Mining System Specific Rules from Change Patterns

Andre Hora, Nicolas Anquetil, Stéphane Ducasse, Marco Tulio Valente

Mining System Specific Rules from Change Patterns

André Hora
RMoD Team
Inria, Lille, France
andre.cavalcante.hora@inria.fr

Nicolas Anquetil
RMoD Team
Univ. of Lille / Inria, Lille, France
nicolas.anquetil@inria.fr

Stéphane Ducasse
RMoD Team
Inria, Lille, France
stephane.ducasse@inria.fr

Marco Tulio Valente
Department of Computer Science
UFMG, Belo Horizonte, Brazil
mtov@dcc.ufmg.br

***Abstract*—A significant percentage of warnings reported by tools to detect coding standard violations are false positives. Thus, there are some works dedicated to provide better rules by mining them from source code history, analyzing bug-fixes or changes between system releases. However, software evolves over time, and during development not only bugs are fixed, but also features are added, and code is refactored. In such cases, changes must be consistently applied in source code to avoid maintenance problems. In this paper, we propose to extract system specific rules by mining systematic changes over source code history, *i.e.,* not just from bug-fixes or system releases, to ensure that changes are consistently applied over source code. We focus on structural changes done to support API modification or evolution with the goal of providing better rules to developers. Also, rules are mined from predefined rule patterns that ensure their quality. In order to assess the precision of such specific rules to detect real violations, we compare them with generic rules provided by tools to detect coding standard violations on four real world systems covering two programming languages. The results show that specific rules are more precise in identifying real violations in source code than generic ones, and thus can complement them.**

I. Introduction

Tools to detect coding standard violations in source code such as PMD [1] and FindBugs [2] are commonly used to ensure source code quality. The rules provided by these tools are usually created for generic purposes, such as checking method complexity (not too many conditional or loop statements), or suggesting the use of best practices. These rules can be targeted towards several goals, such as reliability or maintainability [3], but very few are focused on the system under analysis.

A significant percentage of violations (warnings) reported by these rules are false positives, and as a result, most violations are not fixed by developers over time, remaining in source code [4], [5], [6]. Thus, they pollute the output of the rule checking tools and contribute to discredit them. Another limitation of such rules are the false negatives, *i.e.,* real violations in source code that are not found [7]. On the other hand, some works suggest that rules are not equal in identifying real violations; some rules perform better than others [8], [3], [9], [10], [4], [5]. It means that developers are normally interested in some rules, *i.e.,* there is in fact an effort to fix warnings generated by some rules. One important question arise: how can we provide better rules to the developers?

One solution is to create rules with the help of experts [10], [6]. Such rules focus on important problems of the system, but they must be manually defined. Getting access to experts and capturing their knowledge into rules is a difficult task. Moreover, in legacy systems this solution may not work due to the lack of experts.

Another solution is to extract rules using a mining process given the existence of recurring refactorings in source code history [11], [12], [13]. To counter the fact that generic rules, in general, do not prevent introduction of bugs in software [8], [3], [9], [14], [10], [5], [15], some work is dedicated to mine rules from source code history by learning from bug-fixes [12], [16], [13], [17], [18]. Focusing on bug-fixes allows to reduce the search space to extract rules. Rules can also be extracted from changes between system releases [11], [19], [20], [13], not taking into account the whole source code history.

However, it is difficult to extract meaningful rules from bug-fixes because bugs can be very domain specific, related to the semantics of things that is not captured by the syntax that rules look at. Furthermore, software evolves over time, and during development not only bugs are fixed, but also features are added, and code might be refactored to improve maintenance. In such cases, changes must be consistently applied over the entire code base to avoid maintenance problems [19]. The code history, *i.e.,* not just bug-fixes or system releases, needs therefore to be investigated as relevant source of information to provide more opportunities to extract rules.

In this paper, we propose to extract system specific rules from source code history by monitoring how API is evolving with the goal of providing better rules to developers. We focus on structural changes done to support API modification or evolution. In this process, information is extracted from incremental revisions in source code history to avoid loss of data. Also, rules are mined from predefined rule patterns that filter and ensure their quality. The proposed approach is validated on four open-source systems, covering two programming languages (Java and Smalltalk), from which specific rules are extracted. Such specific rules are compared with generic rules provided by tools to detect coding standard violations. The results show that our specific rules are more precise in identifying real violations in source code than generic rules. It means that the specific rules can be used to complement the set of generic rules. The contributions of this work are:

- A novel approach to mine system specific rules from source code history (Section II).
- A systematic comparison and evaluation using real-world systems between specific and generic rules to verify which one is more likely to point to real violations

(sections III to VI).

The rest of this paper is structured as follows. We propose research questions for such comparison in Section III. We define our experiment setting in Section IV, detail the results of the empirical study in Section V, and discuss the evaluation of the results in Section VI. Section VII discusses related work, and we conclude the paper in Section VIII.

II. Mining Changes from History

When analyzing source code history, information can be extracted from system releases or from revisions (commits) found in source code repositories. Extracting information from releases is normally done by computing the diff at method level and representing changes in graph in order to discover API or object usages, code peers, recurring bug-fixes, among others. In this process relevant information might be lost. Consider, for example, working with a first release $r1$ and its successor release $r2$. Some refactorings happened between releases $r1$ and $r2$ in class C. Class C was then deleted and is absent from release $r2$. Consequently, information relative to refactorings in class C, which could be applied to other classes, is lost. Another problem when adopting system release changes is the large size of the diff between them, making it difficult to extract relevant information [19], [16]. Extracting information at revision level is more reliable as it avoids loss of data by taking into account incremental changes.

The mining at revision level brings another issue: the great amount of information to be analyzed. Some work avoid such issue by only mining changes related to bug-fixes [12], [16], [13], [17], [18]. This is done by mining commit messages in the system history to find bug-fix changes. Two approaches are normally used: searching for keywords related to bug-fixes [21] or searching for references to bug reports [22]. Identifying a bug-fix commit allows one to identify the code changes that fixed the bug, and, thus, where the bug was located in the source code. Lines of code are related to defects if they are modified by a bug-fix change, since to resolve a problem the lines were changed or removed [5], [10]. From such lines, information is extracted to detect similar places in source code where fixes were not applied. In fact, this limits the size of the search space but also ignores ordinary commits (not specific to bug correction). Again, in this situation, relevant information might be lost when not analyzing ordinary commits. As software evolves, naturally not just bugs are fixed, but code is added, removed and refactored. In addition, another issue is related to how to find bug-fix changes. In fact, adopting keywords related to bug-fixes and references to bug reports in commit messages are project specific standards or may not be adopted by developers.

