Identifying the exact fixing actions of static rule violation

Hayatou Oumarou, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Dina Kolyang

Identifying the Exact Fixing Actions of Static Rule Violation

Hayatou Oumarou*, Nicolas Anquetil†, Anne Etien†, Stéphane Ducasse†, Kolyang Dina Taiwe*
*University of Maroua, Maroua, Cameroun
hayaty55@yahoo.fr, dtaiwe@yahoo.fr
†RMoD Team, Inria, Lille, France
{firstName.lastName}@inria.fr

***Abstract*—We study good programming practices expressed in rules and detected by static analysis checkers such as PMD or FindBugs. To understand how violations to these rules are corrected and whether this can be automated, we need to identify in the source code where they appear and how they were fixed. This presents some similarities with research on understanding software bugs, their causes, their fixes, and how they could be avoided. The traditional method to identify how a bug or a rule violation were fixed consists in finding the commit that contains this fix and identifying what was changed in this commit. If the commit is small, all the lines changed are ascribed to the fixing of the rule violation or the bug. However, commits are not always atomic, and several fixes and even enhancements can be mixed in a single one (a large commit). In this case, it is impossible to detect which modifications contribute to which fix. In this paper, we are proposing a method that identifies precisely the modifications that are related to the correction of a rule violation. The same method could be applied to bug fixes, providing there is a test illustrating this bug. We validate our solution on a real world system and actual rules.**

I. Introduction

Research has been devoted to the study of violations of good programming practices detected by automatic rule checkers (*e.g.,* FindBugs[1] or PMD[2]). It often concentrates on predicting if a violation is a "real" one that should be corrected, or a false positive (*e.g.,* [1]). We are more interested in studying whether we could help correcting these violations.

In a related domain, more research focused on understanding how bugs are solved, whether it is possible to provide foretellers of bugs, or how to automatically fix certain types of bugs (*e.g.,* [2], [3], [4]). This involves identifying where in the source code the bugs are located which in turn is typically done by looking at commits known to fix bugs and consider that the modified lines of code in one commit were the ones responsible for the bug it fixes.

If finding the location of a rule violation is trivial because it is an integrant part of the report provided by the rule checker tools, finding out how a rule violation was fixed is a topic rarely studied. Similarly to bug fixes, it could be done by looking at past violations of the rule that were already fixed, and finding out in the source code how this was done. We came to work on this while looking for a way to derive a cost model for fixing rule violations based on past fixes.

[1]http://findbugs.sourceforge.net
[2]http://pmd.sourceforge.net/

All these lines of research therefore share a common interest in identifying how the bugs or the rule violations are fixed in the source code. If there is a known solution to achieve this goal, it suffers from one limitation, it can only work for small commits that fix exactly one bug or one violation [5], [6], [7].

A recent study [8] showed that small commits (only one modification to the Abstract Syntax Tree of the system) are, in majority, related to bug fixes. However, it also found that about only 10% of commits were small. It does not mean that 90% of the commits are not related to bug fixes, but rather, that if they are, it might not be possible to identify exactly what modification fixed the bug since other modifications (other bugs, enhancements) could be mixed in the same commit.

In our experiments we found this to be also true for violations of good programming practices rules. Commits that fixed exactly one rule violation represented only 4% of all commits. If we filter the commits that do not fix any rule violation, then the ones fixing one rule violation still make only 27% of all commits that fix some rule violation.

Overcoming this problem would provide much more data to work on and significantly increase the validity and pertinence of the results on bug or rule violations fixing. In our case it was simply not possible to consider only commits fixing exactly one rule violation as there were too few of them leading to any meaningful result. And even these commits might not be suited for our studies as we discovered that they could still include other unrelated modifications. In our examples, we found that, on average, only 13% of the changes in the commits were responsible for fixing a given rule violation.

We therefore set to propose an automated solution to identify precisely the modifications within a commit that are needed for correcting a rule violation. The same solution could be applied to bug fixing providing there is a test that exercises the bug. We validate our approach and discuss its results on a set of rules applied to real world systems.

The remainder of this paper is divided into six sections and a conclusion: Section II gives more details on the motivation of this work; Section III gives an overview of the approach used to detect the sequence of actions needed to correct a programming rule violation; This approach is validated in Section IV and Section V on real world examples; In Section VI we discuss the threats to validity of the experiment; and finally, Section VII discusses related work.

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

II. Motivation

This paper aims at helping research in two related areas: How bugs or good programming rules violations are fixed.

A. Isolating Bug Fixing Actions

In [8], Martinez and Monperrus conducted an extensive study of 14 open-source Java projects to understand what kind of bug repair actions were used (*e.g.,* adding a method invocation, changing an if's conditional expression), with what frequency, ... Their study is based on very large corpus (close to 90 thousands commits) and thanks to this, they were able to come with some generic conclusions on the distribution of change actions.

