DynaMine: Finding Common Error Patterns by Mining Software Revision Histories

Benjamin Livshits
Microsoft Research
livshits@microsoft.edu

Thomas Zimmermann
Saarland University
tz@acm.org

A great deal of attention has lately been given to addressing software bugs such as errors in operating system drivers or security bugs. However, there are many other lesser known errors specific to individual applications or APIs and these violations of application-specific coding rules are responsible for a multitude of errors. For instance, many APIs rely on methods being called in a specific order. These low-level rules are often not recorded and even developers familiar with the code base may not always know the invariants present in the code. In this paper we propose 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.

The combination of revision history mining and dynamic analysis techniques leveraged in DynaMine proves effective for both discovering new application-specific patterns *and* for finding errors when applied to very large applications with many man-years of development and debugging effort behind them. We have analyzed Eclipse and jEdit, two widely-used, mature, highly extensible applications consisting of more than 3,600,000 lines of code combined. By mining revision histories, we have discovered 56 previously unknown, highly application-specific patterns. Out of these, 21 were dynamically confirmed as very likely valid patterns and a total of 263 pattern violations were found.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging] Tracing; D.2.7 [Distribution, Maintenance, and Enhancement] Version control; H.2.8 [Database Applications] Data mining

General Terms: Management, Measurement, Reliability

Keywords: Error patterns, coding patterns, software bugs, data mining, revision histories, dynamic analysis, one-line check-ins.

1 Introduction

A great deal of attention has lately been given to addressing application-specific software bugs such as errors in operating system drivers [4, 14], security errors [24, 43], or errors in reliability-critical embedded software in domains like avionics [7, 8]. These represent critical errors in widely used software and tend to get fixed relatively quickly when found. A variety of static and dynamic analysis tools have been developed to address these high-profile bugs.

However, many other errors are specific to individual applications or platforms. This is especially true when it comes to extensible development platforms such as J2EE, .NET, and others that have a variety of programmers at all skill levels writing code to use the same sets of APIs. Violations of these application-specific coding rules, referred to as *error patterns*, are responsible for a multitude of errors. Error patterns tend to be re-introduced into the code over and over by multiple developers working on a project and are a common source of software defects. While each pattern may be only responsible for a few bugs in a given project snapshot, when taken together over the project's lifetime, the detrimental effect of these error patterns is quite serious and they can hardly be ignored in the long term if software quality is to be expected.

However, finding the error patterns to look for with a particular static or dynamic analysis tool is often difficult, especially when it comes to legacy code, where error patterns either are recoded as comments in the code or not documented at all [15]. Moreover, while well-aware of certain types of behavior that causes the application to crash or well-publicized types of bugs such as buffer overruns, programmers often have difficulty formalizing or even expressing API invariants. In additional to a handful of patterns that can be collected from the literature, newsgroups, and previous bug reports, application programmers are rarely able to tell which invariants the APIs they use have. The situation is only slightly better when it comes to software architects and API designers who are generally much more aware of application-specific patterns.

In this paper we propose an automatic way to extract likely error patterns by mining software revision histories. Moreover, in order to ensure that all the errors we find are relatively easy to confirm and fix, we pay particular attention in our experiments to errors that can be fixed with a *one-line change*. It is worth pointing out that many well-known error patterns such as memory leaks, double-`free`'s, mismatched locks, open and close operations on operating system resources, buffer overruns, and format string errors can often be addressed with a one-line fix. Looking at incremental changes between revisions as opposed to complete snapshots of the source allows us to better focus our mining strategy and obtain more precise results. Our approach uses revision history information to infer likely error patterns. We then experimentally evaluate the patterns we extracted by checking for them dynamically.

We have performed experiments on Eclipse and jEdit, two large, widely-used open-source Java applications. Both Eclipse and jEdit have many man-years of software development behind them and, as a collaborative effort of hundreds of people across different locations, are good targets for revision history mining. By mining CVS, we have identified 56 high-probability patterns in Eclipse and jEdit APIs, all of which were previously unknown to us. Out of these, 21 were dynamically confirmed as valid patterns and 263 pattern violations were found.

1.1 Contributions

This paper makes the following contributions:

1. We present DynaMine,[1] a *tool for discovering usage patterns and detecting their violations* in large software systems [29, 28]. All of the steps involved in mining and running the instrumented application are accessible to the user from within an Eclipse plugin: DynaMine automates the task of collecting and pre-processing revision history entries and mining for common patterns. Likely patterns are then presented to the user for review; runtime instrumentation is generated for the patterns that the user deems relevant. Results of dynamic analysis are also presented to the user in an Eclipse view.

2. We propose a *data mining strategy* that detects common usage patterns in large software systems by analyzing software revision histories. Our strategy is based on a classic Apriori data mining algorithm, which we augment in a number of ways to make it more scalable, reduce the amount of noise, and provide a new, effective ranking of the resulting patterns.

3. We present a *categorization of patterns* found in large modern object-oriented systems. Our experience with two large Java projects leads us to believe that similar pattern categories will be found in most other systems of similar size and complexity.

4. We propose a *dynamic analysis approach* for validating usage patterns and finding their violations. DynaMine currently utilizes an off-line approach that allows us to match a wider category of patterns. DynaMine supplies default handlers for analyzing most common categories of patterns.

5. We present a *detailed experimental study* of our techniques as applied to finding errors in two large, mature open-source Java applications with many years of development behind them. We have identified 56 patterns in both and found 263 pattern violations with our dynamic analysis approach. Furthermore, 21 patterns were experimentally confirmed as valid.

6. Finally, we provide an overview of the possible design options that combine revision history mining and program analysis techniques. We also give justifications of our design choices.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides an informal description of DynaMine, our pattern mining and error detection tool. In Section 3, we discuss the design choices that we made for DynaMine. Section 4 describes our revision history mining approach. Section 5 describes our dynamic analysis approach. Section 6 summarizes our experimental results for (a) revision history mining and (b) dynamic checking of the patterns. Sections 8, 9, and 10 present related and future work and conclude.

[1] The name DynaMine comes from the combination of Dynamic analysis and Mining revision histories.

Application	Version	Type of Extensions	Number Available
Linux	2.4.xx	drivers	1,739
Apache	2.0.53	modules	385
Eclipse	3.1	plugins	317
jEdit	4.2	plugins	277
Mozilla	1.7.6	plugins	56
Trillian	3.1	plugins	36

Figure 1: Extension statistics for some commonly used software platforms.

2 Overview of DynaMine

It is common for today's large software systems to support mechanisms such as plugins, extension modules, or drivers that allow expanding applications's functionality. Successful software platforms such as Apache, Eclipse rich client platform, Mozilla Firefox, and others support dozens of plugins. Figure 1 summarizes approximate numbers of available plugins, extension modules, or drivers for various software platforms. Extensions are written by programmers who develop code according to a set of predefined APIs.

It is generally recognized that plugins typically consist of lower quality code, in part because plugin writers are usually less aware of the requirements of the APIs they need to use. Inadvertently violating invariants of these APIs may take the form of forgetting to call a function such as `close` or `free` to release a resource or performing an action unnecessarily in an effort to maintain consistency leading to the same action performed multiples times. Many such programming mistakes in plugin code lead to subtle runtime errors that often occur *outside of the plugin* because of violated program invariants later in the program execution; this makes the cause of the error difficult to diagnose and fix.

A great deal of research has been done in the area of checking and enforcing specific coding rules, the violation of which leads to well-known types of errors. However, these rules are not very easy to come by: much time and effort has been spent by researchers looking for worthwhile rules to check [37] and some of the best efforts in error detection come from people intimately familiar with the application domain [14, 41]. As a result, lesser known types of bugs and applications remain virtually unexplored in error detection research. A better approach is needed if we want to attack "unfamiliar" applications with error detection tools. This paper proposes a set of techniques that automate the step of application-specific pattern discovery through revision history mining.

2.1 Motivation for Revision History Mining

Our approach to mining revision histories hinges on the following observation:

Observation 2.1 *Given multiple software components that use the same API, there are usually* common errors *specific to that API.*

File	Revision	Added method calls
Foo.java	1.12	o1.addListener o1.removeListener
Bar.java	1.47	o2.addListener o2.removeListener System.out.println
Baz.java	1.23	o3.addListener o3.removeListener list.iterator iter.hasNext iter.next
Qux.java	1.41	o4.addListener
	1.42	o4.removeListener

Figure 2: Method calls added across different revisions.

In fact, much of research done on bug detection so far can be thought of as focusing on specific classes of bugs pertaining to particular APIs: studies of operating-system bugs provide synthesized lists of API violations specific to operating system drivers resulting in rules such as "do not call the interrupt disabling function `cli()` twice in a row" [14]. In order to locate common errors, we mine for frequent usage patterns in revision histories, as justified by the following observation.

Observation 2.2 *Method calls that are frequently added to the source code simultaneously often represent a pattern.*

Looking at incremental changes between revisions as opposed to full snapshots of the sources allows us to better focus our mining strategy. However, it is important to notice that not every pattern *mined* by considering revision histories is an actual *usage* pattern.