When comparing the differences between changed source code, it is important to define the changes to be analyzed to extract information. For example, one can keep track of syntactical changes (*e.g.,* when adding or removing conditional statements, modifying expressions), or structural changes (*e.g.,* when adding or removing classes, methods, method invocations). While syntactical changes play important role in approaches in the context of bug discovering [13], [12], [17], [18], they have relatively small role in discovering systematic changes, for which structural changes are better suited since it does not take into account low level modifications [11], [16].

In this section, we describe the proposed approach for mining specific rules from source code history. In this process, first, we extract changes from revisions in source code history (Subsection II-A). Next, based on predefined patterns, we generate rules from such extracted changes (Subsections II-B and II-C). Then, we select rules by filtering the ones which are relevant (Subsection II-D).

A. Extracting Changes from Revisions

We investigate changes at revision level. Specifically, we focus on structural changes done to support API modification or evolution. Consider examples shown in Figures 1 and 2 in which method invocation changes occurred in ArgoUML[1] and Apache Ant[2] (on top of the figures, the older version of the code, in the middle of the figures, the newer version of the code). Figure 1 shows the adoption of a new form to retrieve the Facade model, *i.e.,* calls to Facade.getModel(arg) are replaced by calls to Facade.getRoot(arg). Figure 2 shows the adoption of a standard form for closing files, *i.e.,* calls to InputStream.close() are replaced by calls to FileUtils.close(arg). Note that, in such cases, the old replaced API is *not* necessarily deprecated or excluded from the system (which occurences would make the system failing at compiling time). For instance, the older form to close files (*i.e.,*InputStream.close()) can still be used. In fact, both modifications are invocation changes which occurred incrementally in *different* revisions over time. Then, it is important to ensure they are consistently applied over source code to avoid maintenance problems.

Method: NotationUtilityUml.parseModelElement()

Older version (revision 14952)

```
...
Object nspe =
  Model.getModelManagementHelper().getElement(path,
    Model.getFacade().getModel(me));
...
```

Newer version (revision 14960)

```
...
Object nspe =
  Model.getModelManagementHelper().getElement(path,
    Model.getFacade().getRoot(me));
...
```

Predicates

deleted-invoc(1, Facade, getModel(arg))
added-invoc(1, Facade, getRoot(arg))

Fig. 1. Adoption of a new form to retrieve the Facade model in ArgoUML.

We represent the changes (delta) between two revisions with predicates that describe added or deleted method invocations [11]. The predicates are represented as:

deleted-invoc(id, receiver, signature)
added-invoc(id, receiver, signature)

Where, the predicate *deleted-invoc(...)* represents a deleted invocation; the predicate *added-invoc(...)* represents an added

[1]http://argouml.tigris.org
[2]http://ant.apache.org

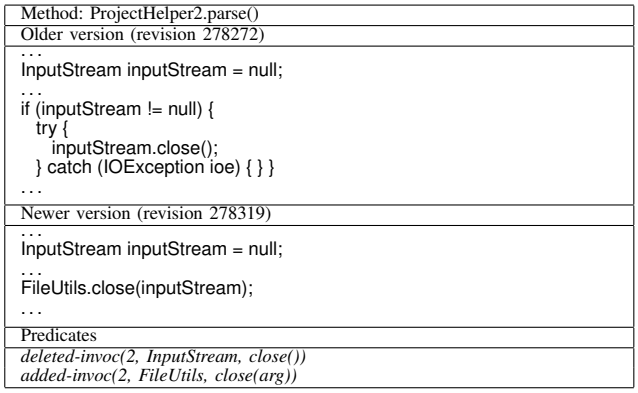

Fig. 2. Adoption of a standard form for closing files in Apache Ant.

invocation; *id* uniquely identifies a change in order to save its context; *receiver* is the type of the receiver; and *signature* is the signature of the invoked method.

Figures 1 and 2 (bottom parts) show the predicates generated by the presented changes. We extract predicates from the changes diff at method level and not from the entire source code of the revisions. Thus, if there are no changes between two method revisions, no predicate is generated. Also, in order to reduce the noise generated by large commits [19], [16], we just consider modifications where one invocation is added and/or deleted. Other differences, such as changes in conditional, are omitted as they have a relatively small role in discovering systematic changes [11]. In fact, refactorings presented in Figures 1 and 2 appear surrounded by different conditionals in other commits.

We use the extracted predicates to mine the rules. These rules are based on predefined invocation change patterns as shown in next subsection.

B. Mining Change Patterns

To improve the quality and relevance of extracted rules, we define change patterns that they must follow. In the case of method invocation, we defined patterns which are likely to be related to API modification or evolution. Table I shows the patterns used in this work. Pattern A finds invocation modification in which receiver and signature changed. Pattern B matches invocation modification in which receivers changed but signatures are the same. Pattern C finds invocation modification in which receivers are the same but signatures changed. Note that there is no overlap between the patterns; each one represents different changes. Additionally, in all the patterns we see that they include a deleted and an added invocation. This is done because we want to assess the violation (deleted invocation) as well as the possible fix (added invocation). This is not an exhaustive list and new patterns can be included.

The extracted predicates representing changes between revisions (Subsection II-A) are used as a database in which we want to find instances of the predefined patterns, *i.e.,* the patterns are queries in such database. When an instance occurs more than a certain frequency it generates a specific rule.

TABLE I
INVOCATION CHANGE PATTERNS.

A	deleted-invoc(id, deletedReceiver, deletedSignature) *and* added-invoc(id, addedReceiver, addedSignature)
B	deleted-invoc(id, deletedReceiver, signature) *and* added-invoc(id, addedReceiver, signature)
C	deleted-invoc(id, receiver, deletedSignature) *and* added-invoc(id, receiver, addedSignature)

Assume, for example, that refactorings shown in Figures 1 and 2 occur frequently over source history. From refactoring shown in Figure 1 we find a rule based on Pattern C where *receiver = Facade*, *deletedSignature = getModel(arg)*, and *addedSignature = getRoot(arg)*:

Rule 1: Facade.getModel(arg) → Facade.getRoot(arg)

From Figure 2 we find a rule based on Pattern A:

Rule 2: InputStream.close() → FileUtils.close(arg)

As a first test, we verified whether these change patterns are frequent enough in practice. We computed them for four large open-source and real-world systems as shown in Table II. We conclude that the defined patterns are in fact recurring over time for the systems under analysis. We provide more information on such frequency in our experiment sections.