However, as research in this field progresses, more and more studies will need to be performed on more and more specific fields. For example, one might very well imagine that a particular application field like aeronautics would have also change actions specific to it. As this happens, it will become more and more difficult to find relevant corpus, large enough to draw significant conclusions.

Already, in their study, the authors found that very small changes, including only one AST change, represented only 11% of the total population of changes. If this does not rule out the importance of the other 89% of changes as provider of meaningful information, it would make it very difficult to exploit with the current approaches (see Section II-C) because these approaches require that one change fixes one bug.

B. Isolating Rule Violation Corrections

Rule checkers [9], [10], [11], [12], [13] are tools that check whether source code violates some rules of good programming practice. For example, rules can specify an upper bound for method size [14], perform flow analysis to verify that a stream is properly closed [15], or check the correct use of an array to avoid out of bound accesses. Rule checkers thus raise alerts (or warnings) each time one of their rules is violated. Correcting alerts improves the quality of the target system [16] and may prevent future bugs [17]. The systematic use of these tools facilitates and improves software maintenance [18].

However, not all alerts are corrected by the developers. Some may be ignored because they would be too difficult to fix or because the developers do not agree that they represent an instance of bad code. On real systems, a rule checker may commonly raises thousands of alerts, many of which are never fixed. In fact, between 35% to 91% [19], [20], [21], [22] of reported alerts are never fixed. Some approaches assume that these un-fixed alerts might be false positives and propose techniques to filter them out. We studied these techniques in a previous publication [1].

To understand better this issue and work toward proposals that could improve the situation, we wished to perform a closer study of the modifications required to fix alerts.

C. Outline of the Envisioned Solution

To automatically find out what modifications are required to fix a given good programming rule violation, we planned to use the traditional approach of identifying the source code responsible for fixing an alert or a bug (used for example in [4]) by looking into past modifications of a system:

- Take two successive versions[3] of a system (*i.e.,* the versions just before and just after a commit);
- Identify the changes applied on the source version to obtain the target version.
- Run the rule checker on each version;
- Compare the two resulting sets of violations;
- If one violation is corrected between the two versions we know the actions required for that are in the set of changes applied.

A similar algorithm can also be applied to discover the modifications responsible for fixing a bug in the source code. This can be useful, for example to build a catalogue of possible repair actions [8]. To identify what commits contain bug fixes, two methods can be used, one consists in searching in the commit messages markers such as: "issue", "bug", "fixed", as well as references to bug report "#1234" [17], [8]; the second solution is to have a test that illustrates the bug, and to run this test on various versions of the software system until one identifies the commit that makes the test turn green. However, in either cases, one still needs to identify the changes in the commit that correct the bug. This might not be as simple as the preceding algorithm suggests:

1) From one version of the system to another, many violations or bugs can be corrected (as well as many new one introduced);
2) Only a small subset of all the changes applied from one version to the other is typically required to correct the alert or bug.

We found the first issue to be especially true for alerts that are not actual bug but good programming practices. It is not uncommon that, in an effort to improve software quality, several alerts be fixed in a single commit.

To fight against the second point, one might try to focus on small changes [5], [6], [7]. In their study [8], Martinez and Monperrus analysed very small commits that consisted in only one change to the Abstract Syntax Tree of the system. They report that 95 of 144 small commits (66%) were bug fixes and conclude that very small commits can generally be considered as bug fixes. Yet, in the same study, they found only 6,953 very small commits out of a total population of 62,179 (11%). This implies that 89% of the commits would have to be ignored, even though they might also contain bug fixes or, in our case, alert fixes. Which brings us back to the problem that in a large change, only a subset of all modifications might be required to correct a given violation.

III. Our Approach

A. Some Definitions

Before presenting the approach, we will introduce some definitions:

[3]We make no difference, in this paper, between the notions of version and revision in version control systems

Change: The set of *modifications* made on source code to go from one version of the source code to the next one. This could also be termed a patch. In our case, every commit in the version control system will correspond to a *change*.

We are specially considering changes that fix at least one alert (this is tested by running the rule checker on the source and target versions of the change). We assume that a change is complete in the sense that it contains all the modifications required to fix an alert (even though it can introduce new unrelated alerts). For example if one modification adds a parameter to a method definition, other modifications must alter the invocations of this method to take the new parameter into account.

Modification: For this first experiment, we worked with large-grained modifications that do not go below the level of method. This means changing anything in the definition of a method will result in only one modification, that of the method's body. It will not allow us to identify the specific Abstract Syntax Tree modification that corrected a violation, but makes little difference on the validity of the algorithm. We come back to this below in the definition of *Violation*. Note that we will consider adding, removing or modifying of comments as modifications. This is suitable because we are working with rule alerts and some rules do check comments (to see if there are enough of them for example). If we were working with bug correction, we would not need to consider comments as they have no incidence on the behaviour of the program.

Sequence: An ordered collection of *modifications* applied on the source code.

