LEARNING FROM HUMAN REPAIRS THROUGH THE EXPLOITATION OF SOFTWARE REPOSITORIES

Hayatou Oumarou and Kolyang

Department of Computer Science, HTTC, University of Maroua, Cameroun

ABSTRACT

Software systems drive our phones, cars, banks, cities, etc. This places a heavy load on software industry because all these systems must be continuously updated, corrected and extended when users and consumers express new needs and high quality software is now perceived as a "must have" rather than "should have". Developers has to do their best to assure that their code has minimal defects.

Nowadays technologies evolve rapidly, software suffer from some majors problems. Firstly software maintainers have to deal with the past (past languages, existing systems, old technologies). It is often misunderstood and treated as a punishment. Secondly developers have to ensure a good software quality. To deal with such problems, approaches have been proposed to support software maintenance and reduce the efforts of developers. One approach commonly adopted is the usage of rules to ensure source code quality and ease maintenance activities.

We propose in this paper an approach to ease corrective maintenance task and help to improve software quality by exploiting human knowledge available in software repositories. This approach support remediation of bad situations; by reducing for example time spent to read, understand and how the code can be changed.

1. INTRODUCTION

Software developers deal with day-to-day defects during the development and maintenance phases. Studies have shown about 80% of the cost of software is devoted to maintenance. This phase play a key role in the lifecycle of the software product [1]. One of the major tasks during maintenance is to fix defects. The fix is generally committed in a software repository, which registers each modification made on the program. However it happens that in addition to the fix, the developer adds refactorizations, optimizations or even fixes several defects in a single commit.

Software quality can be measure by some principles or/and absence of defects. Fixing defects increase software quality. Software quality models are based on a set of metrics and rules (principles) to evaluate the quality of the software. We are interested in studying whether we could help correcting these defects.

We came to work on this while looking for a way to derive a cost model for fixing programming rule violations based on past fixes. The main interest of this work is to identify how defects 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 defects [2]. A recent study [3] 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. And even these commits might not be suited for our studies as we discovered that they could still include other unrelated modifications. It does not mean that 90% of the commits are not related to defects fixes, but rather, that if they are, it might not be possible to identify exactly what modification fixed the defect 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 defects.

In this work we propose a new approach to find in a commit the modifications that are necessary to fix a given defect. Our main motivation is to know which source code modification (s) is (are) necessary to fix a defect. This can help developers to understand similar defects and fix them easily. The remainder of this paper is divided into six sections and a conclusion : Section II present software repositories, Sections III present software quality and gives more details on the motivation of this work; Section IV present formalisation of defect in software. Section V gives an overview of the approach used to detect the sequence of actions needed to correct a defect; This approach is validated in Section V on real world examples; and finally in Section VI we discuss the threats to validity of the experiment.

2. SOFTWARE REPOSITORIES

Many techniques propose to mine data present in the repositories to help the efforts of software engineering. For example, Stephen G Eick et al.[4] measures the "code decay", Graves Harvey et al.[5] predict defects in software modules, and Thomas Zimmermann [6] predicts artefact changes by mining repositories.

Although their techniques and final objectives vary, most approaches working on software artefacts changes are based on the analysis of *change set* or group of modifications that are part of the same commit. This is motivated by the fact that modifications within a given commit often correspond to development work that has been performed as part of a single, coherent task, such as correcting a defect or improving functionality. Unfortunately, approaches based on *change set* can not guarantee that data extracted from a repository actually belongs to a single, coherent task.

We found that *change sets* often violate this assumption when they contain non-essential modifications. A non-essential modification is normally not associated with a specific software development or a specific purpose such as correcting a defect targeted by the commit. We offer an automated technique for detecting essential modifications in a commit to fix a given defect within the commit.

2.1 Software repositories presentation

To better formulate our problem in the rest of the document, we first present our assumptions about software repositories and the kind of information found there.

For our analysis, we need all the modifications made to the software. We obtain this data by mining software repository. A repository provides a good basis for learning the history of defects in the software lifecycle. In a computer project, the majority of changes are applied to the source code. In addition to changes in source code, repositories keep meta information. The most significant are information on the author (name, email, ..), the timestamp and the log written to describe the commit. The repositories store the data as plain text or as binary file. They do not provide support for manipulation of syntactic or semantic structure contained in the files they store. These structures must be rebuilt after retrieving files from the repository.

For a given file *F*, the repository keeps an explicit copy of its latest version only. Let F_i this version, each previous version $F_{j\leq i}$ recorded in the repository is represented as a delta δ_j. Each δ_j describes the modifications between F_j and F_{j+1} in terms of text lines in both file versions. These deltas are typically computed using a line diff-based utility. Each time a developer registers a new version of a file F_i , the repository stores the computed δ_{i-1} and replaces the previous version of F_{i-1} with the latest version F_i . An explicit copy of an earlier version F_j can then be reconstructed and retrieved by the application of the deltas δ_{i-1}, ..., δ_j to the current version F_i. Therefore, in order to facilitate our further discussion, we will simply assume that each version of the file is directly accessible.