Example 1. Figure 2 lists sample method calls that were added to revisions of files `Foo.java`, `Bar.java`, `Baz.java`, and `Qux.java`. All these files contain a usage pattern that says that methods

$$\{\texttt{addListener}, \texttt{removeListener}\}$$

must be precisely matched. However, mining these revisions yields additional patterns like

$$\{\texttt{addListener}, \texttt{println}\}$$

and

$$\{\texttt{addListener}, \texttt{iterator}\}$$

that are definitely *not* usage patterns. Furthermore, we have to take into account the fact that in reality some patterns may be inserted incompletely, e.g., by mistake or to

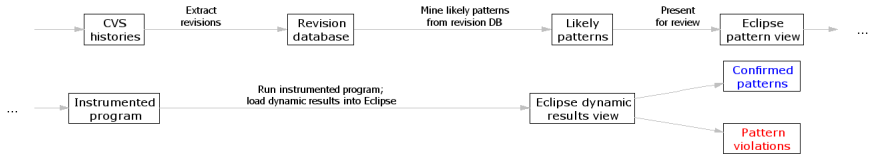

Figure 3: Architecture of our tool. The first row represents revision history mining. The second row represents dynamic analysis.

fix a previous error. In Figure 2 this occurs in file `Qux.java`, where `addListener` and `removeListener` were inserted independently in revisions 1.41 and 1.42. □

The observation that follows gives rise to an effective ranking strategy used in DynaMine.

Observation 2.3 *Small changes to the repository such as one-line additions often represent bug fixes.*

This observation is supported in part by anecdotal evidence and also by recent research into the nature of software changes [36] and is further discussed in Section 4.3. To make the discussion in the rest of this section concrete, we present the categories of patterns discovered with our mining approach.

- **Matching method pairs** represent two method calls that must be precisely matched on all paths through the program.
- **State machines** are patterns that involve calling more than two methods on the same object and can be captured with a finite automaton.
- **More complex patterns** are all other patterns that fall outside the categories above and involve multiple related objects.

The categories of patterns above are listed in the order of frequency of high-likelihood pattern in our experiments.

2.2 DynaMine System Overview

We conclude this section by summarizing how the various stages of DynaMine processing work when applied to a new application. All of the steps involved in mining and dynamic program testing are accessible to the user from within custom Eclipse views. A diagram representing the architecture of DynaMine is shown in Figure 3.

1. Pre-process revision history, compute methods calls that have been inserted, and store this information in a database.
2. Mine the revision database for likely usage and error patterns.
3. Present mining results to the user in an Eclipse plugin for assessment.

4. Generate instrumentation for patterns deemed relevant and selected by the user through DynaMine's Eclipse plugin.

5. Run the instrumented program and dynamic data is collected and post-processed by dynamic checkers.

6. Dynamic pattern violation statistics are collected and presented to the user in Eclipse.

Steps 4–6 above can be performed in a loop: once dynamic information about patterns is obtained, the user may decide to augment the patterns and re-instrument the application.

3 Design Decisions

In this section we discuss some of the design choices we made when working on DynaMine. Our approach for the rest of the section is to outline where our technique stands with respect to several design dimensions:

- Static vs dynamic analysis
- Amount of user involvement
- Granularity of mined information

Each of these dimensions is discussed in an individual section below.

3.1 Static vs Dynamic Analysis

Our approach is to look for pattern violations at runtime, as opposed to using a static analysis technique. This is justified by several considerations outlined below.

- **Scalability.** Our original motivation when starting the DynaMine project was to be able to analyze Eclipse, which is one of the largest Java applications ever created. The code base of Eclipse is comprised of more than 2,900,000 lines of code and 31,500 classes. Most of the patterns we are interested in are spread across multiple methods and need an interprocedural approach to analyze. Given the substantial size of the application under analysis, precise whole-program flow-sensitive static analysis is expensive. Moreover, static call graph construction presents a challenge for applications that use dynamic class loading [27]. In contrast, dynamic analysis does not require call graph information.

- **Validating discovered patterns.** A benefit of using dynamic analysis is that we are able to "validate" the patterns we discover through CVS history mining as real usage patterns by observing how many times they occur at runtime. Patterns that are matched a large number of times with only a few violations represent likely patterns with a few errors. The advantage of validated patterns is that they increase the degree of assurance in the quality of mined results. With static analysis, validating

patterns would not generally be possible unless flow-sensitive "must" information is available.

In contrast to a static technique, runtime analysis does not suffer from false positives because all pattern violations detected with our system actually *do* occur at the time of execution.

- **Opportunity for automatic repair.** Finally, only dynamic analysis provides the opportunity to fix the problem on the fly without any user intervention. This is especially appropriate in the case of a matching method pair when the second method call is missing. While we have not implemented automatic "pattern repair" in DynaMine, we believe it to be a fruitful future research direction. However, care must be taken not to perform the repair action more than once, as most such actions are not idempotent.

While we believe that dynamic analysis is more appropriate than static analysis for the problem at hand, a serious shortcoming of dynamic analysis is its lack of coverage.

In fact, in our dynamic experiments, we have managed to find runtime use cases for some, but not all of our mined patterns. Another concern is that a workload selection may significantly influence how patterns are classified by DynaMine. In our experiments with Eclipse and jEdit we were careful to exercise common functions of both applications that represent hot paths through the code and thus contain errors that may manifest at runtime often. However, we likely have missed error patterns that occur on exception paths that were not hit at runtime.

In addition to the inherent lack of coverage, another factor that reduced the number of patterns available for checking at runtime was that Eclipse contains quite a bit of platform-specific code. This code is irrelevant unless the pattern is located in the portion of the code specific to the execution platform.

Another way to gain some insight into which pattern is more likely is by resorting to static analysis. One approach is to look at flow-sensitive information that is computed statically. If we can conclude that there is a dominance or a post-dominance relation between a set of method calls, that will help us in establishing a pattern. However, doing so interprocedurally is a difficult task.

Another possibility is to use a static checker and examine its results. The intuition is that a pattern that results in a large number of violations is somewhat unlikely.

Williams et al. propose [47] a different interaction between the the mining and analysis stages. They mine information from CVS histories to determine what functions have been "fixed" and use this information to propagate those functions to the top in the static analysis stage. The notion of "being fixed" is pretty rigid, however: they developed a heuristic for the return value checker that does simple syntactic analysis of the relevant change. Notice that unless bug database information can be easily correlated with revision history data, detecting whether a change is a bug fix is a difficult task in general. While "one-line change" heuristic seems to work pretty well in practice, many bug fixes are missed.

```
C1.java:
            class C1 {
                void foo(){
                    ...
+                   workspace.getWidget().addListener();
                    ...
                }
            }

C2.java:
            class C2 {
                void bar(){
                    ...
+                   widget.removeListener();
                    ...
                }
            }
```

Figure 4: Cross-file check-in containing of two lines spread across two different files.

3.2 Amount of User Involvement

Our system currently requires user involvement at the following two stages: first, the candidate patterns need to be formulated by the user based on the methods mined from revision history repositories. Most of the patterns considered in this paper are relatively simple and can be captured with a state machine or, in the more complex cases, a grammar.

However, the need for human involvement can be reduced if we fix the paradigm, i.e. "state machines only." Then, at runtime, we can consider all potential state machines and rank them based on how many times each machine reaches a success state.

For example, for the pair

$$\{\texttt{addListener}, \texttt{removeListener}\}$$

the two machines that are ordered versions of this pair are possible. Of course, other machines, such as the one that requires `addListener` to be called three times, followed by `removeListener`, followed by another call to `addListener` is possible. While an infinite number of potential machines exist, based on our experience, we can often just limit ourselves to permutations of the methods involved.

3.3 Granularity of Mined Information

Our approach to interpreting source code information found in revision repositories is very shallow. We are oblivious to even the type information found in the program, replying only on the syntactic notion of what a method call is. We also use the notion of a common prefix, as described in Section 4.2.2, which is fully oblivious to aliasing information.

In other words, if we have two methods calls, $\texttt{p}_1\texttt{.foo}$ and $\texttt{p}_2\texttt{.bar}$, and we know that $\texttt{p}_1$ and $\texttt{p}_2$ are aliases for the same heap object, as determined by a pointer analysis, the information we mine will be more complete. However, it will also be more noisy because of pointer analysis imprecision.

Having access to parsed abstract syntax trees would enable some sort of analysis, however, the amount of analysis that can be done locally is somewhat limited (pointer analysis is typically global). Having aliasing information as well as parse trees for all revisions of a particular project would enable us to consider patterns that span multiple files. For instance, for the pattern $\{\texttt{addListener}, \texttt{removeListener}\}$, the check-in shown in Figure 4 will contribute to the pattern, assuming access paths $\texttt{workspace.getWidget}()$ in class `C1` and `widget` in class `C2` may refer to the same object.

Finally, another option for the mining strategy is to pay attention to *deletions* as well as additions. It is likely, however, that the number of deletions observed will be fewer.