Valid sequence: A *sequence* (of *modifications*) that can be applied without error and that can fix a given alert without introducing new ones (to some extent to be detailed later in this paper). Because we work specifically with *changes* that fix a given alert, by construction for any *change* there is at least one *valid sequence* that will fix the considered alert (this *valid sequence* being, at worst, the *change* itself).

Coming back to the definition of *change* above, our hypothesis that a change contains all the modifications required to fix an alert can be translated into: a *change* is a *valid sequence*.

Violation: (or Alert) A violation of a programming rule detected by a static rule checking tool. Two violations in two different revisions of the code are considered equivalent if they pertain to the same programming rule on the same software artifact. In this first experiment, it is enough to identify software artifacts by their fully qualified name (package.class.methodSignature) because we are working at a large grain of modification. In the future, we will need to work at the level of AST modifications which will make it more difficult to pair together violations in two different revisions. This identification of software artifact will not allow us to take into consideration renaming of software artifacts (that will be considered as one removal and one addition). At this point, we did not want to deal with renaming. Detection of renaming is difficult and subject to interpretation, such additional manipulations could have polluted the result. An extra filtering that will be discussed shortly (ignoring removed artifacts) will ensure that renamings will actually be ignored altogether in the experiment.

Fixed Violation: A violation present in the base revision of the change but not in the target revision. We distinguish two cases: the software artifact responsible for the violation is found in both revisions of the change (before and after) or it was deleted in the target revision. We decided to ignore violation "fixed" because the software artifact that caused them was removed. As we just saw, for renaming of entities this will be a desired behavior. The rationale behind this decision was that research showed that not all programmers care about rule violations [19], [20], [21], [22], therefore a rule violation may go unnoticed for an extended period of time before it simply vanishes because of unrelated changes in the source code. We wished to lower this risk by ensuring, at least, that the software artifact exhibiting the rule violation was still present, and therefore that the rule violation was actually corrected in one way or another. Some violations do require to remove the software artifact on which they occur, for example for a rule that detects that a method has the same implementation in a super-class and a sub-class, but we preferred ignoring them here to provide a clearer understanding of the properties of our algorithm. We come back on this issue in Section VI.

B. Overview of the Approach

Our proposed solution consists in applying sequentially, randomly chosen, modifications from the commit to find out which one(s) is(are) required to fix an alert. Because the modifications will be applied in random order, three problems need to be considered:

1) First, it can be impossible to apply one modification before applying another one (like trying to create a method before creating the class that owns it);
2) Second, one might apply some unrelated modifications to the violation before coming to apply the modification(s) that are required;
3) Third what are we to do if a modification fixes the violation and introduces a new one?

Of these three issues, the first one is easy to overcome. If it is impossible to apply one randomly chosen modification (because it requires another one to be applied before), we simply ignore the current *sequence* and restart generating a new one from scratch. Another solution would be to backtrack in the construction of the sequence, choosing other modifications, but it does not seem worth the effort. Finally, a third solution would be to compute dependencies between the modifications, and prohibit in the random selection to pick a modification that depends on another modification not yet applied. We will monitor in our formal experiment the existence of such a problem.

The second problem related to the selection of unwanted modifications in the sequence before introducing the ones needed to fix the considered alert. This is solved using a hill-climbing approach. We start with a change that is known to be a valid sequence (founding hypothesis) and randomly select modifications in it to build another valid sequence. If the new valid sequence is shorter than the preceding one, we use it as a starting point for the next iteration. If the valid sequence

size does not decrease for a given number of iterations (we chose 20), we assume we found the minimal valid sequence that fixes the alert.

Note that there are numerous possible ways to fix an alert, and it is quite possible that in a given change, several different valid sequences would fix a given alert. Consider for example an alert stating that some method is not sufficiently commented, and a change where two different comments are added to the method. Any one of these two modifications (adding two comments) could be enough to fix the alert. We do not see this as a problem as exhibiting any of the two modifications as a solution to fix the alert would still be true, even though we would miss another possible solution to the same alert.

Finally, the third problem to consider was what to do if a modification fixes the alert considered but introduces a new one. Our solution consists in not considering valid sequences that create new alerts or result in non-compilable code. If a new alert (or a compilation error) is created by a modification in the sequence being built, we will add this new alert to the set of alerts we wish to fix, so the sequence will not be considered valid until this new alert is also fixed. The only exception is if the new alert also exists in the target version of the change, because it means that applying the entire change would, anyway, not remove it.

C. Algorithm

The final algorithm is presented in two parts for the sake of readability.

We first have a function that given a set of changes and a violation to fix, will compute a sequence of modifications that fixes this violation (see Figure 1). In other words, it returns a *valid sequence* from a *change* (but the sequence might not be minimal). For simplicity the function may also return error, for example in case it tries to apply modifications in an incorrect order.