A repository also lacks information about the purpose of a commit: has it introduced a new feature or has it corrected a defect? Although it is possible to identify these reasons only with the commit log; we found in our analysis that developers do not generally mention this information in the log correction of programming rules.

2.2 Analysing Commits in Repositories

We are presenting a way to extract valuable information from defect correction activities. We try to find actions taken by developers to fix a defect by studying the history of the software through commits in repositories. A commit contains several information such as it creation date, the name of it author and a log to explains the purpose of the commit[7]. We therefore seek among these commits those that introduce defects, those fixing these defects and then try to find necessary modification(s) to fix these defects.

2.2.1 Mining Techniques and Tools

In 2001, Andrew Y Yao [8] introduced a tool to look for fragments of source code using feedback from a versioning system. This tool allows to find the most recent version of a code by looking in previous versions of a commit to better understand the current version.

In 2009, Fluri et al. [9] worked on the relationship between source code and commit log. They found that when the code and comments co-evolve, both are modified in the same revision. They found that 97% of comments changes are made in the same revision as the associated source code change.

2.2.2 Commit That Introduce Defects

Defects fixing commits are widely used to predict the occurrence of defects or to find the risky fragments of the software. However, a defect fixing commit does not contain information about which one initially introduced the defect. Defect fixing commits are relatively easy to retrieve than commit introducing defects.

In 2005, Sliwerski et al. [10] introduced an algorithm to retrieve commits introducing defects. To identify them, the algorithm first finds defects fixing commits by searching for relevant keywords in log, or an explicit link with an error tracking system. The algorithm then executes a *diff* tool to determine modifications in the defects fixing commits. The comparison tool returns a list of the regions that differ in the two commits; each region is called a piece. The algorithm computes for each piece in the source code, the most recent version in which the piece has been modified, and determines the developer who made the modification. The origins discovered are identified as commits that introduce defects.

In 2006, Kim et al. [11] propose an improvement of Sliwerski algorithm [10]. This approach is carried out in five stages:

1. Use annotation graphs to provide more detailed annotation information;
2. Ignore comments and blank lines;
3. Ignore format changes;
4. Ignore revisions that correct errors in which many files have been modified;
5. Manually check all changes use to fix defects.

In 2010, Sinha et al. [12] present an approach based on dependency graph to determine defects introducing commit. The main innovation of this approach is analyzing the effects of modifications in defect fixing commit on program dependencies and, based on these effects, identifies the version of the commit that introduces the default.

2.2.3 Defect Fixing commits

In 2000, Mockus et al. [13] propose an approach to identify defect fixing commits . Their approach is based on identifying keywords in log. It classifies a commit as "corrective" if the log message contains one of the following keywords: **fix**, **bug**, **error**.

In 2003, Fischer et al. [14] presented a historical database that contains commit logs and report information from error-tracking systems. To link the log message and the error tracking database, Fischer et al., searched for commit log messages that correspond to a given regular expression. Later, they improved the linking algorithm. Another work has studied the techniques of connection of the commits to the errors tracking systems. In 2009, Bird et al. [15] have shown that the result of commit linking techniques to error tracking systems can be biased because a developer may for example fail/ignore to enter a keyword in the commit log that corrects a defect. They showed that 54% of defect fixing commits are affected. For example, in 2010, Bachmann et al. [16] proposes a technique to find the missing links.

3. SOFTWARE QUALITY

In the literature, we find many definitions of software quality (ISO 8402 , 9126, 9001 ; [17]). It is clear from these definitions that software quality is define by objectives to be met in order to satisfy the real requirements of the stakeholders through a set of indicators. Some of these objectives can be modeled as rules detected by rule checkers such as PMD, JLint or FindBugs. We present in this section some quality model and the relations between rules and software quality.

3.1 Some Quality Models

In 1977, McCall [18] proposed a model to measure the quality level of a software for the US Air Force. This model is composed of three layers: metrics, criteria and factors. The 11 factors of quality of the software (Reliability, Security, Integrity, Ergonomy, Efficiency, Testability, Maintainability, Flexibility, Reusability, Portability, Compatibility). In 1991 ISO and IEC defined the ISO / IEC 9126 [19] standard based on the McCall model. The attributes of qualities are grouped into six general characteristics specifying a quality requirement, functional or not of the software. Each characteristic is organized into sub-characteristics. The standard proposes a series of measures / metrics to assess the conformity of the product with the formulated requirements.

This model improves the McCall model by providing measurement scales associated with metrics and also gives indications for measuring these characteristics and sub-characteristics.

In 1999, ISO and IEC propose ISO / IEC 14598. This standard groups together the evaluation characteristics according to the client, the developer and the evaluator and is available in several documents ISO / IEC 14598-1, 14598-2 , 14598-3, 14598-4, 14598-5 and 14598-6.

SQUASH [20] is a formalization of SQUALE resulting from the practices of Air France KLM, Qualixo and Peugeot SA. This model was proposed by Karine Mordal and the Inria RMod research team. The SQUASH model is hierarchical, based on the same principles as the McCall Factor-Criterion-Metric model and the ISO 9126 standard.

The SQUASH model consists of four layers: **measures, practices, criteria and factors**.