TABLE II
NUMBER OF INVOCATION PATTERNS AND OVERVIEW SIZE OF OPEN-SOURCE SYSTEMS. (*) NUMBER OF CLASSES IN LAST REVISION.

System	Classes*	Revisions	Pattern A	Pattern B	Pattern C
Ant	1,203	8,787	598	274	915
Tomcat	1,859	6,248	261	411	684
Lucene	2,888	3,372	1,689	997	2,939
Pharo	3,473	2,972	70	119	126

C. Expanded Rules

We detected that some rules found in source code are very similar. For example, consider the example in Figure 3 showing a change in Apache Ant and the generated predicates. This refactoring is in fact a variation of the refactoring shown in Figure 2 which provides a standard form for closing files. Note that even if they are surrounded by different source code, the invocation changes remain similar. In Figure 2 the deleted receiver is InputStream while in Figure 3 the deleted receiver is BufferedWriter, thus, generating the rule:

Rule 3: BufferedWriter.close() → FileUtils.close(arg)

Based on this observation, we expand a selected rule, when it is possible, to obtain more representative rules. The idea is that for a selected rule, we verify if other rule is its variation. This variation occurs when the selected rule and the other rule *differentiate by one added or deleted receiver or invocation*.

For example, Table III shows the accepted expansions (variations) for the selected Rule 2. Expansion 1 (in bold) matches Rule 3. Thus, we obtain a rule merging two similar refactorings:

Rule 2/3: (InputStream or BufferedWriter).close() → FileUtils.close(arg)

Method: FTP.transferFiles()
Older version (revision 808350)

...
BufferedWriter bw = null;
...
if (bw != null)
  bw.close();
...

Newer version (revision 905214)
... BufferedWriter bw = null; ... FileUtils.close(bw); ...
Predicates
deleted-invoc(3, BufferedWriter, close()) *added-invoc(3, FileUtils, close(arg))*

Fig. 3. Standard form for closing files in Apache Ant (variation of Figure 2).

TABLE III
EXPANSIONS FOR RULE 2. EXPANSION 1 (IN BOLD) MATCHES RULE 3.

Rule 2: InputStream.close() → FileUtils.close(arg)
Expansion 1: *.close() → FileUtils.close(arg)
Expansion 2: InputStream.* → FileUtils.close(arg)
Expansion 3: InputStream.close() → *.close(arg)
Expansion 4: InputStream.close() → FileUtils.*

D. Selecting Relevant Rules

In a real set of source code changes, many rules may be found. We want to limit this amount by creating relevant ones in order to avoid rules producing false positives warnings [6], [4]. Then, we create rules by taking into account their frequency over time and the expanded rules:

Frequency over time. We analyze rules that occur over time, taking into account their occurrence over different revisions. Thus, a rule that occurs in two different revisions is more relevant than another that occurs in just one revision. The idea is that when rules occur in different revisions, they are in fact being incrementally fixed by developers. It also means that such rules are *not* about refactorings which could have been detected by the system failing at compile time, since they are being in fact fixed over time.

Expanded rules. We analyze the proposed approach on expanding a rule (see Section II-C).

III. RESEARCH QUESTIONS

A significant percentage of warnings reported by generic rules are false positives, and as a result, most warnings are not fixed by developers over time, remaining in source code [4], [5], [6]. However, some works suggest that rules are not equal in identifying real violations [8], [3], [9], [10], [4], [5]. This means that developers are normally interested in some rules. Based on that, how can we provide better rules to the developers? The proposed research questions are intended to compare the specific rules generated by our approach with generic ones. We evaluate if our approach can be used to improve the set of rules provided by tools to detect coding standard violations, and then provide better rules to developers.

Therefore, firstly, we assess if warnings produced by specific rules are more likely to point to real violations than warnings produced by generic rules.

RQ1 *Are specific warnings more likely to point to real violations than generic warnings?*

Research question RQ1 does not assess rules individually, just two groups exist: specific and generic, which are then compared. However, as aforementioned, rules are not equal in identifying violations; some rules perform better than others. Then, it is reasonable to compare rules individually:

RQ2 *Are specific rules more likely to point to real violations than generic rules?*

Generic rules are somehow easier to define than specific ones, because one can create them once for all; every new rule is a "definitive" contribution. On the other hand, specific rules must be extracted for each new system. Thus, more generic warnings is expected and, consequently, also lower precision (each warning will have a lower probability of indicating a real violation). Thus, we could expect that generic rules will fare lower than specific ones, simply because they are more flexible (*i.e.,* independent of system) and will produce more warnings. To have a fairer comparison, we perform the same experiments as the previous two research questions but comparing just the best rules, *i.e.,* rules which produced at least one true positive warning. Comparing just the best rules was also previously used to reduce the amount of rules/warnings [10].

RQ3 *Are best specific warnings more likely to point to real violations than best generic warnings?*

RQ4 *Are best specific rules more likely to point to real violations than best generic rules?*

IV. EXPERIMENT SETTING

In this section, we plan our experiment to answer the research questions as suggested in [23]. We have two different experiments to setup, firstly, to answer research questions RQ1 and RQ3, and secondly, to answer RQ2 and RQ4.

A. Context

The *context* of the experiment is real systems for which source code history and generic tools to detect coding standard violations are available. We need real systems to ensure that our experiment is meaningful. We need source code history to extract specific rules and we need generic rules to compare with the generated specific rules.

We selected three Java systems (Ant, Tomcat[3], Lucene[4]) and one Smalltalk system (Pharo[5]) to perform our empirical studies. They are open-source, real-world, and non-trivial systems, with a consolidated number of developers and users. Also, they have relevant source code history and different missions, covering different domains: Ant is a tool for automating software build processes, Tomcat is a web server and servlet container, Lucene is an information retrieval software library, and Pharo is a Smalltalk-inspired language and environment. Table II details the size of the systems under analysis.