This first function is then called iteratively following an hill-climbing method to find the *Minimal Valid Sequence* (MVS) that fixes the considered violation (see Figure 2). This part can end on one of two conditions. First if the length of the change is 1 modification, we know that we have reached the MVS. Second we set a threshold on the number of identical sequences returned by FINDVALIDSEQUENCE before deciding it will not be able to improve it (and therefore it is considered to be minimal). In our experiments we used a threshold of 20, so after that number of calls to FINDVALIDSEQUENCE without diminution in the length of the returned sequence, we stop looking for a shorter one. Note that fixing this threshold to the lower acceptable value would be a possible improvement. One must remember that running the algorithm involves modifying the source code and recompiling it; what is not a light weight operation. This is done in the loop of FINDVALIDSEQUENCE which is called by FINDMINIMALVALIDSEQUENCE. Assuming it does not result in halting the iterations too soon (and thus not finding the minimal valid sequence), reducing this threshold of any amount would reduce of the same amount the total number of calls to FINDVALIDSEQUENCE. This can be interesting, but may not be all that significant. We will see that in our experiments, we had several violations requiring more than 100 iterations to find the minimal valid sequence (see Section V-C, RQ3a). Removing 5 or 10 iterations to these might not make that much a difference. We will look more in details into this issue.

```
FINDVALIDSEQUENCE
INPUT: chg, a list of modifications from which to extract a
    valid sequence
INPUT: viol, a violation of interest to fix
INPUT: src, initial state of the source code (will be modi-
    fied by incremental application of modifications)
INPUT: ignore, a set of violations that can be ignored
RETURN: seq, a valid sequence that fixes viol or an ERROR
LOCAL: mod, a modification in chg
LOCAL: toFix, a set of violations that should be fixed

toFix ← { viol }
seq ← {}
while toFix ≠ ∅
    mod ← a random modification from chg
    apply mod on the current source code
    if impossible to apply mod
        return ERROR
    end if
    compute violations introduced and/or removed by
        applying mod
    remove from toFix the violations corrected by mod
    add to toFix the set of violations introduced by mod
        minus ignore
    remove mod from chg
    add mod to seq
end while
return seq
```

Fig. 1. Algorithm to find a valid sequence fixing a given violation from a list of modifications

We added a condition to prevent the algorithm looping endlessly on error condition. We defined that if 100 consecutive iterations resulted in an error, then we drop the entire commit and stop looking for valid sequences that would fix any of its violation(s). Similarly if 200 non consecutive iterations result in an error, we stop. The idea is that some commits may have modifications that can appear only in a very specific order. In theory, given sufficient time and tries, our algorithm should be able to find the minimal valid sequence fixing the violation(s) of this commit. However, in practice this would be too long for a viable experiment. This issue is linked to the way we choose our modifications randomly and how we treat errors. Both issues will be treated jointly.

This algorithm assumes we can safely apply changes to a source code (in FINDMINIMALVALIDSEQUENCE) and then come back to the initial state of the system (conventional version control tools could be used for that).

IV. VALIDATION EXPERIMENT

The context of the experiment is real systems for which we can access the source code history. We chose to experiment on programming rule violations because they were our initial target and it is easier to find automatic tools that will check

```
FINDMINIMALVALIDSEQUENCE
INPUT: comit, the modifications of the considered commit
INPUT: viol, a violation of interest fixed by the commit
INPUT: src, initial state of the source code (before applying
  the commit)
RETURN: seq, the sequence that fixes viol or an error
LOCAL: chg, the set of modifications from comit still
  candidate to be part of seq
LOCAL: ignore, a set of violations that can be ignored
LOCAL: prevSeq, the length of the previous sequence found
LOCAL: thresh, counter checking how many sequence of
  same length we found
LOCAL: conseqError, counting how many consecutive
  iterations resulted in error
LOCAL: totalError, counting how many iterations
  resulted in error

chg ← comit
seq ← {}
thresh ← 1
conseqError ← 0
totalError ← 0
ignore ← set of violations introduced by applying comit
  on src
prevSeq ← length(comit)
while ( (length(chg) > 1) and (thresh < 20)
  and (conseqError < 100) and (totalError < 200) )
  seq ← FINDVALIDSEQUENCE(chg, viol, src, ignore)
  if seq is not ERROR
    if length(seq) < prevSeq
      thresh ← 1
    else
      thresh++
    end if
    chg ← seq
    prevSeq ← length(seq)
    conseqError ← 0
  else
    conseqError++
    totalError++
  end if
end while
return seq
```

Fig. 2. Algorithm to find the minimal valid sequence fixing a given violation from a commit

the presence or not (correction) of a violation. Bugs would be more difficult to experiment with because not all bugs have an accompanying test that exercises them, and when they do, it might not be easy to relate a bug (*e.g.,* in an issue tracker system) to the test. Therefore, we also require a system with some rule checker tool available as well as rules.

