequent traces from the trace database. In FP-growth based algorithms, recursive construction of the FP-tree affects the algorithm's performance. In this paper, a novel approach that maps and compresses the transaction id list of each item (trace) into an interval list using a transaction tree and counts the support of each item (trace) by intersecting these interval lists. The frequent traces are found in a depth-first order along a lexicographic tree. The basic idea is to save the intersection time by mapping trace ids into continuous trace intervals. The rest of the paper is arranged as follows: Section 2 introduces the basic concept of association rules mining, two types of data representation, and the lexicographic tree used in the proposed algorithm. Section 3 addresses the TM-TraceMiner algorithm. Section 4 compares the TM-TraceMiner with two other algorithms-TraceMiner and FP-TraceMiner. Section 5 concludes the paper.

2. BASIC PRINCIPLES

2.1 Association Rules Mining
Let I={e1,e2,…,em} be a set of events and TD be a database having a set of traces where each trace T is a sub set of I. An association rule is an association relationship of the form: X=>Y, where X is a subset of I, Y is a subset of I and $X \cap Y = \emptyset$. The support of rule X => Y is defined as the percentage of traces containing both X and Y in TD. The confidence of X => Y is defined as the percentage of traces containing X that also contain Y in TD. The task of association rules mining is to find all strong association rules that satisfy a minimum support threshold (min-sup) and a minimum confidence threshold (min-conf). Mining association rules consists of two phases. In the first phase, all frequent traces that satisfy the min-sup are found. In the second phase, strong association rules are generated from the frequent traces found in the first phase. Most research considers only the

first phase because once frequent traces are found, mining association rules is trivial.

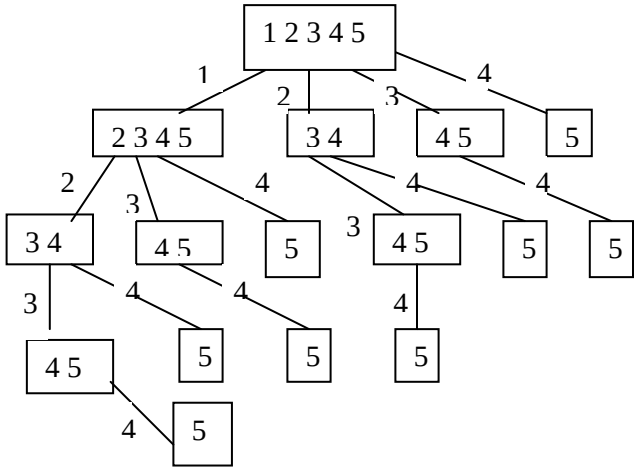


Fig. 1. Illustration of lexicographic tree

2.2 Data Representation

Two types of database layouts are employed in association rule mining: horizontal and vertical. In the horizontal database layout, each trace consists of a set of events and the database contains a set of traces. Most Apriori-like algorithms use this type of layout. For vertical database layout, each event maintains a set of trace ids (denoted by tr-id-set) where this event is contained. This layout could be maintained as a bit vector. It has been shown that vertical layout performs generally better than the horizontal format [10,11]. Table 1, Table 2 and Table 3 show examples for different types of layouts.

Table 1. Horizontal Representation

Tr-id	Traces
1	<2,1,5,3>
2	<2,3>
3	<1,4>
4	<3,1,5>
5	<2,1,3>
6	<2,4>

Table 2 Vertical Tr-id-set Representation

Trace	Tr-id-set
1	<1,3,4,5>
2	<1,2,5,6>
3	<1,2,4,5>
4	<3>
5	<1,4>

Table 3 Vertical Bitvector Representation

Trace	Bitvector
1	<1,0,1,1,1,0>
2	<1,1,0,0,1,1>
3	<1,1,0,1,1,0>
4	<0,0,1,0,0,0>
5	<1,0,0,1,0,0>