With respect to the generic tool to detect coding standard violations in Java, we selected PMD [1], one of the most adopted Java code analysis tool, and also previously used by

[3]http://tomcat.apache.org
[4]http://lucene.apache.org
[5]http://pharo-project.org

related studies [4], [12]. We used PMD because it only requires Java source code as its input. Other code analysis tools, such as FindBugs, depend on Java class files as their input, which would require to deal with legacy libraries, and source code compilation for every revision which is computationally expensive. For Smalltalk, we selected SmallLint [24], the most adopted Smalltalk code analysis tool, also previously used by related studies [10], [6].

B. Experiment for RQ1 and RQ3

1) Hypotheses Formulation:

$H_0^{1,3}$ *Specific and generic warnings are equally precise in identifying violations.*

$H_a^{1,3}$ *Specific and generic warnings are not equally precise in identifying violations.*

2) Subject Selection and Variable: The *subjects* for this experiment are the *warnings* generated by specific rules on lines of code. We measure the number of correctly fixed specific warnings during the experiment timeframe. We also measure the number of fixed generic warnings.

The *dependent variable* is the warnings generated by specific rules. It is categorical and takes two values: true warning and false warning. A true warning occurs when it points to a real violation, *i.e.,* a warning that has been fixed at some moment. A false warning is the one that remains in the source code and has never been fixed. In this experiment we compare the number of (true and false) specific warnings with the number of (true and false) generic warnings, *i.e.,* the expected values.

3) Experiment Design: To test the hypotheses $H^{1,3}$ we use the Chi-square goodness-of-fit test which is used when there is one categorical variable, with two or more possible values. This test allows us to test whether the observed proportions for a categorical variable (true and false specific warnings) differ from expected proportions (true and false generic warnings). The null hypothesis is that the number of observed proportions is equal to the expected proportions. If we cannot reject the null hypothesis, we conclude that observed and expected proportions of true and false specific and generic warnings are equals, *i.e.,* there is no statistically significant difference between true and false specific and generic warnings. When we can reject the null hypothesis, we conclude that observed and expected proportions of true and false specific and generic warnings are not equals. As is customary, the tests will be performed at the 5% significance level.

Note that concluding that the proportions are not equals does not answer RQ1 and RQ3 which state that specific rules performs *better* (they could be different but worse). Therefore, when rejecting the null hypothesis, we also need to check the *residuals*: the difference between the observed and expected warnings. When the absolute value of the residual is greater than 2.00, it is considered that it is major contributor to the rejection of the null hypotheses.

We also report the *effect size* which measures the distance between the null hypothesis and alternative hypothesis, and is independent of sample size. Effect size value 0.1, 0.3 and 0.5 are considered small, medium and large effects, respectively.

C. Experiment for RQ2 and RQ4

1) Hypotheses Formulation:

$H_0^{2,4}$ *Specific and generic rules are equally precise in identifying violations.*

$H_a^{2,4}$ *Specific rules are more precise in identifying violations than generic rules.*

Note that we make a directional (one-tailed) hypothesis. This should be made when there is evidence to support such a direction. This evidence will stem from the results of the first experiment.

2) Subject Selection and Variable: The *subjects* for this experiment are the generic and specific *rules*. We measure the precision of each generic and specific rule, *i.e.,* the precision of the warnings generated by each rule individually. If the rule does not produce any warning, we cannot compute its precision, thus it cannot be evaluated. Therefore, two samples are generated, one sample with precision of generic rules and other sample with precision of specific rules for the experiment timeframe. Note that the *subjects* for this experiment are the *rules* while the subjects for the previous experiment are the *warnings*.

The *independent variable* is the rule group. It is categorical and takes two values: specific or generic. The *dependent variable* is the rule precision.

3) Experiment Design: For this experiment, we use an unpaired setting, which means the rules composing one sample (specific rules) are not the same than those composing the other sample (generic rules). To test the hypotheses $H^{2,4}$ we use Mann-Whitney test which is used for assessing whether one of two samples of independent observations tends to have larger values than the other. It can be used when the distribution of the data is not normal and there are different participants (not matched) in each sample. The null hypothesis is that median rule precision is the same for both samples. If we cannot reject the null hypothesis, we conclude that there are not statistically significant differences between precisions in both samples. Again the tests will be performed at the 5% significance level. We also report the *effect size*.

D. Instrumentation

In this section we describe how generic and specific rules are obtained. For each case, we also detail how the precisions are calculated.

1) Obtaining Generic Rules: We use PMD and SmallLint as the tools that provided the generic rules. They are static code analysis tools, which help developers on detecting coding standard violations. Such tools provides rules that are adopted as common coding standards. For PMD, we used the rule groups: Basic, Clone Implementation, Controversial, Design, Empty Code, Finalizer, Import Statements, Optimization, Strict Exceptions, String and StringBuffer, Security Code Guidelines, Type Resolution, Unnecessary and Unused Code, which result in 180 rules. More details on each group can be found in PMD website[6]. For SmallLint, we used rules related to optimization, design flaws, coding idiom violation, bug and

[6]http://pmd.sourceforge.net/pmd-5.0.2/rules

potential detection, which result in 85 rules. For both tools, we did not use rules related to code size, naming convention and comment/style as they can be specific for each project.

For the generic rules, a true positive (TP) warning is the one which has been removed at some point during the experiment timeframe, and a false positive (FP) warning is the one which remains in source code and has never been removed [4], [5]. Thus, we measure the precision of rule (or a group of rules) from the portion of warnings removed over all warnings generated by such rule, *i.e., precision = TP/(TP + FP)*. Also, we do not take into account removed warnings due to class deletion, which are not necessarily the result of a fix [5].

2) Obtaining Specific Rules: Differently from the generic rules, we do not have the specific rules beforehand. To learn specific rules from project history we make use of the approach developed by Kim *et al.* [12] on navigating through revisions to extract information. It suits well in our approach since it works by learning from incremental changes in revisions. The idea is that we walk through the revision history of a project *learning* rules and *evaluating* at each revision how well our approach works when using only the information available up to that revision [12]. We learn a rule when it occurs f times in different revisions. We evaluate at revision n the rules learned from revisions 1 to $n-1$. If the change in revision n matches the rule, *i.e.,* the modification to obtain revision n matches the deleted and added invocation in the rule, we have a true positive (TP) warning. If the fix in revision n matches the deleted invocation in the rule, but not the added invocation, we have a false positive (FP) warning. Thus, we measure the precision of a rule (or a group of rules) from the portion of warnings correctly predicted over all warnings generated by such rule, *i.e., precision = TP/(TP + FP)*. In this process, revision history is represented as predicates described in Section II-A and the rules follow one of the invocation change patterns described in Section II-B.