We will validate our solution on Moose [23] an open-source, real-world, and non-trivial system, with a consolidated number of developers and users. For us, the system presents the additional advantage that we are part of the development team and are able to validate the quality of our solution's results. This is not a bias as we are working on past commits, anterior to this research and we are not influencing how the minimal valid sequence is extracted, we are only validating it. Since Moose is written in Pharo[4], a Smalltalk inspired language, we selected SmallLint [14], the most adopted Smalltalk code analysis tool. We give some statistics on the system in Section IV-A.

We selected in the history of Moose the commits that solved some rule violations by systematically exploring all the revisions of the system during the considered period:

1) load one revision of the system;
2) run the rule checker tool on it (store the rule violations);
3) apply the commit immediately following this revision;
4) apply the rule checker again and compare the rule violations with the previous one;
5) if some violations were removed, then this commit fixed them in some way
6) further verify that the violation is not removed because the software artifact carrying it was deleted (see discussion in Section III on *Fixed Violation*.)

A. Descriptive statistics

Before presenting the results, we must explain that all variables measured in our experiments follow a distribution that is not normal but with a heavy tail[5], therefore we report medians rather than means.

Our case study has more than 2,500 classes, 210 KLOC, 21 known contributors, and a history of development going back more than 10 years. We will report here on an experiment during four days, on more than 368 violations coming from 65 commits.

B. Research questions

We validate our approach in three steps: First, we will validate our initial hypotheses (RQ1), second, we evaluate whether our approach can actually discover the sequence of source code changes required to fix a given rule violation (RQ2), third, we consider additional points on our algorithm as how we treated errors (RQ3).

For the first question, we have three hypotheses to validate:

RQ1a. A commit may contain more modifications than required to fix any of the violations?
RQ1b. A commit may fix several violations?
RQ1c. A commit may fix violations of several rules?
RQ1d. Focusing on small commits would discard too much information?

For the second question (RQ2), we will simply verify that our algorithm found the correct Minimal Valid Sequence (MVS).

For the third question we will consider the following points:

RQ3a. Is our treatment of errors adequate? (When one modification chosen randomly cannot be applied because

[4]http://pharo-project.org

[5]This is very common in software engineering [24].

another one is needed that has not yet be applied, we simply drop the current sequence being built)

RQ3b. Is our threshold of 20 consecutive iterations with the same solution appropriate to decide that we found the minimal valid sequence?

RQ3c. Additionally, what is the time required to find a minimal valid sequence?

Another aspect that we could have considered is whether more than one source file may be affected by a fix. To illustrate this, let us consider languages that allow spreading the definition of a class over several files (ex: C++ with the .cc/.hh files, or C# with the partial definition of classes). In such languages, adding a method can imply modifying more than one file. This is an issue that we are not able to monitor in this experiment because we used the Pharo environment. In this environement files are hardly used at all. Classes and methods are created within the Pharo environment, they are saved when the environment is saved, and loaded once more when the environment is loaded. Given the the answers to RQ1b and RQ1c, we expect that fixes in general could touch several files, and this would be handled correctly by our algorithm.

V. Experiment Results

A. RQ1. Validating hypotheses

RQ1a: A commit may contain more modifications than required to fix any of the violations?

We manually evaluated 87 fixed rule violations chosen randomly and look at the size of the resulting minimal valid sequence (Table I). The median size per commit was 131 modifications with a maximum of 133. The median size of the minimal valid sequences is 1 with a maximum of 11. These numbers clearly show that commits may contain more modifications than just the ones required to fix a violation.

TABLE I. Size (number of modifications) of commits and Minimal Valid Sequences

	median	max.
commit size	131	133
MVS size	1	11

RQ1b: A commit may fix several violations?

We experimented with 65 commits containing fixes for 368 violations. The maximum violations fixed per commit is 75, and the median 2 (see Table II). 21 of the 65 commits (*i.e.,* 32%) fixed only one violation. This does confirm our hypothesis that commits may fix more than one violation. Consequently, it seems to confirm also that there is a need for a tool to isolate the modifications fixing a given violation (note that the same modification can fix several violations, as discussed in [25]).

TABLE II. Number of violations fixed per commit

	#	median	max.
total commits	65		
total violations fixed	368		
commits fixing one violation	21		
violations fixed / commit		2	75

RQ1c: A commit may fix violations of several rules?

Based on the 65 commits used for the experiment, we report in Table III the number of rules fixed by commits. The minimum is for commits fixing violation(s) from 1 rule, the maximum is commits fixing violations from 7 different rules, and the median is commits fixing violations from 2 different rules. These numbers clearly show that not only do commits fix more than one violation (see RQ1b), but also these violation may come from more than 1 rule.