4 Mining Usage Patterns

In this section we describe our mining approach. We start by providing the terms we use in our discussion of mining. Next we lay out our general algorithmic approach that is based on the Apriori algorithm [1, 30] that is commonly used in data mining for applications such as market basket analysis. The algorithm uses a set of *transactions* such as store item purchases as its input and produces as its output (a) frequent purchasing patterns ("items X, Y, and Z are purchased together") and (b) strong association rules ("a person who bought item X is likely to buy item Y").

However, the classical Apriori algorithm has a serious drawback. The algorithm runtime can be exponential in the number of items. Our "items" are names of individual methods in the program. For Eclipse, which contain 59,929 different methods, calls to which are inserted, scalability is a real concern. To improve the scalability of our approach and to reduce the amount of noise, we employ a number of filtering strategies described in Section 4.2 to reduce the number of viable patterns Apriori has to consider. Furthermore, Apriori does not rank the patterns it returns. Since even with filtering, the number of patterns returned is quite high, we apply several ranking strategies described in Section 4.3 to the patterns we mine. We start our discussion of the mining approach by defining some terminology used in our algorithm description.

Definition 4.1 A *usage pattern* $U = \langle M, S \rangle$ is defined as a set of methods M and a specification S that defines how the methods should be invoked. A *static usage pattern* is present in the source if calls to all methods in M are located in the source and are invoked in a manner consistent with S. A *dynamic usage pattern* is present in a program execution if a sequence of calls to methods M is made in accordance with the specification S.

The term "specification" is intentionally open-ended because we want to allow for a variety of pattern types to be defined. Revision histories record method calls that

have been inserted together and we shall use this data to mine for method sets M. The fact that several methods are correlated does not define the nature of the correlation. Therefore, even though the exact pattern may be obvious given the method names involved, it is generally quite difficult to *automatically* determine the specification S by considering revision history data only and human input is required.

Definition 4.2 For a given source file revision, a *transaction* is a set of methods, calls to which have been inserted.

Definition 4.3 The *support count* of a usage pattern $U = \langle M, S \rangle$ is the number of transactions that contains all methods in M.

In the example in Figure 2 the support count for {`addListener`, `removeListener`} is 3. The changes to `Qux.java` do not contribute to the support count because the pattern is distributed across two revisions.

Definition 4.4 An *association rule* $A \Rightarrow B$ for a pattern $U = \langle M, S \rangle$ consists of two non-empty sets A and B such that $M = A \cup B$.

For a pattern $U = \langle M, S \rangle$ there exist $2^{|M|} - 2$ possible association rules. An association rule $A \Rightarrow B$ is interpreted as follows: whenever a programmer inserts calls to all methods in A, she also insert the calls of all methods in B. Obviously, such rules are not always true. They have a probabilistic meaning.

Definition 4.5 The *confidence* of an association rule $A \Rightarrow B$ is defined as the the conditional probability $P(B|A)$ that a programmer inserts the calls in B, given the condition she has already inserted the calls in A.

The confidence indicates the *strength* of a rule. However, we are more interested in the patterns than in association rules. Thus, we rank patterns by the confidence values of their association rules.

4.1 Basic Mining Algorithm

A classical approach to computing patterns and association rules is the Apriori algorithm [1, 30]. The algorithm takes a *minimum support count* and a *minimum confidence* as parameters. We call a pattern *frequent* if its support is above the minimum support count value. We call an association rule *strong* if its confidence is above the minimum confidence value. Apriori computes (a) the set P of all frequent patterns and (b) the set R of all strong association rules in two phases:

1. The algorithm iterates over the set of transactions and forms patterns from the method calls that occur in the same transaction. A pattern can only be frequent when its subsets are frequent and patterns are expanded in each iteration. Iteration continues until a fixed point is reached and the final set of frequent patterns P is produced.

2. The algorithm computes association rules from the patterns in P. From each pattern $p \in P$ and every method set $q \subseteq p$ such that $p, q \neq \emptyset$, the algorithm creates an association rule of the form $p \setminus q \Rightarrow q$. All rules for a pattern have the same support count, but different confidence values. Strong association rules $p \setminus q \Rightarrow q$ are added to the final set of rules R.[2]

In Sections 4.2 and 4.3 below we describe how we adapt the classic Apriori approach to improve its scalability and provide a ranking of the results.

4.2 Pattern Filtering

The running time of Apriori is greatly influenced by the number of patterns is has to consider. While the algorithm uses thresholds to limit the number of patterns that it outputs in P, we employ some filtering strategies that are specific to the problem of revision history mining. Another problem is that these thresholds are not always adequate for keeping the amount of noise down. The filtering strategies described below greatly reduce the running time of the mining algorithm *and* significantly reduce the amount of noise it produces.

4.2.1 Considering a Subset of Method Calls Only

Our strategy to deal with the complexity of frequent pattern mining is to ignore method calls that either lead to no usage patterns or only lead to obvious ones such as $\{$`hasNext`, `next`$\}$.

- **Ignoring initial revisions**. We do not treat initial revisions of files as additions. Although they contain many usage patterns, taking initial check-ins into account introduces more incidental patterns, i.e. noise, than patterns that are actually useful.
- **Last call of a sequence**. Given a call sequence $c_1().c_2() \ldots c_n()$ included as part of a repository change, we only take the final call $c_n()$ into consideration. This is due to the fact that in Java code, a sequence of "accessor" methods is common and typically only the last call mutates the program environment. Calls like

  `ResourcesPlugin.getPlugin().getLog().log()`

  in Eclipse are quite common and taking intermediate portions of the call into account will contribute to noise in the form of associating the intermediate getter calls. Such patterns are not relevant for our purposes, however, they are well-studied and are best mined from a snapshot of a repository rather than from its history [32, 33, 38].
- **Ignoring common calls**. To further reduce the amount of noise, we ignore some very common method calls, such as the ones listed in Figure 5; in practice, we ignore method calls that were added more than 100 times. These methods tend to get intermingled with real usage patterns, essentially causing noisy, "overgrown" ones to be formed.

[2] $\setminus$ is used in the rest of the paper to denote set difference.

Method name	Number of additions	Method name	Number of additions
`equals`	9,054	`toString`	4,197
`add`	6,986	`getName`	3,576
`getString`	5,295	`append`	3,524
`size`	5,118	`iterator`	3,340
`get`	4,709	`length`	3,339

Figure 5: The most frequently inserted method calls.

4.2.2 Considering Small Patterns Only

Generally, patterns that consist of a large number of methods are created due to noise. Another way to reduce the complexity and the amount of noise is to reduce the scope of mining to *small* patterns only. We employ a combination of the following two strategies.

- **Fine-grained transactions**. As mentioned in Section 4.1, Apriori relies on transactions that group related items together. We generally have a choice between using *coarse-grained* or *fine-grained* transactions. Coarse-grained transactions consist of all method calls added in a single revision. Fine-grained transactions additionally group calls by the access path. In Figure 2, the coarse-grained transaction corresponding to revision 1.23 of `Baz.java` is further subdivided into three fine-grained transactions for objects `o3`, `list`, and `iter`. An advantage of fine-grained transactions is that they are smaller, and thus make mining more efficient. The reason for this is that the runtime heavily depends on the size and number of frequent patterns, which are restricted by the size of transactions. Fine-grained transactions also tend to reduce noise because processing is restricted to a common prefix. However, we may miss patterns containing calls with different prefixes, such as pattern {`iterator`, `hasNext`, `next`} in Figure 2.
- **Mining method pairs**. We can reduce the the complexity even further if we mine the revision repository only for method pairs instead of patterns of arbitrary size. This technique has frequently been applied to software evolution analysis and proved successful for finding evolutionary coupling, etc. [19, 20, 52]. While very common, method pairs can only express relatively simple usage patterns.

4.3 Pattern Ranking

Even when filtering is applied, the Apriori algorithm yields many frequent patterns. However, not all of them turn out to be good usage patterns in practice. Therefore, we use several ranking schemes when presenting the patterns we discovered to the user for review.

4.3.1 Standard Ranking Approaches

Mining literature provides a number of standard techniques we use for pattern ranking. Among them are the pattern's (1) support count, (2) confidence, and (3) strength, where

the strength of a pattern is defined as following.

Definition 4.6 The *strength* of pattern p is the number of strong association rules in R of the form $p \setminus q \Rightarrow q$ where $q \subset p$, both p and q are frequent patterns, and $q \neq \emptyset$.

For our experiments, we rank patterns lexicographically by their strength and support count. However, for matching method pairs $\langle a, b \rangle$ we use the product of confidence values $conf(a \Rightarrow b) \times conf(b \Rightarrow a)$ instead of the strength because the continuous nature of the product gives a more fine-grained ranking than the strength; the strength only takes the values of 0, 1, and 2 for pairs. The advantage of products over sums is that pairs where both confidence values are high are favored. In the rest of the paper we refer to the ranking that follows classical data mining techniques as *regular ranking*.