2.3 Lexicographic Prefix Tree

In this paper, a lexicographic prefix tree data structure is employed to generate candidate trace sets and count their frequency, which is similar to the lexicographic tree used in the TreeProjection algorithm [12]. An example of this tree is shown in Fig. 1. Each node in the tree stores a collection of frequent trace sets together with the support of these trace sets. The root contains all frequent 1-trace sets. Each edge in the tree is labeled with a trace event. Trace sets in any node are stored as singleton sets with the understanding that the actual trace set also contains all the events found on the edges from this node to the root node. For example consider the leftmost node in level 2 of the tree in Fig. 1. There are four 2-trace sets in this node, namely {1,2},{1,3},{1,4} and {1,5}. The singleton sets in each node of the tree are stored in the lexicographic order. If the root contains {1},{2},…,{n}, then, the nodes in level 2 will contain {2},{3},…{n};{3},{4},…,{n};…;{n}, and so on. For each candidate trace set a list of transaction ids are stored. This tree will not be generated in full. The tree is generated in depth-first order and minimum information needed to continue the search only stored. This means that at any instance, at most a path of the tree will be stored. As the search progresses, if the expansion of a node cannot possibly lead to the discovery of trace sets that have minimum support, then the node will not be expanded and the search will backtrack. As a frequent trace set that meets the minimum support requirement is found, it is output. Candidate traces sets generated by depth-first search are the same as those generated by the joining step (without pruning) of the Apriori algorithm.

Table 4. A Sample Trace Database

Tr-id	Traces	Ordered frequent traces
1	<2,1,5,3,19,20>	<1,2,3>
2	<2,6,3>	<2,3>
3	<1,7,8>	<1>
4	<3,1,9,10>	<1,3>
5	<2,1,11,3,17,18>	<1,2,3>
6	<2,4,12>	<2,4>
7	<1,13,14>	<1>
8	<2,15,4,16>	<2,4>

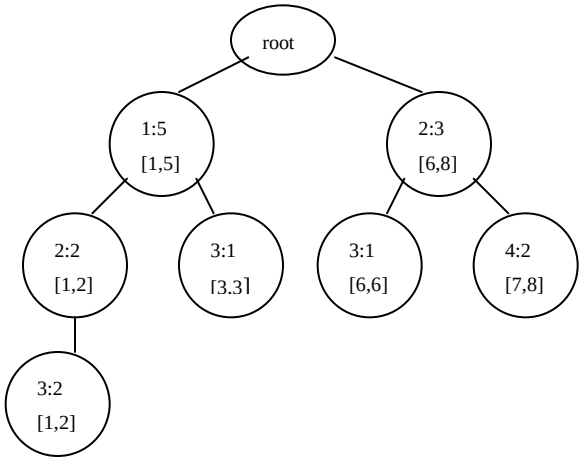


Fig. 2. A transaction tree for the database shown in Table 4

3. TRANSACTION MAPPING ALGORITHM

The contribution is that transaction ids are compressed for each trace set to continuous intervals by mapping trace ids into different space appealing to a transaction tree. Frequent trace sets are found by intersecting these interval lists instead of intersecting the trace id lists. The construction of transaction tree [15] is as follows.

3.1 Transaction Tree

The transaction tree is similar to FP-tree except that there is no header table or node link. The transaction tree can be thought of as a compact representation of all the transactions in the database. Each node in the tree has an id corresponding to an event and a counter that keeps the number of transactions that contain this event in this path. The construction of the transaction tree is as follows:

1. Scan through the trace database once and identify all the frequent 1- trace sets and sort them in descending order of frequency. At the beginning, the transaction tree consists of just a single node (which is a dummy root).
2. Scan through the trace database for a second time. For each trace, select items that are in frequent 1-trace sets, sort them according to the order of frequent 1-trace sets, and insert them into the transaction tree. When inserting an event, start from the root. At the beginning, the root is the current node. In general, if the current node has a child node whose id is equal to this event, then just increment the count of this child by 1; otherwise, create a new child node and set its counter as 1.

Table 4 and Fig. 2 illustrate the construction of a transaction tree. Table 4 shows an example of a trace database and Fig. 2 displays the constructed transaction tree assuming the minimum support count is 2.