TABLE III. Number of different rules fixed per commit (min.=1, median=2, max.=7)

total commits	65
commits fixing violations from 1 rule	29
commits fixing violations from 2 rules	18
commits fixing violations from 3 rules	6
commits fixing violations from 4 rules	5
commits fixing violations from 5 rules	3
commits fixing violations from 6 rules	3
commits fixing violations from 7 rules	1

RQ1d: Focusing on small commits would discard too much information?

For the 65 commits in our experiment, the median size of commit is 15 modifications with a maximum of 320. The numbers show that commits are not small, a median of 15 modifications per commit is very far from the one-AST-change definition of small commits in [26].

From the results for RQ1, we deduce that our initial hypotheses are valid and there is a necessity for solutions like ours to precisely understand the fix of a violation.

B. RQ2. Validating our algorithm

For our main validation, we use again the 87 rule violations that we evaluated manually. In 84 cases (97%), the algorithm found the correct minimal valid sequence that fixed the violation considered. In the remaining 3 cases, the sequence found was not minimal, but the search stopped because we reached the threshold of 20 consecutive identical solutions (see RQ3b).

We conclude that our algorithm is working although there is a balance to reach between ensuring we find the right minimal valid sequence and not doing too much computation uselessly.

C. RQ3. Additional discussion on the algorithm

RQ3a: Is our treatment of errors adequate?

This initial algorithm chooses randomly modifications to put in the sequence and drop the sequence if some modification cannot be applied. Further more, for experimental reasons, we dropped all commits that had more than 100 consecutive errors or more than 200 non consecutive errors. On the 65 commits, 5 (8%) had more than 100 consecutive errors and 5 others had more than 200 total errors (see also Table IV). These commits with errors contained 89 violations. Interestingly enough, for two of these commits, we successfully solved some of the violations (*i.e.,* they produced less than 100 errors), and dropped the rest because of too many errors on another

violation. We actually dropped 85 violations in total for the 10 commits and 4 violations for these 2 special commits could find a solution.

TABLE IV. OCCURRENCES OF ERRORS IN OUR EXPERIMENT

total commits	65
total violations	368
total violations solved	283
violations solved with some error	33
commits dropped (too many errors)	10
violations they fixed	89
violations dropped	85
violations solved	4

This shows that dropping the entire commit is not always a good idea. We did it only for experimental reasons, to gather enough results in a reasonable amount of time.

On the remaining 55 commits (+ 2 partial) that fix together 283 violations, 33 violations (12%) produce some error when looking for the minimal valid sequence that fix them. The median for the violation with some error is 5 errors per violation, with a maximum of 57.

We conclude that treatment of error is one direction on which we still need to work. Of course the real discussion would be how to avoid altogether these errors. We discussed some possible solution in Section III.

RQ3b: Is our threshold of 20 consecutive iterations with the same solution appropriate to decide that we found the minimal valid sequence?

TABLE V. IMPACT OF ERRORS ON OUR EXPERIMENT

violations solved	283
MVS with 1 modification	189
MVS with >1 modification	94

First (table V), 189 of the 283 violations (*i.e.,* 67%) for which the algorithm proposed a solution have a minimal valid sequence with only 1 modification. These did not require the threshold of 20 similar solutions to stop the search because 1 is the absolute lower bound for a valid sequence. We measured within these violations the longest succession of iterations with the same valid sequence found. In our experiment, the absolute longest succession was 17 successive iterations with the same valid sequence before a shorter one was found. The median length of such succession is 1.

On the other hand, as we already said in Section V-B 3 of the 87 MVS that we evaluated manually were not correctly found by the algorithm found because the search stopped on the threshold of 20 consecutive identical solutions. This seems to indicate that the threshold might be too low. But the random nature of our algorithm impedes us to guarantee that we will always find the minimal valid sequence. Moreover, setting it at a higher value would me spending more time finding the exact MVS.

RQ3c: What is the time required to find a minimal valid sequence?

On the 368 violations in our experiment, the median execution time of FINDMINIMALVALIDSEQUENCE was 3 minutes 50 seconds and the maximum time 3 hours 58 minutes. Our experiment with 293 violations lasted about four days on a current commodity machine.

Obviously the execution time is linked to the number of iterations (calls to FINDVALIDSEQUENCE). Once again, it could be improved by improving the treatment of errors (modifications that could not be applied) and by using a smaller threshold to accept a recurring valid sequence as minimal.

However, one must consider that this algorithm would be applied only once at the beginning of a study to get data on which to work. Therefore, long execution time if it is not desirable, should not be a blocking issue.

VI. THREATS TO VALIDITY

Apart from the obvious Internal Validity threat from an error in the tool we programmed, in the historical data extracted, or in the rule checker program, there are two External Validity threats:

1) The system studied might not be entirely representative of a larger population of systems, either from another application domain, or written in another programming language. This is always a difficult threat to mitigate as there is little information on what property of a system is important to ensure representativeness. Replication of the experiment for other systems must be realized.
   This said, we strongly believe our approach is independent of the programming language and the application domain. We also believe Moose is a credible, real world, non-trivial, case study. It is a medium to big system (2,500 classes, 210KLOC[6]), and it includes a significant number of versions
2) We disregarded violations that were fixed because the software artifact where they appeared before the commits had been removed during the commit. This was done to eliminate a possible problem with the obviously simple solution that removing the artifact that raises an alert is the best way to "fix" any violation. Some violation can really be fixed this way, but this would not be the case for all, and we lacked the mean to tell one from the other.