In 2005, the Square standard is proposed under the reference ISO / IEC 25000: 2005. It unifies normative documents covering the process of specification of measurement and evaluation requirements.

In 2009, the Software Engineering Institute (SEI) and the Object Management Group (OMG) proposed the CISQ Consortium based on the ISO / IEC 2500 standard. The aim of this set of metrics is to define a unique standard for finely measuring software quality for large systems and n-tiers systems. This standard aims to address the growing complexity of systems. The most used standards in software quality evaluation are the ISO / IEC 9126 and 14598 standards. In the following, our study will be based on these two standards.

3.2 Evaluation of Quality and Rules

The models used to evaluate the quality of software are based on metrics and measurements. These models set out quality rules, formal principles and provide guidance for assessing the level of quality on the basis of metrics.

The models of evaluation of the quality of the software are generally hierarchical. They define notions of high level of quality that they specify by principles of quality. There are generic models and models defined for a type of software. The generic models are derived from the McCall model and define concepts that must be adapted to the software type. Specific models on the other hand evaluate only one type of software: object-oriented software. However, all these models assess the quality of the code through good and bad practices expressed by rules. Good practices are expressed by rules to be followed while bad practices are principles to be avoided.

All these models are based on notes attributed to the practices by experts of the system studied according to their experiences. The use of violations of these practices (code technical rules) that can be extracted from system history in software repositories will improve the relevance of these notes.

We propose in this paper a method to extract in commits the actions needed to correct violations of practices. This would allow developers to spend less time to improve source code quality by learning from past actions and experts to have, in addition to their experience, data on which they can use to attribute notes to violations of practices.

4. Defects In The Software

4.1 Formalization of Defects in the Software

4.1.1 Coding Conventions

Coding conventions have been widely studied in the literature. According to Allamanis [21], coding conventions define a coherent syntactic style, promoting readability and thus maintainability. Coding conventions (naming, formatting decisions) determine the clarity of the source code, increasing code portability, accessibility to new coders, reliability and maintainability. Its application can avoid defects.

Oman and Cook [22] have found through empirical studies that the style used when writing or maintaining code has a direct impact on the quality of the software, particularly on the intelligibility and maintainability of a program.

Identifiers naming is at the heart of software engineering. According to the work of Broy et al. [23] identifiers represent about 70% of the syntactic elements of the source code. Haiduc and Marcus [24] have shown that approximately 40 % of domain-specific terms are used in the source code. The use of naming conventions is therefore a crucial practice to improve the comprehensibility of the source code.

4.1.2 Code Smells

In 1995, Bruce Webster [25] introduced the concept of bad smell in the context of object-oriented programming. He used the word "pitfall" for this purpose.

In 1996, Riel Arthur [26] introduced 61 heuristics, each characterizing a good practice of object-oriented programming.

In 1998, Brown et al. [27] described 40 design "smells". They grouped these smells under the term anti-Pattern. An anti-pattern according to their definition is a description of a solution to a problem that generates negative consequences. Among the best known are the spaghetti code.

In 1999, Fowler Martin [28] describes 22 defects that correspond to structures in the source code that suggest a possibility of refactoring. Refactoring is a technique used to improve the internal structure of a system without altering its external behavior by making changes in the source code.

In 2003, Mäntylä et al. [29] proposes a classification of defects into 5 groups: Bloaters, object orientation abusers, change preventer, dispensables, and couplers (see annex ref appendix). This classification has been used as a basis for rule checkers.

4.1.3 Bugs

According to the adapted definition of [30] a bug is an error, failure or defect in a computer program that causes it to violate at least one of its functional or non-functional requirements.

This definition brings out a critical problem: in order to identify bugs in a software, we need to know what the software is supposed to do and also know the non-functional requirements that the software must meet. However, in most cases the intention behind the software is not clearly defined. Moreover, the majority of software is developed in an environment where requirements are unclear and constantly changing. This often leads to considering certain software behaviors as erroneous, no matter what the software is intended to do. For example, there are very few situations where a NULL pointer dereferencing leads to desirable behavior. This complicates the classification of NULL pointer dereferences as bug.

Many techniques exist to find bugs in software, each has advantages and disadvantages. In software testing, software is run with variable input set to determine if the correct output is produced in each case. The main limitation of testing is that it is difficult to build a suite of tests that achieve a high degree of code coverage. In general, even relatively simple programs can have an infinite number of behaviors at run-time, making it impossible to make exhaustive tests. For these reasons, it is easy to use tests to verify that the basic functionality of the software works normally, but it remains difficult to use the tests to discover new bugs.

Code Inspection [31], implemented by some tools such as FindBugs, allows software developers to inspect source code for bugs. An advantage of code inspection on testing is that one can use high-level reasoning on the code to detect erroneous behaviors. Code inspection also has the advantage of facilitating the analysis of difficult to execute portions of source code such as exception handling.

Programs proofs are also used to detect bugs. However, they are not widely used, for several reasons. They are difficult to implement for the unskilled and it is often very complicated to make a complete proof of a program. For example, program proof would require a complete specification of the behavior of the program, and this is extremely difficult to achieve. In addition, as mentioned above, most software project requirements change over time, resulting in an additional complication. There are several other ways to discover bugs and improve the quality of software.