3.2 Transaction Mapping and the Construction of Interval lists

After the transaction tree is constructed, all the transactions that contain an item are represented with an interval list. Each interval corresponds to a contiguous sequence of relabeled ids. Each node in the transaction tree will be associated with an interval. The construction of interval lists for each event is done recursively starting from the root in a depth-first order. In addition to the events, each element of a node in the lexicographic tree also stores a trace interval list. By constructing the lexicographic tree in a depth-first order, the support count of the candidate trace set is computed by intersecting the interval lists of the two events.

3.3 Transaction Mapping-TraceMiner Algorithm (TM-TraceMiner)

There are four steps involved in this algorithm:

1. Scan through the trace database and identify all frequent-1 trace sets.
2. Construct the transaction tree with counts for each node.
3. Construct the transaction interval lists. Merge intervals if they are mergeable.
4. Construct the lexicographic tree in a depth-first order keeping only the minimum amount of information necessary to complete the search. This means that no more than a path in the lexicographic tree will ever be stored. While, at any node, if further expansion of that will not be fruitful, then the search backtracks. When processing a node in the tree for every element in the node, the corresponding interval lists are computed by interval intersections. As the search progresses, trace set with enough support is output.

4. EXPERIMENTS AND PERFORMANCE EVALUATION

4.1 Comparison with TraceMiner and FP-TraceMiner

Experiments had been performed on both synthetic and real datasets to evaluate the scalability of our mining algorithm and the effectiveness of our pruning strategy. Three datasets are used in these experiments: a synthetic and two real datasets. Synthetic data generator provided by IBM was used with modification to ensure generation of sequences of events. The generators accept a set of parameters. The parameters D, C, N and S correspond

respectively to the number of sequences (in 1000s), the average number of events per sequence, the number of different events (in 1000s) and the average number of events in the maximal sequences. The experiment is tested with the dataset D5C20N10S20. It is also experimented on Gazelle dataset from KDD Cup 2000 which was also used to evaluate TraceMiner and FP-TraceMiner. It contains 29369 sequences with an average length of 3 and a maximum length of 651.

To evaluate this algorithm performance on mining from program traces, we generate traces from a simple Traffic alert and Collision Avoidance System (TCAS) from the Siemens Test Suite [13], which has been used as one of the benchmarks for research in error localization [14]. The test suite comes with 1578 correct test cases. We run these test cases to obtain 1578 traces. To test for scalability, instead of tracing method invocations, we trace executions of basic blocks of TCAS's control flow graph. A basic block is a maximal sequence of statements such that the execution of one statement will always results in the execution of the subsequent statements in the sequence. Each trace of basic block ids is treated as a sequence. The sequences are of average length of 36 and maximum length of 70. It contains 75 different events - the events are the basic block ids of the control flow graph of TCAS as shown in Table 5. This dataset is called as TCAS dataset.

Environment and Pattern Miners: All experiments were performed on a Pentium 4 3.0GHz PC with 2GB main memory running Windows XP Professional. Algorithms were written using Java running with Net Beans Frame work.

Table 5. Performance details of TM-TraceMiner

Details	TM-TraceMiner
No. of Testcases	1578
No of Traces	1578
Average length	36
Maximum length	70
No. of Events	75

Experiment Results and Analysis: The results of experiments performed on the D5C20N10S20, Gazelle and Siemens dataset using closed and full-set iterative pattern miners are shown in Figures 3, 4 & 5 respectively. The Y-axis (in log scale) corresponds to the runtime taken or the number of generated patterns. The X-axis corresponds to the minimum support thresholds. The thresholds are reported relative to the number of sequences in the database. Note that, different from sequential patterns, due to repeated patterns within a sequence this number can exceed 1