4.3.2 Corrective Ranking

While the ranking schemes above can generally be applied to any data mining problem, we have come up with a measure of a pattern's importance that is specific to mining revision histories. Observation 2.3 is the basis of the metric we are about to describe. A check-in may only add *parts* of a usage pattern to the repository. Generally, this is a problem for the classic Apriori algorithm, which prefers patterns, all parts of which are "seen together". However, we can leverage these incomplete patterns when we realize that they often represent bug fixes.

A recent study of the dynamic of small repository changes in large software systems performed by Purushothaman et al. sheds a new light on this subject [36]. Their paper points out that almost 50% of all repository changes were small, involving less than 10 lines of code. Moreover, among one-line changes, less than 4% were likely to cause a later error. Furthermore, only less than 2.5% of all one-line changes were *perfective* changes that add functionality, rather than *corrective* changes that correct previous errors. These numbers imply a very strong correlation between one-line changes and bug corrections or fixes.

We use this observation to develop a *corrective ranking* that extends the ranking that is used in classical data mining. For this, we identify one-line fixes and mark method calls that were added at least once in such a fix as *fixed*. In addition to the measures used by regular ranking, we then additionally rank by the number of fixed methods calls which is used as the first lexicographic category. As discussed in Section 6, patterns with a high corrective rank result in more dynamic violations than patterns with a high regular rank.

4.4 Locating Added Method Calls

In order to speed-up the mining process, we pre-process the revision history extracted from CVS and store this information in a general-purpose database; our techniques are further described in Zimmermann et al. [51]. The database stores method calls that have been inserted for each revision. To determine the calls inserted between two revisions r_1 and r_2, we build abstract syntax trees (ASTs) for both r_1 and r_2 and compute the set of all calls C_1 and C_2, respectively, by traversing the ASTs. $C_2 \setminus C_1$ is the set of inserted calls between r_1 and r_2.

Application	Lines of code	Source files	Java classes	CVS revisions	Method calls inserted	Methods called in inserts	Developers checking in	CVS history since
Eclipse	2,924,124	19,115	19,439	2,837,854	465,915	59,929	122	May 2^{nd}, 2001
jEdit	714,715	3,163	6,602	144,495	56,794	10,760	92	Jan 15^{th}, 2000

Figure 6: Summary of information about our benchmark applications.

Unlike Williams and Hollingsworth [46, 47] our approach does not build snapshots of a system. As they point out such interactions with the build environment (compilers, makefiles) are extremely difficult to handle and result in high computational costs. Instead we analyze only the differences between single revisions. As a result our preprocessing is cheap and platform- and compiler-independent; the drawback is that types cannot be resolved because only one file is investigated. In order to avoid noise that is caused by this, we additionally identify methods by the count of arguments.

5 Checking Patterns at Runtime

In this section we describe our dynamic approach for checking the patterns discovered through revision history mining.

5.1 Pattern Selection & Instrumentation

To aid with the task of choosing the relevant patterns, the user is presented with a list of mined patterns in an Eclipse view such as the one shown in Figure 9. The list of patterns may be sorted and filtered based on various ranking criteria described in Section 4.3 to better target user efforts. Human involvement at this stage, however, is optional, because the user may decide to dynamically check *all* the patterns discovered through revision history mining.

After the user selects the patterns of interest, the list of relevant methods for each of the patterns is generated and passed to the instrumenter. We use JBoss AOP [9], an aspect-oriented framework to insert additional "bookkeeping" code at the method calls relevant for the patterns. However, the task of pointcut selection is simplified for the user by using a graphical interface. In addition to the method being called and the place in the code where the call occurs, values of all actual parameters are also recorded.

Our instrumenter inserts Java bytecode at call sites to each relevant method that outputs a *dynamic event descriptor* $\langle T, E, L \rangle$ each time a relevant call site is hit at runtime, where:

1. T is a unique timestamp for the method call.
2. E is the environment that contains values of each object passed into or returned by the method call.
3. L is the source code locations of the relevant call site.

	Method pair $\langle a, b \rangle$		Confidence			Support	Dynamic		Static		Type
	Method a	**Method** b	$conf$	$conf_{ab}$	$conf_{ba}$	$count$	v	e	v	e	
Corrective Ranking											
Eclipse	`NewRgn`	`DisposeRgn`	0.76	0.92	0.82	49					
(16 pairs)	`kEventControlActivate`	`kEventControlDeactivate`	0.69	0.83	0.83	5					
	`addDebugEventListener`	`removeDebugEventListener`	0.61	0.85	0.72	23	4	1	4	1	Unlikely
	`beginTask`	`done`	0.60	0.74	0.81	493	332	759	41	28	Unlikely
	`beginRule`	`endRule`	0.60	0.80	0.74	32	7	0	4	0	Usage
	`suspend`	`resume`	0.60	0.83	0.71	5					
	`NewPtr`	`DisposePtr`	0.57	0.82	0.70	23					
	`addListener`	`removeListener`	0.57	0.68	0.83	90	143	140	35	29	Error
	`register`	`deregister`	0.54	0.69	0.78	40	2,854	461	17	90	Error
	`malloc`	`free`	0.47	0.68	0.68	28					
	`addElementChangedListener`	`removeElementChangedListener`	0.42	0.73	0.57	8	6	1	1	1	Error
	`addResourceChangeListener`	`removeResourceChangeListener`	0.41	0.90	0.46	26	27	1	21	1	Usage
	`addPropertyChangeListener`	`removePropertyChangeListener`	0.40	0.54	0.73	140	1,864	309	54	31	Error
	`start`	`stop`	0.39	0.59	0.65	32	69	18	20	9	Error
	`addDocumentListener`	`removeDocumentListener`	0.36	0.64	0.56	29	38	2	14	2	Usage
	`addSyncSetChangedListener`	`removeSyncSetChangedListener`	0.34	0.62	0.56	24					
jEdit	`addNotify`	`removeNotify`	0.60	0.77	0.77	17	3	0	3	0	Unlikely
(8 pairs)	`setBackground`	`setForeground`	0.57	0.67	0.86	12	75	175	5	5	Unlikely
	`contentRemoved`	`contentInserted`	0.51	0.71	0.71	5	17	11	7	5	Error
	`setInitialDelay`	`start`	0.40	0.80	0.50	4	0	32	0	2	Unlikely
	`registerErrorSource`	`unregisterErrorSource`	0.28	0.45	0.62	5					
	`start`	`stop`	0.20	0.39	0.52	33	83	98	10	13	Error
	`addToolBar`	`removeToolBar`	0.18	0.60	0.30	6	24	43	5	5	Error
	`init`	`save`	0.09	0.40	0.24	31					
(24 pairs)	**Subtotals for the corrective ranking scheme:**						**5,546**	**2,051**	**241**	**222**	3 U, 8 E

Figure 7: Effectiveness of corrective ranking. Matching method pairs discovered through CVS history mining. The support count is $count$, the confidence for $\{a\} \Rightarrow \{b\}$ is $conf_{ab}$, for $\{b\} \Rightarrow \{a\}$ it is $conf_{ba}$. The pairs are ordered by $conf = conf_{ab} \times conf_{ba}$. In the last column, usage and error patterns are abbreviated as "U" and "E", respectively. Empty cells represent patterns that have not been observed at runtime.

	Method pair $\langle a, b \rangle$		Confidence			Support	Dynamic		Static		Type
	Method a	**Method** b	$conf$	$conf_{ab}$	$conf_{ba}$	$count$	v	e	v	e	
Regular Ranking											
Eclipse	`createPropertyList`	`reapPropertyList`	1.00	1.00	1.00	174					
(15 pairs)	`preReplaceChild`	`postReplaceChild`	1.00	1.00	1.00	133	40	0	26	0	Usage
	`preLazyInit`	`postLazyInit`	1.00	1.00	1.00	112					
	`preValueChange`	`postValueChange`	1.00	1.00	1.00	46	63	2	11	2	Usage
	`addWidget`	`removeWidget`	1.00	1.00	1.00	35	2,507	16	26	6	Usage
	`stopMeasuring`	`commitMeasurements`	1.00	1.00	1.00	15					
	`blockSignal`	`unblockSignal`	1.00	1.00	1.00	13					
	`Hlock`	`HUnLock`	1.00	1.00	1.00	9					
	`addInputChangedListener`	`removeInputChangedListener`	1.00	1.00	1.00	9					
	`preRemoveChildEvent`	`postAddChildEvent`	1.00	1.00	1.00	8	0	171	0	3	Unlikely
	`progressStart`	`progressEnd`	1.00	1.00	1.00	8					
	`CGContextSaveGState`	`CGContextRestoreGState`	1.00	1.00	1.00	7					
	`addInsert`	`addDelete`	1.00	1.00	1.00	7					
	`annotationAdded`	`annotationRemoved`	1.00	1.00	1.00	7	0	10	0	4	Unlikely
	`OpenEvent`	`fireOpen`	1.00	1.00	1.00	7	3	0	1	0	Unlikely
jEdit	`readLock`	`readUnlock`	1.00	1.00	1.00	16	8,578	0	14	0	Usage
(13 pairs)	`setHandler`	`parse`	1.00	1.00	1.00	6	12	0	8	0	Usage
	`addTo`	`removeFrom`	1.00	1.00	1.00	5					
	`execProcess`	`ssCommand`	1.00	1.00	1.00	4					
	`freeMemory`	`totalMemory`	1.00	1.00	1.00	4	95	0	2	0	Usage
	`lockBuffer`	`unlockBuffer`	1.00	1.00	1.00	4					
	`writeLock`	`writeUnlock`	0.85	1.00	0.85	11	38	0	8	0	Usage
	`allocConnection`	`releaseConnection`	0.83	1.00	0.83	5					
	`getSubregionOfOffset`	`xToSubregionOffset`	0.80	0.80	1.00	4					
	`initTextArea`	`uninitTextArea`	0.80	0.80	1.00	4					
	`undo`	`redo`	0.69	0.83	0.83	5	0	4	0	1	Unlikely
	`setSelectedItem`	`getSelectedItem`	0.37	0.50	0.73	11	7	17	7	7	Unlikely
	`addToSelection`	`setSelection`	0.29	0.57	0.50	4	12	27	1	9	Unlikely
(28 pairs)	**Subtotals for the regular ranking scheme:**						**11,355**	**247**	**104**	**32**	7 U
(52 pairs)	**Overall totals:**						**16,901**	**2,298**	**245**	**254**	10 U, 8 E

Figure 8: Effectiveness of regular ranking.