In the learning process we assess (see Subsection II-D): frequency over time, and expanded rules. Below we describe how they are used to support discovering relevant rules:

Frequency over time. A rule is frequent when it occurs f times in different revisions with $f \geq 2$.

Expanded rules. When a rule is detected to be frequent in revision n, it can be expanded to be grouped with similar rules (see Section II-C). This process can be done in two ways, by merging the new rule with *past* or *future* changes. Past changes (between revisions 1 and $n-1$) mean that we merge the new rule with other similar ones that we were considering as candidates. Future changes mean that, in subsequent revisions, changes similar to the new rule will be merged with it. In this last case, it means that the rule will keep evolving over time as similar changes will be found. Note that expanded rules may also not be used (*none*).

Table IV shows the results of the learning process as measured by the number of produced rules and their precision. For each system, we select the specific rules obtained by the learning process that produced the best precision and produced at least 20 rules. Thus, the selected training set is when adopting frequency $f = 2$ and expanding rules with *future* / *none*. It means that a modification following Patterns A, B, or C is a specific rule when it occurs two times in different revisions. It also means that such rules are not about changes which could have been detected by compiling time fails, since they are being in fact fixed over the revisions. Also, using the *future* training improved the precision of the rules for three systems. This shows that expanded rules can in fact be useful to have better rules. Note, however, that the precision in this case is for the group of specific rules, not for rules individually.

TABLE IV
OBTAINING SPECIFIC RULES.

System	Frequency	Expanding rules	Rules	Precision
Ant	2	future	31	0.12
Tomcat	2	future	20	0.35
Lucene	2	future	37	0.18
Pharo	2	none	37	0.49

3) Obtaining Best Rules: We also need to assess best generic and specific rules as described in Section III (see RQ3 and RQ4). The best rules are the ones that generate at least one true positive warning.

V. EXPERIMENT RESULTS

In this section, we present the results of our empirical study. We first present the results to answer RQ1 and RQ2 when all rules are compared. Then, we present the results to answer RQ3 and RQ4 when just the best are compared.

A. Evaluating All Rules: RQ1 and RQ2

RQ1 *Are specific warnings more likely to point to real violations than generic warnings?*

H_0^1 *Specific and generic warnings are equally precise in identifying violations.*

H_a^1 *Specific and generic warnings are not equally precise in identifying violations.*

Table V shows the number of generic (expected values) and specific (observed values) warnings for the systems under analysis. When applying the Chi-square goodness-of-fit test the *p-value* < 0.001 for the four systems. We can reject the null hypothesis for such systems with a very small probability of error, and conclude that specific and generic warnings are not equally precise in identifying violations. Then, as the null hypothesis is rejected, we check the residuals: the difference between the observed and expected warnings. Table V shows the expected number of specific warnings (based on provided generic warnings) and residuals for the specific warnings. We see that residual for true positives (TPs) are positively over-represented (> 2) so they are the major contributor to the rejection of the null hypothesis. Finally, the *effect size* are all > 0.5 (large effect).

Next, we evaluate generic and specific rules that produced at least one warning.

RQ2 *Are specific rules more likely to point to real violations than generic rules?*

H_0^2 *Specific and generic rules are equally precise in identifying violations.*

H_a^2 *Specific rules are more precise in identifying violations than generic rules.*

TABLE V
GENERIC AND SPECIFIC WARNINGS (RQ1).

System	Analysis	TPs	FPs	Warnings	Prec.
Ant	Generic	1,301	37,870	39,171	0.03
	Specific	175	1,285	1,460	0.12
	Expected	44	1,416		
	Residual	+19.2	-3.5		
Tomcat	Generic	5,071	77,123	82,194	0.06
	Specific	205	372	577	0.35
	Expected	35	542		
	Residual	+30	-7.3		
Lucene	Generic	9,025	126,172	135,197	0.07
	Specific	334	1,493	1,827	0.18
	Expected	128	1,699		
	Residual	+18.2	-5		
Pharo	Generic	202	13,315	13,517	0.015
	Specific	136	137	273	0.49
	Expected	4.1	268.9		
	Residual	+65.2	-8		

We set two samples of precision for each system, the first sample with generic rule precisions, and the second sample with specific rule precisions. Table VI shows the number of rules (which produced at least one warning) and the average precision in each sample. When applying the Mann-Whitney test in such samples we have *p-value* < 0.01 for Tomcat and Pharo, and *p-value* > 0.05 for Ant and Lucene. We can reject the null hypothesis and conclude that specific rules are more precise in identifying violations than generic rules for Tomcat and Pharo, but we cannot reject the null hypothesis for Ant and Lucene. Also, the *effect size* for Tomcat is $= 0.19$ (small effect) and for Pharo is 0.48 (medium effect).

TABLE VI
GENERIC AND SPECIFIC RULES (RQ2).

	All Generic Rules		All Specific Rules	
System	Rules	Avg. precision	Rules	Avg. precision
Ant	143	0.16	25	0.11
Tomcat	143	0.14	14	0.51
Lucene	142	0.08	32	0.12
Pharo	67	0.03	16	0.52

For the next experiment, we removed Ant because its RQ2 was rejected and its average precision for specific rules is lower than the average precision for generic rules (see Table VI).

B. Evaluating Best Rules: RQ3 and RQ4

In this experiment we evaluate the best rules, *i.e.,* rules that produced at least one true positive warning.