4.1.4 Programming rule

A rule is a description of an action to follow or not when writing code, a description of a particular solution to a problem commonly encountered in software design. The rules are used to ensure the consistency of the source code, and ultimately to ensure the quality of the software.

According to Andre Hora [32], rules can be generic or system-specific. Generic rules are created by non-experts in the system being analyzed. The same rules can be used by separate tools (FindBugs, PMD, SMALLLINT etc.). For example, "check the number of variables and methods" or "detect nested conditional if " . On the other hand, the specific rules are created by experts of the system being analyzed. For example, verification of the use of the correct API, or control of the methods that should be used / defined together, etc., within the framework of the system being analyzed.

In the following the term defect will designates both the defects, the non-respect of the programming rule or bugs.

4.2 Defect Detection Techniques

Several studies on defect detection techniques have been carried out.

According to Ebers et al. [33], the code review consists in having the code re-read in order to detect defaults closest to their introduction phase, throughout the development of the software. They are a form of static analysis that requires a lot of time. They have studied more than 12 000 projects and have shown that collective code review enable the detection of an average 65% of the defects, 50% for the review by a peer, while the tests detect on average only 30%.

Static analysis involves analyzing the source code of a program to extract information from it. This analysis is performed without running the program. In 2003, Ted Kremenek et al. [34] proposed z-ranking, a technique for detecting defects. In 2006, Boogerd et al. [35] proposed a generic method for classification of defects. This method is based on the probability of executing a piece of code in the program. They use a graph to represent the source code. In 2007, Sunghun Kim et al. [36] proposed a prioritization algorithm for violation generated by rule checker based on the source code history. Indeed the violations generated are classified by types and categories. A category includes several types of alerts.

In 2011, Sarah Heckmann et al. [37] have made an inventory and evaluation of defects detections technique. To do this work, they first collected articles in scientific journals (ACM, IEEE, Springer); They sorted these articles manually according to their importance in relation to the problem. In 2011, Haihao Shen et al. [38] proposed a technique to improves Find Bugs.

The authors present a statistical method for finding the defects and compared this approach with FinBugs according to the precision, the recall and the F1-Score.

One of the first definitions of the test was stated in 1979 by Myers [39] as "Testing is executing the program with intent to find abnormalities or defects." According to the IEEE definition "Testing is the execution or evaluation of a system or component by automatic or manual means to verify if it meets the specifications or to identify the differences between expected results and results obtained " It is clear from these definitions that the main purpose of software testing is to examine or run software to find defects.

A formal method is a technique allowing the modeling of all or part of a system on mathematical basis that allow reasoning on properties and to elaborate proofs. This proofs demonstrates that the properties expressed on the program are well respected. There are many definitions of formal methods.

Some definitions are given by Eric Jaeger [40] or Hird [41]. Ludovic [42] presents some advantages of formal methods. The main ones are:

- The precision in the formal specification is greater than that provided by an informal description in natural language which is often ambiguous.
- The basis for enunciating properties and studying the behavior of the system as a whole.

5. Overview Of The Dacord Approach

This section provides an overview of the methodology for extracting modifications related to the correction of defects in a commit. In this analysis we consider the existence of two versions of the same program p and p'. Since the program p' is derived from the program p. All 2 versions of the program are supposed to be syntactically correct and compilable. The key concept of our approach is the decomposition of the source code into atomic modifications. These atomic modifications may have inter-dependencies.

In the proposed approach, defects will be synonymous with rules or tests. The first step is to determine the set of affected rules / tests (corrected defects). Then, in a second step, for each rule / test affected, a set of atomic modifications related to the correction to the affected rules / tests will be extracted. Our method is conservative in the sense that the set of atomic modifications calculated contains at least all the modifications that cause behavioral changes in the rules / tests.

5.1 Background

Before presenting the approach, we will introduce an essential notion : atomic modifications and the inter-dependencies between them.

5.1.1 Atomic modification

Atomic modification is modifications know to be at a low level possible. In literature we find fine-grained modification for example at the level of line of code or coarse grained modification at level of method/class/package in Object Oriented Paradigm for example. These modifications can have syntactic dependencies. In a simple way, an atomic modification MA_1 depends on another atomic modification MA_2 if the application to the original source code (base) version of the MA_1 program without the application in advance of MA_2 results in a syntactically invalid program. In other words, MA_1 depends on MA_2 if MA_2 is a prerequisite for applying MA_1. These inter-dependencies induce a partial order which determines the order in which the MAs are to be applied to the program P to obtain a syntactically correct program P ‖ and P 'if we apply all the MA.

5.1.2 Some terminologies

Change set: A set of atomic modifications made on the source code to move from one version of the source code to the next. This could also be called a patch. In our case, any commit in the version control system will be a *change set* . We focus on commits that fix at least one default. We assume that a *change set* is complete in the sense that it contains all the necessary modifications to fix a violation (even if it can introduce other defects). For example, if a modification adds a parameter to a method definition, other modifications must change the invocations of that method to take the new parameter into account.