5.2 Post-processing Dynamic Traces

The trace produced in the course of a dynamic run are post-processed to produce the final statistics about the number of times each pattern is followed and the number of times it is violated. We decided in favor of off-line post-processing because some patterns are rather difficult and sometimes impossible to match with a fully online approach. In order to facilitate the task of post-processing in practice, DynaMine is equipped with checkers to look for matching method pairs and state machines. Users who wish to create checkers for more complex patterns can do so through a Java API exposed by DynaMine that allows easy access to runtime events.

Dynamically obtained results for matching pairs and state machines are exported back into Eclipse for review. The user can browse through the results and ascertain which of the patterns she thought must hold do actually hold at runtime. Often, examining the dynamic output of DynaMine allows the user to correct the initial pattern and re-instrument.

5.2.1 Dynamic Interpretation of Patterns

While it may be intuitively obvious what a given coding pattern means, what kind of *dynamic behavior* is valid may be open to interpretation, as illustrated by the following example.

Example 2. Consider a matching method pair $\langle$`beginOp`, `endOp`$\rangle$ and a dynamic call sequence

$$seq = \texttt{o.beginOp}() \ldots \texttt{o.beginOp}() \ldots \texttt{o.endOp}().$$

Obviously, a dynamic execution consisting of a sequence of calls `o.beginOp`$() \ldots$ `o.endOp`$()$ follows the pattern. However, execution sequence seq probably represents a pattern violation.

While declaring seq a violation may appear quite reasonable on the surface, consider now an implementation of method `beginOp` that starts by calling `super.beginOp`$()$. Now seq is the dynamic call sequence that results from a static call to `o.beginOp` followed by `o.endOp`; the first call to `beginOp` comes from the static call to `beginOp` and the second comes from the call to `super`. However, in this case seq may be a completely reasonable interpretation of this coding pattern. □

As this example shows, there is generally no obvious mapping from a coding pattern to a dynamic sequence of events. As a result, the number of dynamic pattern matches and mismatches is interpretation-dependent. Errors found by DynaMine at runtime can only be considered such with respect to a particular dynamic interpretation of patterns. Moreover, while violations of application-specific patterns found with our approach represent *likely* bugs, they cannot be claimed as definite bugs without carefully studying the effect of each violation on the system.

In the implementation of DynaMine, to calculate the number of times each pattern is validated and violated we match the unqualified names of methods applied to a given dynamic object. Fortunately, complete information about the object involved is available at runtime, thus making this sort of matching possible. For patterns that involve only one object, we do not consider method arguments when performing a match: our

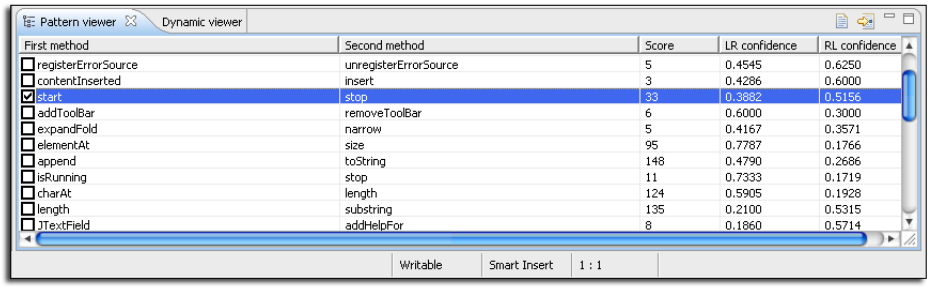

Figure 9: DynaMine pattern selection view.

goal is to have a dynamic matcher that is as automatic as possible for a given type of pattern, and it is not always possible to automatically determine which arguments have to match for a given method pair. For complex patterns that involve more than one object and require user-defined checkers, the trace data saved by DynaMine contains information allows the relevant call arguments to be matched.

5.2.2 Dynamic vs Static Counts

A single pattern violation at runtime involves one or more objects. We obtain a *dynamic count* by counting how many object combinations participated in a particular pattern violation during program execution. Dynamic counts are highly dependent on how we use the program at runtime and can be easily influenced by, for example, recompiling a project in Eclipse multiple times.

Moreover, dynamic error counts are not representative of the work a developer has to do to fix an error, as many dynamic violations can be caused by the same error in the code. To provide a better metric on the number of errors found in the application code, we also compute a *static count*. This is done by mapping each method participating in a pattern to a static call site and counting the number of unique call site combinations that are seen at runtime. Static counts are computed for validated and violated patterns.

5.2.3 Pattern Classification

We use runtime information on how many times each pattern is validated and how many times it is violated to classify the patterns. Let v be the number of validated instances of a pattern and e be the number of its violations. The constants used in the classification strategy below were obtained empirically to match our intuition about how patterns should be categorized. However, clearly, ours is but one of many potential classification approaches.

We define an error threshold $\alpha = min(v/10, 100)$. Based on the value of α, patterns can be classified into the following categories:

1. **Likely usage patterns**: patterns with a sufficiently high support that are mostly validated with relatively few errors
   ($e < \alpha \wedge v > 5$).

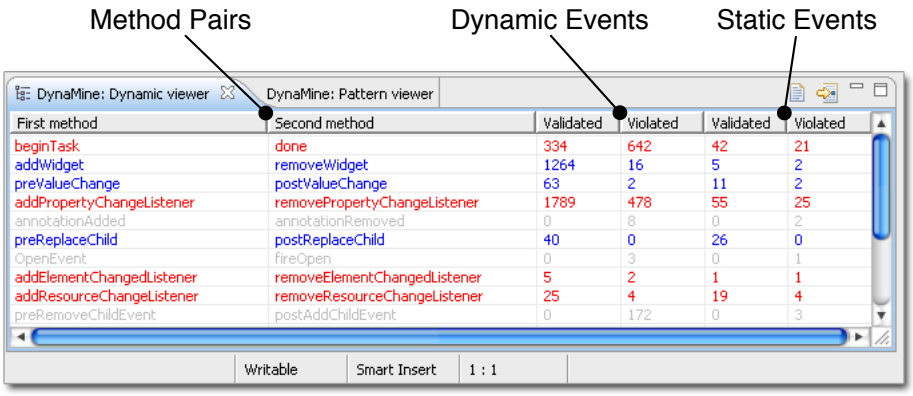

Figure 10: DynaMine dynamic results view. Error patterns are shown in red. Usage patterns are shown in blue. Unlikely patterns are grayed out.

2. **Likely error patterns**: patterns that have a significant number of validated cases as well as a large number of violations
   ($\alpha \leq e \leq 2v \wedge v > 5$).

3. **Unlikely patterns**: patterns that do not have many validated cases or cause too many errors to be usage patterns
   ($e > 2v \vee v \leq 5$).

In DynaMine, a categorization of patterns is presented to the user in an Eclipse view, as shown in Figure 10. The patterns are color-coded in the Eclipse view to represent their type, which blue being a likely usage pattern, red being a likely error pattern, and gray being an unlikely pattern.

6 Experimental Results

In this section we discuss our practical experience of applying DynaMine to real software systems. Section 6.1 describes our experimental setup; Section 6.2 evaluates the results of both our patterns mining and dynamic analysis approaches.

6.1 Experimental Setup

We have chosen to perform our experiments on Eclipse [11] and jEdit [35], two very large open-source Java applications; in fact, Eclipse is one of the largest Java projects

Figure 11: Pattern evaluation scale.

ever created. A summary of information about the benchmarks is given in Figure 6. For each application, the number of lines of code, source files, and classes is shown in column 2–4. Both applications are known for being highly extensible and having a large number of plugins available; in fact, much of Eclipse itself is implemented as a set of plugins. In addition to these standard metrics that reflect the size of the benchmarks, we show the number of revisions in each CVS repository in column 5, the number of inserted calls in column 6, and the number of distinct methods that were called in column 7. Both projects have a significant number of individual developers working on them, as evidenced by the numbers in column 8. The date of the first revision is presented in column 9.