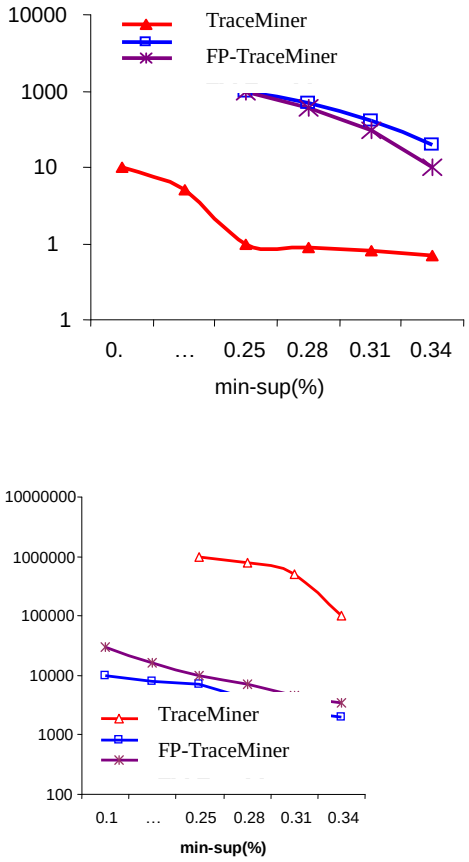
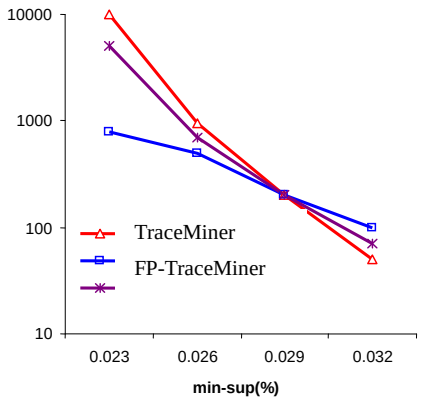


Fig 3: Performance results of varying *min_ sup* for D5C20N10S20 dataset



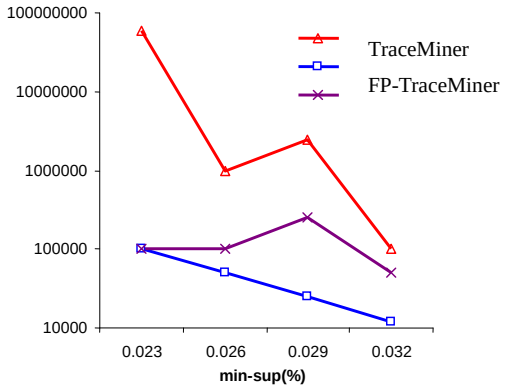


Fig. 4: Performance results of varying *min_ sup* for Gazelle dataset

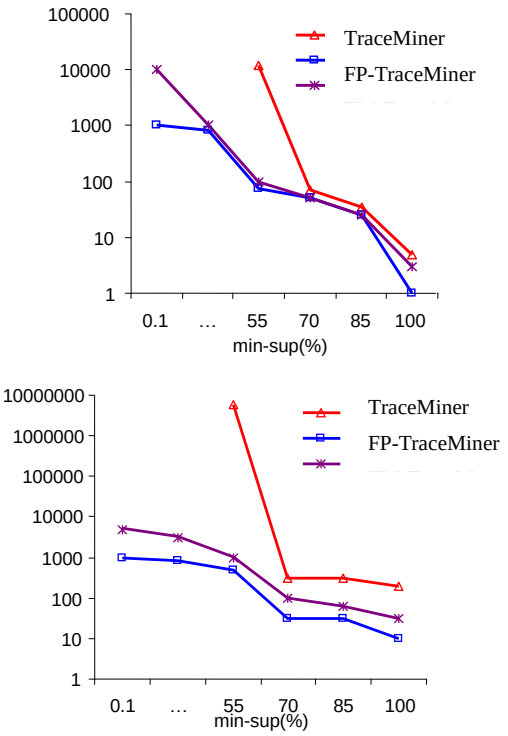


Fig 5: Performance results of varying *min sup* for TCAS dataset

From the plotted results it is noted that the pruning strategy significantly reduces the runtime and the number of patterns mined especially on low support threshold and when the reported patterns are long. Admittedly, when the numbers of closed and full-set of patterns differ by only a small factor, the overhead of mining using TraceMiner may result in longer runtime as compared to mining a FP-TraceMiner. However, when the length of the patterns is long, the number of TraceMiner is likely to be much less than that of a FP-TraceMiner.