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

Sequence valid: it is a sequence (of modifications) that can be applied without error, and that can fix a given defect without introducing others. Because we work specifically with commits that fix a given defect, by building the *change set* there is at least one valid sequence that fixes the defect considered (a valid sequence is at worst the change set itself).

Rule: a description of a fault. It can be a rule of a rule checker, a test that illustrates a bug, or a formal description usable by the proof wizards.

Oracle: A tool or person that parses the source code to detect defects.
Concerning programming rules the oracle is usually a Rule Checker like Find Bugs, PMD, SmallLint Rule checkers consist of a set of rules, each of which is related to a specific type of defect.

We can also work on a formalized version of the program using proof assistants. A proof assistant is software for writing and verifying evidence on assertions related to the execution of computer programs. The proof assistants help the user to develop proof, and verify that the proof developed is correct. In general, proof verification is automated while their construction is partially automated, and generally requires human intervention. As examples of proof assistant we can cite Coq and Isabelle [43]. For bugs, programmers typically use tests. The oracle will be the test environments like JUnit or SUnit.

Violation: A violation of a rule detected by an oracle (or test) that fails. Two violations in two different versions of the code are considered equivalent if they relate to the same rule / test on the same software artefact . In our case, it is enough to identify the software artefact s by their full name (package.class.method.signature) because we are working with coarse grain modifications. In the future we will need to work at the level of AST modifications that will make it more difficult to couple violations together in two different versions. This software artefact identification does not allow us to consider renaming (which will be considered deletion and addition).

We do not want to deal with the renaming at first because the detection of name change is difficult and subject to interpretation, such additional manipulations could pollute our results. Additional filtering will ensure that the renamings are actually ignored in the experiment (we do not consider the deleted artefacts first).

Fixed Violation: A violation found in the base version of the change set, but not in the target version. We distinguish two cases: the software violation artefact is in both versions of the change set (before and after) or it has been removed in the target version. We decided to ignore "fixed" violations because the software artefact that presented it was removed. As we have just seen, for the renaming of entities this will be an adequate behavior. The reasoning behind this decision was that research showed that the majority of programmers did not care about violations [37]. Therefore, a violation may go unnoticed for a long period of time before it simply disappears due to unexpected changes in the source code. We wanted to reduce this risk by ensuring, at least, that the artefact of the software presenting the violation is always present, and thus that the violation has in fact been corrected in one way or another. Some violations require the removal of an artefact of the software on which they occur, for example, for a rule that detects that a method has the same implementation in a superclass and subclass. However, we preferred to ignore them to provide a better understanding of the properties of our algorithm.

5.2 Dacord

DACORD is a technique for detecting the actions needed to correct a defect. In the following, we describe the four stages of the DACORD technique using a common pattern: we present the process of each stage including the inputs and outputs of the step, the description of the process and the follow-up of a discussion.

5.2.1 Step 1: Analysis of commits

In this step we filter commits that correct defects. This activity depends on the type of fault. For example, for rules we use a static code analysis tool like oracle while for bugs one can simply look in the commit logs.

Input: the commits of the software system to be studied. *Output*: commits correcting at least one defect.

Description: commits are sorted according to their creation order. Then they are analyzed two by two to find the violations (defects).

- For the bugs one uses the traditional techniques of detection of the commits correcting bugs [10];
- For rules, the violations obtained for two consecutive commits are compared and classified into three categories:
    - Violations corrected: these are the violations present in the oldest commit but are not present in the recent version of the commit;
    - New violations: these are the violations present in the recent version of the commit but are not present in the oldest version of the commit;
    - Violations in both versions.

Discussions: Rule checkers generate thousands of violations including many violations that are not actually real (false positives) and sometimes violations are not detected by these tools (false negatives). In addition 2 separate rule checkers do not detect exactly the same violations on a single source code.

Algorithm for step 1

```
Begin
    1. load one revision of the system;
    2. run the oracle on it (store the violations);

    3. load the next revision of the system;

    4. run the oracle again (store the new set of violations);
    5. compare the two set of violations ;

    6. if some violations were removed, then this commit fixed corresponding defects in some
       way

    7. further verify that the violation is not removed because the software artefact carrying it
       was deleted
    8. keep all commits couples (the one introducing and the one fixing the defects)
End
```

5.2.2 Step 2: Calculation of actions

In this step, all the atomic modifications of a commit are identified. *Input*: A commit that corrects a default and its predecessor. *Output*: a set of atomic modifications.

Description: Reconstructing changes in a commit not only involves identifying the commit, but first identifying the source code entities that are part of the commit, and then those that have been changed between different commits.

The identification of the source code entities is done either by parsing or by lexical analysis. Lexical analysis is preferred [45], because it is the approach that presents the best performance in time.

To compare two commits, it is necessary to follow three steps.

- find entities defined in the source code of the base commit;
- find the entities defined in the target commit;
- Finally, find the differences between the source code entities of the base commit and the target commit. The comparison can be used to obtain the set of added entities, deleted entities, and the set of modified entities.