6.1.1 Mining Setup

When we performed the pre-processing on Eclipse and jEdit, it took about four days to fetch all revisions over the Internet because the complete revision data is about 6GB in size and the CVS protocol is not well-suited for retrieving large volumes of history data. Computing inserted methods by analyzing the ASTs and storing this information in a database takes about a day on a Powermac G5 2.3 Ghz dual-processor machine with 1 GB of memory.

Once the pre-processing step was complete, we performed the actual data mining. Without any of the optimizations described in Sections 4.2 and 4.3, the mining step does not complete even in the case jEdit, not to mention Eclipse. Among the optimizations we apply, the biggest time improvement and noise reduction is achieved by disregarding common method calls, such as `equals`, `length`, etc. With *all* the optimizations applied, mining becomes orders of magnitude faster, usually only taking several minutes.

6.1.2 Dynamic Setup

Because the incremental cost of checking for additional patterns at runtime is generally low, when reviewing the patterns in Eclipse for inclusion in our dynamic experiments, we were fairly liberal in our selection. We would usually either just look at the method names involved in the pattern or briefly examine a few usage cases. We believe that this strategy is realistic, as we cannot expect the user to spend hours pouring over the patterns. To obtain dynamic results, we ran each application for several minutes on a Pentium 4 machine running Linux, which typically resulted in several thousand dynamic events being generated.

6.2 Discussion of the Results

Overall, 32 out of 56 (or 57%) patterns were hit at runtime. Furthermore, 21 out of 32 (or 66%) of these patterns turned out to be either usage or error patterns. The fact that two thirds of all dynamically encountered patterns were likely patterns demonstrates the power of our mining approach.

In this section we discuss the categories of patterns briefly described in Section 2 in more detail.

6.2.1 Matching Method Pairs

The simplest and most common kind of a pattern detected with our mining approach is one where two different methods of the same class are supposed to match precisely in execution. Many of known error patterns in the literature such as ⟨`fopen`, `fclose`⟩ or ⟨`lock`, `unlock`⟩ fall into the category of function calls that require exact matching: failing to call the second function in the pair or calling one of the functions twice in a row is an error.

Figure 8 lists matching pairs of methods discovered with our mining technique. The methods of a pair $\langle a, b \rangle$ are listed in the order they are supposed to be executed, e.g., a should be executed before b. For brevity, we only list the names of the method; full method names that include package names should be easy to obtain. A quick glance at the table reveals that many pairs follow a specific naming strategy such as `pre`–`post`, `add`–`remove`, `begin`–`end`, and `enter`–`exit`. These pairs could have been discovered by simply pattern matching on the method names. Moreover, looking at method pairs that use the same prefixes or suffixes is an obvious extension of our technique.

However, a significant number of pairs have less than obvious names to look for, including ⟨`HLock`, `HUnlock`⟩, ⟨`progressStart`, `progressEnd`⟩, and ⟨`blockSignal`, `unblockSignal`⟩. Finally, some pairs are very difficult to recognize as matching method pairs and require a detailed study of the API to confirm, such as ⟨`stopMeasuring`, `commitMeasurements`⟩, ⟨`suspend`, `resume`⟩, etc.

Figure 8 summarizes dynamic results for matching pairs. The table provides dynamic and static counts of validated and violated patterns as well as a classification into usage, error, and unlikely patterns. Below we summarize some observations about the data. About a half of all method pair patterns that we selected from the filtered mined results were confirmed as likely patterns, out of those 5 were usage patterns and 9 were error patterns. Many more potentially interesting matching pairs become available if we consider lower support counts; for the experiments we have only considered patterns with a support of four or more.

Several characteristic pairs are described below. Both locking pairs in jEdit ⟨`writeLock`, `writeUnlock`⟩ and ⟨`readLock`, `readUnlock`⟩ are excellent usage patterns with no violations. ⟨`contentInserted`, `contentRemoved`⟩ is not a good pattern despite the method names: the first method is triggered when text is added in an editor window; the second when text is removed. Clearly, there is no reason why these two methods have to match. Method pair ⟨`addNotify`, `removeNotify`⟩ is perfectly matched, however, its support is not sufficient to declare it a usage pattern. A somewhat unusual kind of matching methods that at first we thought was caused by noise in the data consists of a constructor call followed by a method call, such as the pair ⟨`OpenEvent`, `fireOpen`⟩. This sort of pattern indicates that all objects of type `OpenEvent` should be "consumed" by passing them into method `fireOpen`. Violations of this pattern may lead to resource and memory leaks, a serious problem in long-running Java programs such as Eclipse, which may be open at a developer's desktop for days.

Overall, corrective ranking was significantly more effective than regular ranking schemes that are based on the product of confidence values. The top half of the table that addresses patterns obtained with corrective ranking contains 24 matching method

pairs; the second half that deals with the patterns obtained with regular ranking contains 28 pairs. Looking at the subtotals for each ranking scheme reveals 241 static validating instances vs only 104 for regular ranking; 222 static error instances are found vs only 32 for regular ranking. Finally, 11 pairs found with corrective ranking were dynamically confirmed as either error or usage patterns vs 7 for regular ranking. This confirms our belief that corrective ranking is more effective.

6.2.2 State Machines

In many of cases, the order in which methods are supposed to be called *on a given object* can easily be captured with a finite state machine. Typically, such state machines must be followed precisely: omitting or repeating a method call is a sign of error. The fact that state machines are encountered often is not surprising: state machines are the simplest formalism for describing the object life-cycle [40]. Matching method pairs are a specific case of state machines, but there are other prominent cases that involve *more that two methods*, which are the focus of this section.

An example of state machine usage comes from class `org.eclipse.jdt.internal.formatter.Scribe` in Eclipse responsible for pretty-printing Java source code. Method `exitAlignment` is supposed to match an earlier `enterAlignment` call to preserve consistency. Typically, method `redoAlignment` that tries to resolve an exception caused by the current `enterAlignment` would be placed in a `catch` block and executed optionally, only if an exception is raised. The regular expression

$$\texttt{o.enterAlignment o.redoAlignment}^{?}\ \texttt{o.exitAlignment}$$

summarizes how methods of this class are supposed to be called on an object `o` of type `Scribe`. In our dynamic experiments, the pattern matched 885 times with only 17 dynamic violations that correspond to 9 static violations, which makes this an excellent usage pattern.

Another interesting state machine below is found based on mining jEdit. Methods `beginCompoundEdit` and `endCompoundEdit` are used to group editing operations on a text buffer together so that undo or redo actions can be later applied to them at once.

$$\begin{array}{l}\texttt{o.beginCompoundEdit}()\\ \qquad (\texttt{o.insert}(...) \mid \texttt{o.remove}(...))^{+}\\ \texttt{o.endCompoundEdit}()\end{array}$$

A dynamic study of this pattern reveals that (1) methods `beginCompoundEdit` and `endCompoundEdit` are *perfectly* matched in all cases; (2) 86% of calls to `insert`/`remove` are *within* a compound edit; (3) there are three cases of several $\langle$`begin`$-$, `endCompoundEdit`$\rangle$ pairs that have no `insert` or `remove` operations between them. Since a compound edit is established for a reason, this shows that our regular expression most likely does not fully describe the life-cycle of a `Buffer` object. Indeed, a detailed study of the code reveals some other methods that may be used within a compound edit. Subsequently adding these methods to the pattern and re-instrumenting the jEdit led to a new pattern that fully describes the `Buffer` object's life-cycle.

Precisely following the order in which methods must be invoked is common for C interfaces [14], as represented by functions that manipulate files and sockets. While such dependency on call order is less common in Java, it still occurs in programs that have low-level access to OS data structures. For instance, methods `PmMemCreateMC`, `PmMemFlush`, and `PmMemStop`, `PmMemReleaseMC` declared in `org.eclipse.swt.OS` in Eclipse expose low-level memory context management routines in Java through the use of JNI wrappers. These methods are supposed to be called in order described by the regular expression below:

$$\texttt{OS.PmMemCreateMC}\ (\texttt{OS.PmMemStart}\ \texttt{OS.PmMemFlush}\ \texttt{OS.PmMemStop})?\ \texttt{OS.PmMemReleaseMC}$$

The first and last lines are mandatory when using this pattern, while the middle line is optional. Unfortunately, this pattern only exhibits itself at runtime on certain platforms, so we were unable to confirm it dynamically.

Another similar JNI wrapper found in Eclipse that can be expressed as a state machine is responsible for region-based memory allocation and can be described with the following regular expression:

$$(\texttt{OS.NewPtr} \mid \texttt{OS.NewPtrClear})\ \texttt{OS.DisposePtr}$$

Either one of functions `NewPtr` and `NewPtrClear` can be used to create a new pointer; the latter function zeroes-out the memory region before returning.