RQ3 *Are best specific warnings more likely to point to real violations than best generic warnings?*

H_0^3 *Best specific and generic warnings are equally precise in identifying violations.*

H_a^3 *Best specific and generic warnings are not equally precise in identifying violations.*

Table VII shows the number of generic (expected values) and specific (observed values) warnings for the best rules. Applying the Chi-square goodness-of-fit test gives a *p-value* < 0.001 for Tomcat, Lucene and Pharo. We can reject the null hypothesis for such systems. Then, as the null hypothesis is rejected, we check the residuals. Table V also shows expected number of specific warnings and residuals for the specific warnings. We see that residual for true positives (TPs) are positively over-represented (> 2) and they are the major contributor to the rejection of the null hypothesis. Finally, the *effect size* for Tomcat and Pharo is > 0.5 (large effect), and for Lucene $= 0.37$ (medium effect).

TABLE VII
BEST GENERIC AND SPECIFIC WARNINGS (RQ3).

System	Analysis	TPs	FPs	Warnings	Prec.
Tomcat	Generic	5,071	53,334	58,405	0.09
	Specific	205	151	356	0.58
	Expected	32	324		
	Residual	+30.5	-9.6		
Lucene	Generic	9,025	45,559	54,584	0.16
	Specific	334	794	1,128	0.30
	Expected	180	948		
	Residual	+11.4	-4		
Pharo	Generic	202	3,077	3,279	0.06
	Specific	136	104	240	0.56
	Expected	14.4	225.6		
	Residual	+32	-8.1		

Finally, we test if best specific rules are better than best generic rules when they are compared individually.

RQ4 *Are best specific rules more likely to point to real violations than best generic rules?*

H_0^4 *Best specific and generic rules are equally precise in identifying violations.*

H_a^4 *Best specific rules are more precise in identifying violations than best generic rules.*

We set two samples of precision for each system, the first sample with best generic rule precisions, and the second sample with best specific rule precisions. Table VIII shows the number of rules (which produced at least one true positive warning) and the average precision in each sample. Applying the Mann-Whitney test in such samples gives a *p-value* < 0.01 for Tomcat and Pharo and $= 0.10$ for Lucene. We can reject the null hypothesis for Tomcat and Pharo, but we cannot reject the null hypothesis for Lucene. Also, the *effect size* for Tomcat is $= 0.28$, for Lucene is $= 0.14$ (small effects), and for Pharo is $= 0.15$ (large effect). Even if we cannot reject the null hypothesis for Lucene, we still see a small effect.

TABLE VIII
BEST GENERIC AND SPECIFIC RULES (RQ4).

	Best Generic Rules		Best Specific Rules	
System	Rules	Avg. precision	Rules	Avg. precision
Tomcat	78	0.26	10	0.71
Lucene	61	0.18	11	0.36
Pharo	22	0.10	12	0.69

VI. DISCUSSION

In this section, we discuss the results of our experiments. We also present the threats to the validity of these experiments.

A. Evaluating Rules

1) All Rules: We studied whether specific warnings are more likely to point to real violations than generic ones. The outcome of this experiment is that specific warnings are in fact more likely to point to real violations (RQ1). This was true for all the case studies. As expected, in general, tools to detect coding standard violations produce too many false positives [6], [4], [5]. In this case, precision of generic warnings remained between 0.015 and 0.07 (which is coherent with previously published results [5], [10], [12]), while precision of specific warnings remained between 0.12 and 0.49.

However, rules are not equal in identifying real violations, *i.e.,* some rules perform better than others. Thus, we also studied whether specific rules are more likely to point to real violations than generic ones. This was true for Tomcat and Pharo (RQ2). We could not show that the specific rules performed better than the generic ones for Ant and Lucene. This was mostly because warnings generated by some generic rules in these systems were almost all fixed during the experiment timeframe. For instance, in Ant, warnings generated by six rules[7] were all fixed.

We conclude that, for the case studies under analysis, specific warnings are more effective to point to real violation in source code than generic ones. When comparing rules individually it depends of the case study.

2) Best Rules: In this experiment we are fairer to both goups, since we compare just rules which produced at least one true positive warning, thus excluding "bad" rules. We studied whether best specific warnings are more likely to point to real violations than best generic ones. The outcome of this experiment is that best specific warnings can be in fact more likely to point to real violations (RQ3). This was true for all the systems under analysis. In this case, precision of generic warnings remained between 0.06 and 0.16, while specific rule precisions remained between 0.30 and 0.58. Finally, we studied whether specific best rules are more likely to point to real violations than best generic ones. It was true for Tomcat and Pharo, but we could not show that for Lucene (RQ4).

It is important to note that in this experiment we are comparing the just discovered specific rules with generic rules that are known to be "good" rules. This means that in fact there was an effort to fix warnings generated by such generic rules, so they are important for developers and in software maintenance. Thus, with the results showed in this experiment, we are able to find specific rules as good as (in Lucene) or even better (in Tomcat and Pharo) than such best generic rules.

We conclude that, for the case studies, best specific warnings are the more effective to point to real violation than best generic ones. When comparing rules individually, we are able to detect specific rules as good as or even better than the best generic rules, and thus complement them.

B. Concrete Cases

An important outcome of our experiments is that rules can be in fact extracted from code history. As shown in Table IV a total of 125 specific rules following Patterns A, B and C were extracted from the four case studies. In this subsection we discuss concrete cases we found during the experiments.

In Ant, we detected the rules described in Figures 2 and 3 with respect to the adoption of a standard form for closing files. This feature was introduced in Ant in class FileUtils in revision 276747 (Aug 2004). The idea was to adopt this refactoring in source code to avoid code duplication as pointed by the committer. We firstly detected this refactoring in class Echo in revision 276748 (Aug 2004), just after the feature was indroduced. We keep seeing such modification in, for example, class AntClassLoader in revision 533214 (Apr 2007) and in class KeySubst in revision 905214 (Feb 2010). In addition, such modification came not just from source code originally created *before* the feature addition, but also from source code created *after* the feature addition. This means that source code was inserted wrongly even when it was possible to be correctly inserted since the feature could have been used. In practice this refactoring has never been fully adopted as even almost six years later, in revision 905214 (Feb 2010), the refactoring was still being applied. This rule produced 100 warnings from which just 37 were fixed by developers over time. Our approach caught it and thus can be used to avoid similar maintenance problems.