For all datasets, even at very low support, TraceMiner is able to complete within less than 17 minutes. TCAS dataset especially highlights performance benefit of our pruning strategy. TM-TraceMiner is able to run even at the *lowest possible support threshold* (at 1 instance) within less than 17 minutes. On the other hand, Fp-TraceMiner runs with excessive runtime (> 6 hours) even at a relatively high support threshold of 867 instances. But TM-TraceMiner runs within 25 minutes.

5. CONCLUSION

In this paper, a new algorithm TM-TraceMiner is presented using the vertical database representation. Trace ids of each trace set are transformed and compressed to continuous transaction interval lists in a different space using transaction tree and frequent trace sets are found by transaction intervals intersection along a lexicographic tree in depth-first order. Through experiments the TM-TraceMiner algorithm has been shown gain to significant performance improvement over TraceMiner and FP-TraceMiner. This paper also gives an efficient method to mine the specifications from program execution traces. Traces deviating from common trace population rules are removed. The resultant filtered traces are then separated into multiple clusters. By clustering common traces together, it is expected that the learner is able to learn better and over-generalization of a subset of traces is not propagated to other clusters. These clusters of filtered traces are then inputted to a specification miner. This algorithm confirms the usefulness of the proposed method in discovering software specifications in iterative pattern form. Besides mining software behavioral pattern, it is believed that the proposed mining technique can potentially be applied to other knowledge discovery domains

6. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 4–16, 2002

[2] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proc. of Int. Conf. on Software Engineering*, 2006.

[3] The Java hotspot performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html

[4] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2001.

[5] R.Agrawal, T. Imielinski, and A.N.Swami, "Mining Association Rules Between Sets of Items in Large Databases",Proc.ACM SIGMOD Int'l Conf.Management of Data,pp.207-216, May 1993.

[6] R.Agrawal, T. Imielinski, and A.N.Swami, "Mining Association Rules Between Sets of Items in Large Databases",Proc.ACM SIGMOD Int'l Conf.Management of Data,pp.207-216, May 1993.

[7] R.Jeevarathinam and Antony Selvadoss Thanamani, "An Efficient Algorithm for Mining Software Specifications from Program Execution Traces" presented at Int'l Conf. Sensors, Security, Software and Intelligent Systems (ISSSIS 2009)

[8] Han J., Pei J., Yin Y.: Mining frequent patterns without candidate generation. Proc. of the 2000 ACM SIGMOD Conf. on Management of Data (2000)

[9] R.Jeevarathinam and Antony Selvadoss Thanamani, "An Implementation of FP-growth algorithm for Software Specification Mining", CIIT Int'l Journal of Data mining and Knowledge Engineering, Vol-1, pp.18-23, April 2009.

[10] M.J.Zaki, S. Parthasarathy, M.Ogihara, and W.Li, "New Algorithms for Fast Discovery of Association Rules," Proc. Third Int'l Conf. Knowledge Discovery and Data Mining, pp. 283-286,1997

[11] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M.Bawa, and D. Shah, "Turbo-Charging Vertical Mining of Large Databases," Proc. ACM SIGMOD Int'l Conf. Management of Data, pp.22-23, May 2000.

[12] R.Agrawal, C.Aggrawal, and V.Prasad, "A Tree Projection Algorithm for Generation of Frequent Item Sets," Parallel and Distributed Computing, pp.350-371, 2000

[13] D. Lo and S-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specifications miner. In SIGSOFT FSE, 2006.

[14] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In ICDE, 2004.

[15] M.Song and S. Rajesekaran, " A Transaction Mapping Algorithm for Frequent Item sets Mining", IEEE Trans. On Knowledge and Data Engineering, Vol. 18, No. 4, pp. 472-481. April 2006.