Although we did not experiment with them, it seems reasonable to suppose that the same algorithm could be used to isolate modifications responsible for bug fixes. This would of course require that we have some oracle (a test exhibiting the bug) to tell us whether the bug is present or not in a given version of the system. Yet because we experimented with rule violation fixes, this claim should be made with some caution.

VII. RELATED WORKS

Different studies have already been done on programming rules violations and rules checker tools.

Heckman *et al.* [27], synthesizes available research results on ranking algorithms that try to estimate whether an alert is a valid one (that should be corrected) or not. Different algorithms are presented and compared according to different criteria like

[6]Moose is written in Pharo which is a concise language

information used, or algorithm used. In a related work Allier *et al.* [1], compared the same algorithms on their results on real case studies. This line of research tries to remove false positives from the alerts raised by the rule checker. It is a "pre-fixing" approach. We try to identify how a given alert was corrected.

Some approaches were proposed to study the relation between alerts and bugs. Boogerd *et al.* [28], [29] empirically assess the relation between faults and violations of coding standard rules raised by MISRA C, using coding standard rules for embedded C development on industrial cases. They found that only 10 out of 88 rules for the case study presented in [29], and 12 out of 72 in rules for the case study presented in [28] were significant predictors of fault location. Basalaj *et al.* [30] studied the link between QA C++ warnings and faults for snapshots from 18 different projects and found a correlation for 12 out of 900 rules. Wagner *et al.* [31] evaluated two Java bug-finding tools (FindBugs and PMD) on two different software projects, in order to evaluate their use in defect-detection. Their study could not confirm this possibility for their two projects. Couto *et al.* [2] also showed that overall there is no correspondence between the static location of the warnings raised by FindBugs and the methods changed by software maintainers in order to remove defects. Tracy *et al.* [32] show that some code smells have a significant but small effect on faults. They investigate the relationship between faults and five code smells. They developed a tool to detect these smells and built Negative Binomial regression models to analyse the relationships between smells and faults. In [17], Hora *et al.* studied the relevance of generic rules against that of system (or domain) specific ones. They concluded that system specific rules where more likely to be fixed. All these studies concentrate on the usefulness of rule checking to improve the quality of the code, specifically considering the co-occurrence of alerts and bugs. Our long term goal is to understand how alerts were corrected. One possible application of this would be to establish some cost model for correcting future similar alerts.

The number and the cost of bug fixes in industrial practice motivated the search to automatically minimize the effects of defects in software systems. Program fault repair consists generally in three steps: fault localization, patch generation, and patch validation. Automated program fixing is an active area of research [33], [34], [35], [36]; the goal of this domain can be to catch faulty behavior of a program just before it happens and subdue its effects [37], or transform the program or execute it in a way that excludes certain types of behaviour [38], or generate patches to the source to prevent a class of bugs or fix a particular bug [39]. To fulfill this goal number of approaches to detect repair action was proposed. Martinez et Monperrus [8] mine program history and related artefacts to suggest repair patches templates based on a fine-grained AST level. They filter transactions to retain only those related to bug fix by mining the messages in the transaction and in this set they select only the one which contains one modification. But our approach can find in a transaction with large change the modification related to programming rule fix. Nguyen *et al.* [40] propose an algorithm based on genetic programming to generate patches. The algorithm maintains a population of chromosomes (programs), selects a pool of individuals based on their fitness (score according to number of pass and fail test), and modifies them with mutation and crossover operators until reaching a terminating criterion.

Other publications are more related to the pertinence of the domain. For example, Zheng *et al.* [18] are following the GQM approach[7] paradigm to determine whether rule checkers can help an organization to improve the economic quality of software products. Their results indicate that rule checkers are an economic complement to other verification and validation techniques.

VIII. Conclusion

In this paper, we have presented the idea that one can mine violation fix actions from software repositories. In other words, one can extract from past, a sequence of modifications needed to fix an alert (*e.g.,* adding a method call, deleting method). This can be made difficult by three issues:

- One commit can fix several violations and/or introduce new features. In this case, one has to extract only those modifications responsible for fixing the alert considered;
- On modification required to fix an alert might introduce another violation. In this case, one as to decide whether fixing this new violation is part of fixing the first one or not;
- The modifications are not independent and one has to find the correct order in which they need to be applied.