To identify the set of modified entities, it is necessary to check the differences between the two versions of the entities of each commit. This approach to find the source code entities changed from commit to commit, assumes that all modifications are stored in the same branch of history (software version). However, this may not be the case. When it is necessary to maintain a stable version of the code, the history of the application is divided: the main story line (the trunk) can have several branches.

Changes to the same files may occur in branches and trunk at the same time. Therefore, to be able to correctly identify changes on a file, it is necessary to analyze the commits of each branch separately, or by analyzing that trunk.

Discussions: To get reliable results we have opted for a method based on syntactic analysis because it gives a very precise idea of the entities and their relationships. It means that it checks the correctness of the source code.

5.2.3 Step 3: Detection

This is DACORD's milestone. Our solution is to sequentially apply selected modifications in the *change set* to identify which one (s) is necessary to fix a violation.

Input: a *change set* and a defect.

Output: the actions required to correct the defect.

Description: We start from the fact that we have a solution to the defect: the commit (*change set*) that corrects the violation and we want to improve it.

For this we opt for the use of local search methods [46]. These methods, also called iterative improvement algorithms, start from an initial solution and aim to improve it. The general principle is the following: starting from an initial solution S_i to the problem, we look for a better solution S_{i+1} in the neighborhood of S_i. If we cannot improve S_i, then we stop, otherwise we replace S_i by S_{i+1}, and We start again.

We use the Hill-Climbing algorithm which is one of simplest local search methods of the literature. It was used for the first time by Lin and Kernighan in 1973 to find a good solution to the problem of the commercial traveler [47]. Since then, local search methods have played a central role in optimization problems. The principle consists of starting from an initial solution, then choosing at each iteration a neighbor who improves the solution. The choice of the neighbor can be done randomly in this case we talk about *first improvement* or by looking for the best neighbor that improves the current solution we speak of *best improvement* [48]. In all cases, the stop criterion is reached when no neighboring solution improves the current solution. In our case, the initial solution S_i will be the whole change set and that is to say a set of atomic modifications and the neighboring solutions will simply be a subset of the initial solution S_i. Our stopping criterion focuses on the number of atomic modifications in the solution. We distinguish two cases. First, if the length of the solution is one atomic modification, it is trivial that we have reached the minimal valid sequence. Second, we set a threshold on the number of iterations that do not decrease the size of the solution. Since we work on operations that require a lot of resources (program compilation), the choice of the threshold will affect the efficiency of the algorithm. Setting this threshold to a very small acceptable value would be a possible improvement because, assuming that the search does not stop the iterations too early (and

therefore does not find the minimum valid sequence), reducing the threshold would reduce resource consumption. This may be interesting, but perhaps not at all significant. In theory, by having sufficient time and testing, our algorithm is able to find the minimal valid sequence that fixes the violation. However, in practice this would be too long for a viable experience. This question is related to how we choose our modifications and how we deal with errors. These two issues will be addressed jointly. In practice, we will have to add conditions to prevent the search from looping endlessly. For example, we could set a threshold on the total number of iterations of the search algorithm.
Discussions: By applying the modifications randomly, three problems must be taken into account:

1. It may not be possible to apply a modification before applying another because of the dependencies discussed above (such as trying to create a method before creating the class to which it belongs);
2. You could apply some unrelated modifications before coming to implement the modification (s) that are needed;
3. What should we do if a modification that fixes the violation introduces another?

Of these three problems, the first is easy to overcome. If it is impossible to apply a randomly chosen modification (since it requires one to be applied before), we have identified three solutions:

1. The simple solution is to simply ignore the current sequence and regenerate a new sequence from scratch. This is the solution we have implemented for the rules.
2. Another solution would be to go back in the construction of the sequence, choosing other modifications.
3. Finally, a third solution would be to calculate the dependencies between the modifications and to prohibit in the random selection to choose a modification that depends on another modification not yet applied. This is the solution we have implemented for bugs.

The second problem related to the selection of undesirable modifications in the sequence before introducing those necessary to fix the violation considered. This is solved using a Hill-Climbing approach.

Note that there are many possible ways to fix a violation, and it is quite possible that in a given change set, several different valid sequences fix a given violation. Consider, for example, a violation indicating that a method does not have enough comments, and a change where two different comments are added to the method. Each of these two amendments could be sufficient to fix the violation. We do not see this as a problem because each of the two modifications is a solution to the violation, even if we were going to miss another possible solution to the violation.

Finally, the third problem to be considered is what to do if a modification fixes the violation under consideration but introduces a new violation. Our solution is not to consider the valid sequence whose application create new violations or resulted in a non-compilable code. If a new violation (or a compilation error) is created by a modification of the sequence being constructed, we will add this new violation to the set of violations we want to fix, so that the sequence will not be considered Valid until the new infringement is also fixed. The only exception is if the new violation also exists in the target version of the program, as this means that applying the entire set will not remove the violation.

```
DACORD

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
LOCAL: chg, the set of modifications from commit still candidate to be part of seq
LOCAL: cpt, counter checking how many sequences of same length we found

chg ← comit
seq ← {}
cpt ← 1
while ( cpt<threshold value)
while viol is not fixed
  mod← a random modification from chg
  apply mod on the current source code
  remove mod from chg
  add mod to seq
end while
if length(seq) < length(chg)
cpt ← 1
chg ← seq
else

cpt++
end if
end while
return seq
```

Algorithm 1: Algorithm to find the minimal valid sequence fixing a given violation from a commit

5.2.4 Step 4: Validation

Validation is defined by ISO 9001 as the process of ensuring that the expected results are achieved. This standard also provides prerequisites for validation. The most important validation prerequisites for our case are knowledge of critical points and a trained and competent staff :

- **Knowledge of critical points**. Before validating an approach, one must know its critical points. In our case it is knowledge of the rules / tests, the causes of their violations and their corrections;
- **Trained and competent personnel**. The experts selected for validation should master the system because some systems have specific rules.

