DynaMine, a tool that analyzes source code check-ins to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns. Potential patterns discovered through mining are passed to a dynamic analysis tool for validation; finally, the results of dynamic analysis are presented to the user.

Legacy code error patterns either are recoded as comments in the code or not documented at all. Some of the best efforts in error detection come from people intimately familiar with the application domain. As a result, lesser-known types of bugs and applications remain unexplored. To address this and apply error detection tools on “unfamiliar” applications, they propose an automatic way to extract likely error patterns by mining software revision histories. However, it is important to notice that not every pattern mined by considering revision histories is an actual usage pattern.

They looked for pattern violations at runtime, as opposed to using a static analysis technique. Some reasons they cited for this approach to make sense are static call graph construction presents a challenge for applications that use dynamic class loading. There are no chances of false positives as all pattern violations detected with our system actually do happen. One downside to this dynamic analysis is that error patterns that occur on exception paths that were not hit at runtime cant be detected.

In order to mine the patterns of method calls, they use a modified apriori algorithm. The algorithm iterates over the set of transactions (For each source revision, a transaction is a set of methods, calls to which have been inserted.) and forms patterns from the method calls that occur in the same transaction. It then computes association rules from the discovered patterns in P. The modification of apriori lies in that they restrict the frequent pattern mining to ignore method calls that either lead to no usage patterns. They also ignore the initial revisions. We’re not sure if this leads to some loss of patterns. Also given a call sequence c1().c2() . . . Cn() included as part of a repository change, they only take the final call Cn() into consideration. They also sensibly ignore most common repeated calls. They consider only the small patterns and fine-grained transactions in addition to mining the revision repository only for method pairs instead of patterns of arbitrary size.

They used standard ranking approaches to rank pattern found. A check-in may only add parts of a usage pattern to the repository. While this might be a problem for a general apriori algorithm, it implies a very strong correlation between one-line changes and bug corrections or fixes. Patterns with a high corrective rank result in more dynamic violations than patterns with a high regular rank.

A database stores method calls that have been inserted for each revision. To determine the calls inserted between two revisions r1 and r2, abstract syntax trees (ASTs) for both r1 and r2 are built and the set of all calls C1 and C2 can be computed by traversing the ASTs.

Then postprocessing is done to produce the final statistics about the number of times each pattern is followed and the number of times it is violated. These are then exported back into Eclipse for review. The user can browse through the results and ascertain which of the patterns she/he thought must hold do actually hold at runtime. Often, examining the dynamic output of DynaMine allows the user to correct the initial pattern.