In Tomcat, our approach detected some rules defined by FindBugs such as "DM_NUMBER_CTOR: Method invokes inefficient Number constructor; use static valueOf instead". It is applied to constructors such as Long, Integer, and Character. This rule is intended to solve performance issues and it states that using valueOf is approximately 3.5 times faster than using constructor. In fact, our approach detected such rules because Tomcat developers have been using FindBugs over time. Even if there was an effort to fix such warnings, they were not completely removed. For instance, class GenericNamingResourcesFactory in its last revision 1402014 (Oct 2012) still contains such warnings. Moreover, in the same class, some revisions before (in 1187826), a similar FindBugs warning (also detected by our approach) was fixed, but the one cited (DM_NUMBER_CTOR) remained in source code. This means that some developers may not be aware of common refactorings even when tools to detect coding standard violations are adopted. Also, the great amount of warnings generated by such tools in real-world systems is not easy to manage [6], [4], [5]. This rule produced 59 warnings from which 35 were fixed by developers. Our approach caught this refactoring and again it can be used to avoid similar problems where changes are not consistently applied.

In Lucene, rules are related, for example, to structural changes (*e.g.,* replace Document.get() by StoredDocument.get()) and to internal guidance to have better performance (*e.g.,* replace Analyzer.tokenStream() by Analyzer.reusableTokenStream(), replace Random.nextInt() by SmartRandom.nextInt()). The Java systems also produced rules related to Java API migrations such as the replacement of calls from the classes Vector to ArrayList, Hashtable to Map, and StringBuffer to StringBuilder, which were incrementally fixed by developers, and thus also detected by our approach.

In Pharo, some rules are also related to structural changes (*e.g.,* replace FileDirectory.default() by FileSystem.workingDirectory()). None of the parts that should be replaced of the rules involved explicitly deprecated methods, *i.e.,* there is no overlap between discovered rules and deprecated methods. In some cases, the methods involved with the parts that should be replaced are no more presented in the last release. This means that warnings generated by such rules are in fact real bugs[8]. Also, some rules point to methods which are strong candidates to be removed or deprecated. For example, in the rule stating to replace OSPlatform.osVersion() by OSPlatform.version(), the the part that should be replaced is kept in the system for compatibility with the previous release, but it should be soon deprecated as stated in its comment; in the rule stating to replace RPackageOrganizer.default()

[7]PMD id for the rules: BrokenNullCheck, CloseResource, FinalizeShouldBeProtected, IdempotentOperations, MisplacedNullCheck, UnnecessaryConversionTemporary

[8]Notice that invocations to methods not presented in the system do not fail at compiling time because Smalltalk is dynamic typed.

by RPackageOrganizer.organizer(), the part that should be replaced is kept for special reasons, and it must be invoked only in strict cases as stated in its comment. By analysing incremental changes we were able to discover important rules, and then provide more focused rules to developers

C. Threats to Validity

1) Internal Validity: The specific rules extracted from source code history may become "obsolete" over time. This problem is also common in other studies related to mining information from code history. We try to minimize this problem by expanding rules, *i.e.,* merging small variations of the same rule such that the rule keeps evolving. Most importantly, we minimize it in our experiments because we validate the rules incrementally over the revisions. It means that when a rule is valid it will be validated accordingly, and if the same rule becomes obsolete it will not produce warnings. By adopting this approach, we are able to validate all the rules that happened over the history, so we are fairer to them.

The patterns used to extract rules from source code history may be an underestimation of the real changes occurring in commits: some changes are more complex, only introduce new code, or only remove old code. We do not extract rules from such cases, and they might represent relevant sources of information. Dealing with larger changes remains future work.

2) External Validity: Ant, Tomcat, Lucene, and Pharo are credible case studies as they are open-source, real-world and non-trivial systems with a consolidated number of developers and users. They also come from different domains and include a large number of revisions. Despite this observation, our findings – as usual in empirical software engineering – cannot be directly generalized to other systems, specifically to systems implemented in other languages or to systems from different domains. Closed-source systems, due to differences in the internal processes, might have different properties in their commits. Also, small systems or systems in initial stage may not produce information sufficient to generate specific rules.

With respect to the generic rules, PMD and Smalllint are related to common coding standards. However, we did not use rules related to code size, naming convention and comment/style as they can be specific for each project.

VII. Related Work

Williams and Hollingsworth [18] focus on discovering warnings which are likely real bugs by mining code history. They investigate a specific type of warning (checking if return value is tested before being used), which is more likely to happen in C programs. They improve bug-finding techniques by ranking warnings based on historical information. While the authors investigate a single known type of warning, we are intended to discover new system specific types of warnings. Another similar research in the sense that authors use historical information to improve ranking mechanism is proposed by Kim and Ernst [5]. They propose to rank warnings reported by static analysis tools based on the number of times such warnings were fixed over the history. Again, they focus on defined rules while we focus on new system specific rules that are not included in static analysis tools.

Kim *et al.* [12] aims to discover system specific bugs based on source code history. They analyze bug-fixes over history extracting information from bug-fix changes. Information is extracted from changes such as numeric and string literals, variables and method calls, and stored in a database. The evaluation of their approach, like our study, is done by navigating through revisions and evaluating in subsequent revisions what was previously learned. This is done to simulate the use of their approach in practice where the developer receiver feedback based on previous changes. Our approach and the related work are intended to find system specific warnings. However, there are important differences between the two approaches in the way this is performed and provided to developers. While the authors check revision changes related bug-fixes in the learning process, we check all the revisions. Also, when extracting calls from changes, the type of the receiver is not considered, while this is done by our approach. Most importantly, our approach is based on predefined change patterns and expansions to discover rules. It means that we materialize warnings in rules, while the related work stores warnings in a database, which is can be then accessed. Comparing our approach with theirs is not suited because they are not intended to produce rules.

Livshits and Zimmermann [16] propose to discover system specific usage patterns over code history. These patterns are then dynamically tested. In order to support discovering such pattern, they use the data mining technique Apriori. Results show that usage patterns, such as method pairs, can be found from history. Our approach and the related work are intended to find system specific patterns. While the related work extract usage patterns to understand how methods should be invoked, we extract invocation changes patterns to understand how invocations should be updated due API modification or evolution. Also, the related work does not establish a fixed pattern to be found. As they use the data mining technique Apriori, they have more pattern flexibility. On the other hand, with our predefined patterns, we have less flexibility (because they are fixed) but we have more expressivity (because we can vary receiver and signature, and also expand rules).