The sequence of actions obtained in step 3 is manually validated by a qualitative validation method to verify whether it actually corrects the defect.

Input: a defect and a set of atomic modifications fixing the defect. *Output*: Accept or reject the solution.

Description: The initial step of the work is to describe the methods and procedures. We write precisely what the experts must do. At the end, one should not have any doubt, error, or imprecision in the text. The points to be validated are clearly explained. Then, we select several experts from the system being analyzed. They are presented with the defect, the textual description of the defect, the solution detected in the previous step. Each expert will indicate whether the proposed solution actually corrects the defect and whether there are unnecessary modifications in the solution.

Discussions: Qualitative methods sometimes have a degree of subjectivity in the process of approving the result. To mitigate this, at least three experts will be used to validate our results.

6. 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 and bugs because they were our initial target and it is easier to find an oracle that will check the presence or not (correction) of a violation of these types. Therefore, we also require a system with some rule checker and test runner tool available as well as rules. We will validate our solution on Moose and Pharo [49] two open-source, real-world, and non-trivial systems, with a consolidated number of developers and users. Pharo is an open-source dialect of Smalltalk and implementation of its programming environment. Currently, Pharo has approximately 60 collaborators throughout the world; it is used by more than 15 universities to teach programming and by 10 research groups. Moose is written in Pharo, a Smalltalk inspired language, we selected SmallLint [50], the most adopted Smalltalk code analysis tool. 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

6.1 Research Questions

We validate our approach in three steps: First, we will evaluate whether our approach can actually discover the sequence of source code changes required to fix a given rule violation (RQ1) second, we will evaluate the relation between unnecessary modifications and violations (RQ2). Third, we will evaluate if our approach help developer to improve their quality and source code quality (RQ3).

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

1. A commit may fix several violations?
2. Focusing on small commits would discard too much information?
3. Our aproach can find the correct Minimal Valid Sequence to fix a given violation?

For the second question, we will consider the following points :

1. Do unnecessary modifications are related to defects correction ?
2. Do unnecessary modifications (related to defect) increase code quality ?

For the third question we will simply verify that our approach helps developers to easily fix defaults.

6.2 Results of the Experiment

In total we analyze more than 30 000 commits. 65(couple) met our conditions for programming rule [51] and 37(couple) for bugs [52]. These 102 commit fix in total 437 violations. Only 34 commits out of 102 fix exactly one defect.

RQ1 evaluation of the approach

RQ1a A commit may fix several violations?

a) Programming rules

We experimented with 65 commits containing fixes for 368 violations of good programming practices expressed in rules and detected by static analysis checkers. The maximum violations fixed per commit is 75, and the median 2 (see Table 1). 21 of the 65 commits (32%) fixed only one violation.

Table 1: 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

b) bugs

We experimented with 37 commits containing fixes for 104 bugs extracted from the repository (SmalltalkHub.com). For all these commits, the corresponding commit message references at least one issue report that had been marked as resolved. We considered the commit message and the actual applied modifications when performing manual inspection. If the commit message indicated that the applied modifications solve more than one issue report, we classified the commit as tangled. In the majority of cases, the commit message clearly indicated tangled commit. If we had no doubt that the commit targeted more than one issue or contained also additional enhancement (e.g. refactoring), we classified the commit as tangled. Similarly, if we had no doubt that the commit is atomic (not tangled), we classified it as atomic.

Table 2 presents the details of our analysis. The table shows that the maximum bugs fixed per commit is 5, and the minimum 1. 32 of the 37 bug fix commit fix more than one bug and thus automatically classified as tangled.

Table 2: Distribution of bugs in commit

	#	min	max
Total commit	37		
Bug fix commits fixing more than one bug	32		
Bug fix Commit fixing only one bug	5		
Bug fixed / commit		1	5

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 [53]).

RQ1b Focusing on small commits would discard too much information?

a) Programming rules

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 [54].
b) bugs

32 of the 37 bug fix commit fix more than one bug and thus automatically classified as not small. In addition, 3 of the 5 commits fixing only one bug were also tangled (see table 2 and 3). The result is that, 35 bug fix commits out of 37 studied were not small (i.e. 95%).

Table 3: Statistics of untangled bug fix commit

	#
bug fix Commit	37
Tangled bug fix Commit	35
Correctly untangled commits	32

From the results for RQ1, we deduce that large commit introduces noise and skews into any analysis of the history to find fix actions. Thus, it seems to confirm that this fraction is large enough to threaten the detection of a set of source code modifications that frequently appear together for fixing a given defect and there is a need for an approach to isolate the modifications fixing a given defect in a large or tangled defect fixing commit.