We presented a methodology that answers to these considerations and we applied it on a real system. The results are that, for 283 out of 368 evaluated violations, we could find automatically the sequence of modifications that fixed them. We also identified a solution to deal with the 85 remaining violations that our algorithm dropped because too many errors were generated when randomly selecting modifications to be applied. The solution should be based on constraining the random choice of modification to apply to those that can actually be applied at a given time.

Another extension of the work would be to deal with finer grained modifications, typically at the level of the AST. From this, we could start to explore whether it would be possible to build on this knowledge to deduce some suggestion for fixing a given rule violations. The idea would be to extract an abstract summary of all the violation fixes for the given rule.

ACKNOWLEDGMENT

This research has been supported by a grant from Inria, France

References

[1] S. Allier, N. Anquetil, A. Hora, and S. Ducasse, "A Framework to Compare Alert Ranking Algorithms," in *Working Conference on Reverse Engineering*, 2012.

[2] 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.

[7] Goal/Question/Metric, a process used to define a set of metrics to answer an abstract question.

[3] C. Couto, P. Pires, M. T. Valente, R. Bigonha, A. Hora, and N. Anquetil, "Bugmaps-granger: A tool for causality analysis between source code metrics and bugs," in *Proceedings of the 4th Brazilian Conference on Software: Theory and Practice (CBSoft'13)*, 2013. [Online]. Available: http://rmod.lille.inria.fr/archives/papers/Cout13a-BugMapsGranger-CBSoft13.pdf

[4] A. Hora, N. Anquetil, S. Ducasse, M. Bhatti, C. Couto, M. T. Valente, and J. Martins, "Bugmaps: A tool for the visual exploration and analysis of bugs," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12) - Tool Demonstration Track*, 2012. [Online]. Available: http://rmod.lille.inria.fr/archives/papers/Hora12a-Official-CSMR2012Tools-BugMaps.pdf

[5] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API Evolution under Control," in *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*, 2014, pp. 420–424.

[6] 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.

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

[8] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, pp. 1–30, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9282-8

[9] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software–Practice & Experience*, vol. 30, pp. 775–802, jun 2000.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *Conference on Programming language design and implementation*, 2002, pp. 234–245.

[11] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software*, vol. 19, pp. 42–51, 2002.

[12] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved, "SABER: Smart Analysis Based Error Reduction," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 243–251, jul 2004.

[13] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Goanna - A Static Model Checker," in *FMICS/PDMC*, 2006, pp. 297–300.

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

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

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

[17] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain specific warnings: Are they any better?" in *International Conference on Software Maintenance*, 2012, p. *to appear*.

[18] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, apr 2006.

[19] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," in *International Symposium on Empirical Software Engineering and Measurement*, 2008, pp. 41–50.

[20] 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.

[21] T. Kremenek and D. Engler, "Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," in *International Conference on Static Analysis*, 2003, pp. 295–315.

[22] C. Boogerd and L. Moonen, "Prioritizing Software Inspection Results using Static Profiling," in *International Workshop on Source Code Analysis and Manipulation*, 2006, pp. 149–160.

[23] M. U. Bhatti, N. Anquetil, and S. Ducasse, "An environment for dedicated software analysis tools," *ERCIM News*, vol. 88, pp. 12–13, Jan. 2012. [Online]. Available: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf

[24] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 1–26, sept 2008.

[25] A. Lozano, G. Arévalo, and K. Mens, "Co-occurring code critics," in *SATToSE 2014—Pre-proceedings*, V. Zaytsev, Ed., July 2014, pp. 10–13. [Online]. Available: http://grammarware.github.io/sattose/SATToSE2014.pdf

[26] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with AST analysis," *CoRR*, vol. abs/1309.3730, 2013.

[27] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, pp. 363–387, apr 2011.

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

[29] ——, "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.

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

[31] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An Evaluation of Two Bug Pattern Tools for Java," in *International Conference on Software Testing, Verification, and Validation*, 2008, pp. 248–257.

[32] D. B. Tracy, Min Zhang and H. Yi Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, p. 33, 2014.

[33] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, Jun. 2011.

[34] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *Proceedings of the 13 IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, ser. CHARME'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 35–49.

[35] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374.

[36] J. A. Jones, "Semi-automatic fault localization," Ph.D. dissertation, Atlanta, GA, USA, 2008, aAI3308774.

[37] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 233–244.

[38] R. L. Alexey and S. Tzi-Cker Chiueh, "PASAN: Automatic patch and signature generation for buffer overflow attacks," in *In Systems and Information Security*, 2006, p. 165–170.

[39] Bassem and E. Sarfraz Juzi, "A tool for repairing complex data structures," in *In International Conference on Software Engineering*, 2008, p. 855–858.

[40] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest, "Using execution paths to evolve software patches," in *International Conference on Software Testing, Verification and Validation Workshops, ICSTW '09*, P. McMinn and R. Feldt, Eds., Denver, Colorado, USA, 1-4 Apr. 2009, pp. 152–153.