Other studies focus on extracting information by analyzing the differences between two releases/versions of a system. Nguyen *et al.* [13] aim to discover recurring bug-fixes by analyzing two versions of a system. Then, they propose to recommend the fix according to learned bug-fixes. Our work is not restricted to bug-fix analysis and we also extract rules from the system history. Mileva *et al.* [19] focus on discovering changes which must be consistently applied in source code. These changes are obtained by comparing two versions of the same project, determining object usage, and deriving patterns. In this process they compare object usage models and temporal properties from each version. By learning systematic changes, they are able to find places in source code where changes were not correctly applied. Although the idea of our research is similar to theirs, our focus is different. Both approaches are intended to ensure changes to be consistently applied in source code. However, we focus on invocation changes (in order to produce rules) while the related work focus on object usage. Also, while they extract changes between two versions, we extract changes from incremental versions.

Kim and Notkin [11] propose a tool to support making differencing between two system versions. Each version is represented with predicates that capture structural differences. Based on the predicates, the tool infers systematic structural differences. While they use predicates to infer logic rules,

we also adopt predicates to support the definition of change patterns. Sun *et al.* [17] propose to extend static analysis by discovering specific rules. They focus on mining a graph, with data or control dependences, to discover specific bugs. We focus on mining changes to deal with API evolution. While they extract data from a single version, we extract from code history. Also, we are not restricted to bug-fix analysis.

Other studies focus on recovering rules from execution traces [25], [26]. While such studies extract rules via dynamic analysis of a single system version to produce temporal rules (*e.g.,* every call to *m1()* must be preceded by a call to *m2()*), we extract rules via static analysis of changes from incremental versions to produce evolution rules (*e.g.,* every call to *m1()* must be replaced by a call to *m2()*).

Finally, one can compare our approach with auto-refactoring functions of existing IDEs. Our approach is intended to discover new rules while auto-refactoring functions are intended to apply "rules". In addition, suppose that at some moment a change should have been done by the developer. Then, if the developer used the auto-refactoring to perform the change, it is important to ensure such change to be also followed by subsequent versions or by client systems. If the developer partly used the auto-refactoring to perform the change, it is also important to ensure changes are consistently applied (*e.g.,* the Ant and Tomcat cases shown in Section VI-B). Our approach is also suited to support in such two cases.

VIII. Conclusion

In this paper, we proposed to extract system specific rules from source code history by monitoring how API is evolving with the goal of providing better rules to developers. The rules are extracted from incremental revisions in source code history (not just from bug-fixes or system releases). Also, they are mined from predefined rule patterns that ensure their quality. The extracted rules are related to API modification or evolution over time and are used to ensure that changes are consistently applied in source code to avoid maintenance problems.

We compared specific rules, extracted from four systems covering two programming languages (Java and Smalltalk), with generic rules provided by two static analysis tools (PMD and SmallLint). We showed that the analysis of source code history can produce rules as good as or better than the overall generic rules and the best generic rules, *i.e.,* rules which are known to be relevant to developers. Thus, specific rules can be used to complement the generic ones. All the results reported in this comparison were statistically tested, and they are not due to chance. With the reported results, we expect more attention to be given to specific rules extracted from source code history in complement to generic ones.

As future work, we plan to extend this research by introducing new patterns. Also, we plan to extract rules from other structural changes such as class access and inheritance.

Acknowledgments: This research is supported by Agence Nationale de la Recherche (ANR-2010-BLAN-0219-01).

References

[1] T. Copeland, *PMD Applied*. Centennial Books, 2005.

[2] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *Object Oriented Programming Systems Languages and Applications*, 2004, pp. 132–136.

[3] C. Boogerd and L. Moonen, "Assessing the Value of Coding Standards: An Empirical Study," in *International Conference on Software Maintenance*, 2008, pp. 277 –286.

[4] S. S. Joao Araujo Filho and M. T. Valente, "Study on the Relevance of the Warnings Reported by Java Bug-Finding Tools," *Software, IET*, vol. 5, no. 4, pp. 366–374, 2011.

[5] S. Kim and M. D. Ernst, "Which Warnings Should I Fix First?" in *European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, 2007, pp. 45–54.

[6] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz, "Domain-Specific Program Checking," in *Objects, Models, Components, Patterns*, 2010, pp. 213–232.

[7] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *International Conference on Automated Software Engineering*, 2012, pp. 50–59.

[8] W. Basalaj and F. van den Beuken, "Correlation Between Coding Standards Compliance and Software Quality," Programming Research, Tech. Rep., 2006.

[9] C. Boogerd and L. Moonen, "Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions," in *Working Conference on Mining Software Repositories*, 2009, pp. 41–50.

[10] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain Specific Warnings: Are They Any Better?" in *International Conference on Software Maintenance*, 2012.

[11] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *International Conference on Software Engineering*, 2009, pp. 309–319.

[12] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of Bug Fixes," in *International Symposium on Foundations of Software Engineering*, 2006, pp. 35–45.

[13] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring Bug Fixes in Object-Oriented Programs," in *International Conference on Software Engineering*, 2010, pp. 315–324.

[14] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static Correspondence and Correlation Between Field Defects and Warnings Reported by a Bug Finding Tool," *Software Quality Journal*, pp. 1–17, 2012.

[15] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," in *Symposium on Foundations of Software Engineering*, 2004, pp. 83–93.

[16] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," in *European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, 2005, pp. 296–305.

[17] B. Sun, G. Shu, A. Podgurski, and B. Robinson, "Extending static analysis by mining project-specific rules," in *International Conference on Software Engineering*, 2012, pp. 1054–1063.

[18] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Transactions on Software Engineering*, vol. 31, pp. 466–480, 2005.

[19] Y. M. Mileva, A. Wasylkowski, and A. Zeller, "Mining Evolution of Object Usage," in *European Conference on Object-Oriented Programming*, 2011, pp. 105–129.

[20] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010, pp. 302–321.

[21] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases," in *International Conference on Software Maintenance*, 2000, pp. 120–130.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do Changes Induce Fixes?" in *International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[24] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, pp. 253–263, 1997.

[25] D. Lo, S.-C. Khoo, and C. Liu, "Mining Temporal Rules for Software Maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 227–247, 2008.

[26] D. Lo, G. Ramalingam, V.-P. Ranganath, and K. Vaswani, "Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation," *Science of Computer Programming*, vol. 77, no. 6, pp. 743–759, 2012.