Another commonly used pattern that can be captured with a state machine has to do with hierarchical allocation of resources. Objects request and release system resources in a way that is perfectly nested. For instance, one of the patterns we found in Eclipse suggests the following resource management scheme on objects of type component:

$$\texttt{o.createHandle}()\ \texttt{o.register}()\ \texttt{o.deregister}()\ \texttt{o.releaseHandle}()$$

The call to `createHandle` requests an operating system resource for a GUI widget, such as a window or a button; `releaseHandle` frees this OS resource for subsequent use. `register` associates the current GUI object with a `display` data structure, which is responsible for forwarding GUI events to components as they arrive; `deregister` breaks this link.

6.2.3 More Complex Patterns

More complicated patterns, that are concerned with the behavior of more than one object or patterns for which a finite state machine is not expressive enough, are quite widespread in the code base we have considered as well. Notice that approaches that use a restrictive model of a pattern such as matching function calls [15], would not be able to find these complex patterns.

Due to space restrictions, we only describe one complex pattern in detail here, which is motivated by the the code snippet in Figure 12. The lines relevant to the pattern

```
try {
    monitor.beginTask(null, Policy.totalWork);
    int depth = -1;
    try {
        workspace.prepareOperation(null, monitor);
        workspace.beginOperation(true);
        depth = workspace.getWorkManager().beginUnprotected();
        return runInWorkspace(Policy.subMonitorFor(monitor,
            Policy.opWork,
            SubProgressMonitor.PREPEND_MAIN_LABEL_TO_SUBTASK));
    } catch (OperationCanceledException e) {
        workspace.getWorkManager().operationCanceled();
        return Status.CANCEL_STATUS;
    } finally {
        if (depth >= 0)
            workspace.getWorkManager().endUnprotected(depth);
        workspace.endOperation(null, false,
        Policy.subMonitorFor(monitor, Policy.endOpWork));
    }
    } catch (CoreException e) {
        return e.getStatus();
} finally {
    monitor.done();
}
```

Figure 12: Example of workspace operations and locking discipline usage in class `InternalWorkspaceJob` in Eclipse.

are highlighted in bold. Object `workspace` is a runtime representation of an Eclipse workspace, a large complex object that has a specialized transaction scheme for when it needs to be modified. In particular, one is supposed to start the transaction that requires workspace access with a call to `beginOperation` and finish it with `endOperation`.

Calls to `beginUnprotected`() and `endUnprotected`() on a `WorkManager` object obtained from the `workspace` indicate "unlocked" operations on the workspace: the first one releases the workspace lock that is held by default and the second one re-acquires it; the `WorkManager` is obtained for a `workspace` by calling `workspace.getWorkManager`. Unlocking operations should be precisely matched if no error occurs; in case an exception is raised, method `operationCanceled` is called on the `WorkManager` of the current workspace. As can be seen from the code in Figure 12, this pattern involves error handling and may be quite tricky to get right. We have come across this pattern by observing that pairs ⟨`beginOperation`, `endOperation`⟩ and ⟨`beginUnprotected`, `endUnprotected`⟩ are both highly correlated in the code. This pattern is easily described as a context-free language that allows nested matching

brackets, whose grammar is shown below.[3]

$$
\begin{array}{lcl}
S & \rightarrow & O^{\star} \\
O & \rightarrow & \texttt{w.prepareOperation}() \\
 & & \texttt{w.beginOperation}() \\
 & & U^{\star} \\
 & & \texttt{w.endOperation}() \\
U & \rightarrow & \texttt{w.getWorkManager}().\texttt{beginUnprotected}() \\
 & & S \\
 & & \texttt{w.getWorkManager}().\texttt{operationCanceled}()\ ? \\
 & & \texttt{w.getWorkManager}().\texttt{beginUnprotected}()
\end{array}
$$

This is a very strong usage patterns in Eclipse, with 100% of the cases we have seen obeying the grammar above. The nesting of `Workspace` and `WorkManager` operations was usually 3–4 levels deep in practice.

As this example shows, characterizing the pattern with a grammar or some other specification is not an easy task. In DynaMine, this task is delegated to the user. However, restricting the formalism used for describing the pattern such as state machines in Whaley et al. [45] may make it possible to determine the pattern automatically.

7 Extensions

While the patterns DynaMine discovers can be fed into bug detection tools to detect coding effects in code after it has been written, it would be even better to prevent problems from happening before the coding phase is finished. In this section we outline one such extension where mined patterns are translated into coding templates supported by the developer's editor. As a result, common coding mistakes are avoided from the start.

As we observed earlier, matching method pairs represent one of the most commonly used temporal patterns in software. While pairs such as ⟨`fopen`, `fclose`⟩ are widespread in C, as this paper has shown, similar method pairs are present in large Java code bases as well. Eclipse APIs have a number of such methods pairs scattered throughout the code, including ⟨`register`, `unregister`⟩, ⟨`beginCompoundEdit`, `endCompoundEdit`⟩, etc.

A common coding idiom pertaining to using method pairs consists of making sure that the second method is placed within a `finally` block so that the "closing bracket" method is always executed. I.e., given a pair of methods ⟨`A`, `B`⟩, the coding pattern shown in Figure 13 is quite common. To ensure that method `B` is called on *all* execution paths, the call to `B` is placed within the `finally` block. Placing it outside the `finally` block may lead to `B` not being called when an exception is thrown. This coding idiom in Java is explored in more detail in [44].

We take this concept further by creating a set of *coding templates* common to plugin development. According to the Eclipse documentation, "templates are a structured description of coding patterns that reoccur in source code", and thus represent a perfect

[3] S is the grammar start symbol and * is used to represent 0 or more copies of the preceding non-terminal; ? indicates that the preceding non-terminal is optional.

```
try {
    ...
    A(...);
    ...
} finally {
    B(...);
}
```

Figure 13: A typical `try` — `finally` block involving a matching method pair

mechanism for our purposes: whenever the user introduces a call to method `A`, the machinery built into Eclipse is responsible for expanding the template to build the structure shown in Figure 13.

8 Related Work

A vast amount of work has been done in bug detection. C and C++ code in particular is prone to buffer overrun and memory management errors; tools such as PREfix [10] and Clouseau [23] are representative examples of systems designed to find specific classes of bugs (pointer errors and object ownership violations respectively). Dynamic systems include Purify [22], which traps heap errors, and Eraser [39], which detects race conditions. Both of these analyses have been implemented as standard uses of the Valgrind system [34]. Much attention has been given to detecting high-profile software defects in important domains such as operating system bugs [21, 23], security bugs [41, 43], bugs in firmware [25] and errors in reliability-critical embedded systems [7, 8].

Space limitations prohibit us from reviewing a vast body of literature of bug-finding

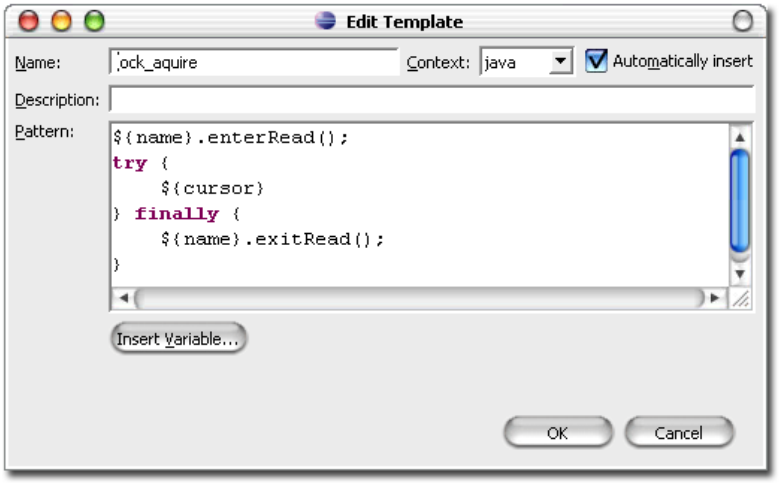

Figure 14: Template creation for ⟨`acquireRead`, `exitRead`⟩ pattern.

techniques. Engler et al. are among the first to point out the need for extracting rules to be used in bug-finding tools [15]. They employ a static analysis approach and statistical techniques to find likely instantiations of pattern templates such as matching function calls. Our mining technique is not a-priori limited to a particular set of pattern templates, however, it is powerless when it comes to patterns that are never added to the repository after the first revision.

Several projects focus on application-specific error patterns, including SABER [37] that deals with J2EE patterns. Their work was motivated by the desire to analyze really large Java systems such as WebSphere. Based on the experience of IBM's developers, they have identified a range of usage patterns in Java AWT, Swing, and EJB code and analyses created to find violations of these patterns. The Metal system [21] addresses the types of bugs in OS code.

PQL is a language that allows one to express and enforce API usage patterns [31]. PQL supports both runtime and static checking as well as a hybrid mode when static analysis removes superfluous runtime checks to reduce the overhead. The need to come up with useful patterns for PQL severed as the original inspiration for our work.

Certain categories of patterns can be gleaned from AntiPattern literature [13, 42], although many AntiPatterns tend to deal with high-level architectural concerns than with low-level coding issues. In the rest of this section, we review literature pertinent to revision history mining and software model extraction.

8.1 Revision History Mining

One of the most frequently used techniques for revision history mining is co-change. The basic idea is that two items that are changed together, are related to one another. These items can be of any granularity; in the past co-change has been applied to changes in modules [19], files [5], classes [6, 20], and functions [50]. Recent research improves on co-change by applying data mining techniques to revision histories [49, 52]. Michail used data mining on the source code of programming libraries to detect reuse patterns, but not for revision histories only for single snapshots [32, 33]. Our work is the first to apply co-change and data mining based on method calls. While Fischer et al. were the first to combine bug databases with dynamic analysis [18], our work is the first that combines the mining of revision histories with dynamic analysis.

The work most closely related to ours is that by Williams and Hollingsworth [46]. They were the first to combine program analysis and revision history mining. Their paper proposes error ranking improvements for a static return value checker with information about fixes obtained from revision histories. Our work differs from theirs in several important ways: they focus on prioritizing or improving existing error patterns and checkers, whereas we concentrate on discovering new ones. Furthermore, we use dynamic analysis and thus do not face high false positive rates their tool suffers from.

Recently, Williams and Hollingsworth also turned towards mining function usage patterns from revision histories [47]. In contrast to our work, they focus only on pairs and do not use their patterns to detect violations.

8.2 Model Extraction

Most work on automatically inferring state models on components of software systems has been done using dynamic analysis techniques. The Strauss system [3] uses machine learning techniques to infer a state machine representing the proper sequence of function calls in an interface.

Dallmeier et al. trace call sequences and correlate sequence patterns with test failures [12]. Whaley et al. [45] hardcode a restricted model paradigm so that probable models of object-oriented interfaces can be easily automatically extracted. Alur et al. [2] generalize this to automatically produce small, expressive finite state machines with respect to certain predicates over an object. Lam et al. use a type system-based approach to statically extract interfaces [26]. Their work is more concerned with high-level system structure rather than low-level life-cycle constraints [40].

Daikon is able to validate correlations between values at runtime and is therefore able to validate patterns [16]. Weimer et al. use exception control flow paths to guide the discovery of temporal error patterns with considerable success [44]; they also provide a comparison with other ing specification mining work.

Perracota uses a runtime analysis to propose a set of temporal properties [48] in a manner similar to Daikon [17]. The resulting properties can then be verified using a theorem prover. In cotrast to Perracota, our approach is designed to be much more lightweight, sidestepping costly static analysis or theorem proving and only requiring dynamic analysis for validation of the candidate properties.

PR-Miner relies on parsing code snapshots and frequent itemset mining to detect common patterns that may include functions, variables, and files citepr-miner. Rules with a low threshold are removed and additional pruning techniques are used to reduce the number of rules used for error checking further. A combination of intra- and inter-procedural analysis is then used to find bugs. However, generating a lot of "candidate" rules does not help in a real-life setting unless the noise is very low. Moreoever, PR-Miner reports a total of 23 bugs and 75 false positives, suggesting that runtime analysis lacking false positives is a good way to proceed.

9 Future Work

DynaMine is one of the first cross-over projects between the areas of revision history mining and bug detection. We see many potential extensions for our work, some of which are listed below:

- Patterns discovered by DynaMine can be used in a variety of bug-finding tools. While whole-program static analysis is expensive, applying a lightweight intraprocedural static approach to the patterns confirmed using dynamic analysis will likely discover interesting errors on rarely executed exceptional paths.

- Extends the set of patterns discovered with DynaMine by simple textual matching. For example, if $\langle$`blockSignal`, `unblockSignal`$\rangle$ is known to be a strong pattern, then perhaps, all pairs of the form $\langle X$, `un`$X\rangle$ are good patterns to check.

- As with other approaches to pattern discovery, there are ample opportunities for programmer assistant tools. For example, if a developer types `blockSignal` in a Java code editor, then a call to `unblockSignal` can be suggested or automatically inserted by the editor.

10 Conclusions

In this paper we present DynaMine, a tool for learning common usage patterns from the revision histories of large software systems. Our method can learn both simple and complicated patterns, scales to millions of lines of code, and has been used to find more than 250 pattern violations. Our mining approach is effective at finding coding patterns: two thirds of all dynamically encountered patterns turned out to be likely patterns.

DynaMine is the first tool that combines revision history information with dynamic analysis for the purpose of finding software errors. Our tool largely automates the mining and dynamic execution steps and makes the results of both steps more accessible by presenting the discovered patterns as well as the results of dynamic checking to the user in custom Eclipse views.

Optimization and filtering strategies that we developed allowed us to reduce the mining time by orders of magnitude and to find high-quality patterns in millions lines of code in a matter of minutes. Our ranking strategy that favored patterns with previous bug fixes proved to be very effective at finding error patterns. In contrast, classical ranking schemes from data mining could only locate usage patterns. Dynamic analysis proved invaluable in establishing trust in patterns and finding their violations.

11 Acknowledgements

We would like to thank Wes Weimer, Ted Kremenek, Chris Unkel, Christian Lindig, and the anonymous reviewers for providing useful feedback on how to improve this paper. We are especially grateful to Michael Martin for his assistance with dynamic instrumentation and last-minute proofreading. The first author was supported by the National Science Foundation under Grant No. 0326227. The second author was supported in part by the Graduiertenkolleg “Leistungsgarantien für Rechnersysteme” and the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference*, pages 487–499. Morgan Kaufmann, 1994.

[2] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM Sysposium on Principles of Programming Languages*, pages 98–109, Long Beach, California, USA, 2005.

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

[4] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft, 2004.

[5] J. Bevan and J. Whitehead. Identification of software instabilities. In *Proceedings of the Working Conference on Reverse Engineering*, pages 134–143, Victoria, British Columbia, Canada, Nov. 2003.

[6] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 196–207, San Diego, California, USA, June 2003.

[8] G. Brat and A. Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference*, Big Sky, MT, 2005.

[9] B. Burke and A. Brock. Aspect-oriented programming and JBoss. `http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html`, 2003.

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice Experience (SPE)*, 30(7):775–802, 2000.

[11] D. Carlson. *Eclipse Distilled*. Addison-Wesley Professional, 2005.

[12] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005.

[13] B. Dudney, S. Asbury, J. Krozak, and K. Wittkopf. *J2EE AntiPatterns*. Wiley, 2003.

[14] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implentation*, pages 1–16, 2000.

[15] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[18] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the Working Conference on Reverse Engineering*, pages 90–101, Victoria, British Columbia, Canada, Nov. 2003.

[19] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 190–198, Washington D.C., USA, Nov. 1998.

[20] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings International Workshop on Principles of Software Evolution*, pages 13–23, Helsinki, Finland, Sept. 2003.

[21] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, 2002.

[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–138, San Francisco, California, December 1992.

[23] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, San Diego, California, June 2003.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52, New York, NY, USA, May 2004.

[25] S. Kumar and K. Li. Using model checking to debug device firmware. *SIGOPS Operating Systems Review*, 36(SI):61–74, 2002.

[26] P. Lam and M. Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 275–302, Darmstadt, Germany, July 2003.

[27] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *LNCS 3780*, pages 139–160),, Nov. 2005.

[28] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-13)*, pages 296–305, Sept. 2005.

[29] B. Livshits and T. Zimmermann. Locating matching method calls by mining revision history data. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[30] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, Washington D.C., July 1994.

[31] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005.

[32] A. Michail. Data mining library reuse patterns in user-selected applications. In *Proceedings of the 14th International Conference on Automated Software Engineering*, pages 24–33, Cocoa Beach, Florida, USA, Oct. 1999. IEEE.

[33] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the International Conference on Software Engineering*, pages 167–176, Limerick, Ireland, June 2000.

[34] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89:1–23, 2003.

[35] S. Pestov. jEdit user guide. `http://www.jedit.org/`, 2005.

[36] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 90–94, Edinburgh, Scotland, UK, May 2004.

[37] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. SABER: Smart Analysis Based Error Reduction. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 243–251, Boston, MA, July 2004.

[38] F. V. Rysselberghe and S. Demeyer. Mining version control systems for FACs (frequently applied changes). In *Proceedings of the International Workshop on Mining Software Repositories*, pages 48–52, Edinburgh, Scotland, UK, May 2004.

[39] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[40] S. R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math, 2004.

[41] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 Usenix Security Conference*, pages 201–220, Washington, D.C., 2001.

[42] B. Tate, M. Clark, B. Lee, and P. Linskey. *Bitter EJB*. Manning Publications, 2003.

[43] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, San Diego, California, Feb. 2000.

[44] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms For The Construction And Analysis Of Systems*, pages 461–476, Apr. 2005.

[45] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis*, pages 218–228, Rome, Italy, July 2002.

[46] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[47] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, May 2005.

[48] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the International Conference on Software Engineering*, May 2006.

[49] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sept. 2004.

[50] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proceedings International Workshop on Principles of Software Evolution*, pages 73–83, Helsinki, Finland, Sept. 2003.

[51] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 2–6, Edinburgh, Scotland, UK, May 2004.

[52] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Edinburgh, Scotland, May 2004.