RQ1c our aproach can find the correct Minimal Valid Sequence to fix a given violation?

a) Programming rules

For programming rules we set to 20 the threshold on the number of identical sequences before deciding it will not be able to improve it (and therefore it is considered to be minimal).

For our main validation, we manually evaluated 87 fixed rule violations chosen randomly. 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. But, setting this *threshold* to a higher value would spend more time finding the exact minimal valid sequence. 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.

b) bugs

For RQ1.c we evaluate the algorithm with a set of bug fix commits which was extracted from the repository. The last line of table 3 shows the number of tangled bug fix commits in which we found correctly modifications related to bug fixes. To evaluate the accuracy of our algorithm, we manually evaluated the results. The results are that, we could automatically extract bug fix actions in 32 commits out of 35 tangled commits. For the three remaining commits, the problem is due to the tests incorrectly linked to bug fixes. We conclude that our algorithm is working, although there is a need to increase precision when coupling bug with tests cases.

RQ2 evaluate the relation between unnecessary modifications and violations
RQ2.a Do unnecessary modifications are related to defects correction?

To exhibit the influence of unnecessary modifications on defect correction; we focus on commits fixing exactly one defect. We use for this research question only 34 commits which fix exactly one defect. We present to 4 expert modification identified by our approach as necessary to fix a defect and the ones left in the same commit. We ask them if some unnecessary modifications can be part of correction of the given defect. In 5 case out of 34 (14%) of case programming rule they found relations.

RQ2.b Do unnecessary modifications (related to defect) increase code quality ?

We investigate if these unnecessary modifications help to increase code quality (comprehensibility, maintenability, coherence). the experts respond to a binary question. they answer that all presented modifications help to increase the comprehensibility of the source code.

RQ3 learning from the past

For the third research question we selected a set of 10 violations that we grouped into two groups of respectively 6 and 4 violations according to the rules. We have named the first group A and the second B. For each group of violations we have three examples of violation correction of the same rule detected by our approach. We have presented to two groups of two developers working in pair programming named Group I and Group II. In group I, in addition to the 10 violations to be corrected, the examples of corrections of violations of the same type as the violations of group A were presented while in group II we presented those similar to the violations of group II. We have timed the time taken by each group and the results are presented in table 4.

Table 4: Time spent to correct violations

— Violations Developpers —	Group I	Group II
Group A	127 minutes	185 minutes
Group B	213 minutes	152 minutes

We find that each group spent less time to fix violations of the group for which they has similar examples of corrections of violations detected by our approach. We conclude that using our approach allowed developers to improve their qualities to correct defects by learning from past knowledge and thereby improve the quality of the software.

7. THREATS TO VALIDITY

As any empirical evaluation, the results of our experiments are subject to threats to validity. We identified the following noteworthy threats:

1. The studied system 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.

2. The way in which we chose bug fix commit may introduce bias because we chose a project who use a repository, a bug tracking system and tests. We also believe Pharo is a credible, real world, non-trivial, case study. It is a medium to big system and it includes a significant number of versions. However, we believe firmly that our approach is language independent.
3. We disregarded violations that were fixed because the software artefact 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 artefact 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.
4. Internal threats to validity are related to the implementation of our approach. It is still possible that our implementation of the approach contains errors which can affect the exactitude of our results. To counter this threat, we manually studied a subset of the results and we did not find any obvious errors. Bias with respect to developer working habits might also occur in our selection of evaluation subjects. To reduce this risk, we selected diverse developers, all of them working in different projects and even in different physical locations. Thus, we believe the developers represent a heterogeneous enough population of Pharo developers.

8. CONCLUSION

In this paper, we have presented the idea that one can mine defect fix actions from software repositories. In other words, one can extract from past, a sequence of modifications needed to fix an alert (adding a method call, deleting method). We presented a methodology that help developer to learn to fix defect from the past of the system, and we applied it on a real system. An extension of the work would be to deal with finer grained modifications, typically at the level of the AST. The idea would be to extract an abstract summary of all the violation fixes for the given defect.

REFERENCES

[1] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, ―Understanding and predicting the process of software maintenance release,in Proceedings of the 18th international conference on Software engineering. IEEE Computer Society, 1996, pp. 464–474.

[2] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, ―Apievolutionminer: Keeping api evolution under control,in Proceedings of the Software Evolution Week (CSMR-WCRE'14), 2014.

[3] 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.

[4] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, —Does code decay? assessing the evidence from change management data,Software Engineering, IEEE Transactions on, vol. 27, no. 1, pp. 1–12, 2001.

[5] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, —Predicting fault incidence using software change history,Software Engineering, IEEE Transactions on, vol. 26, no. 7, pp. 653–661, 2000.

[6] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, —Mining version histories to guide software changes,Software Engineering, IEEE Transactions on, vol. 31, no. 6, pp. 429–445, 2005.

[7] H. Kagdi and J. Maletic, —Software repositories: A source for traceability links,in 4th ACM International Workshop on Traceability in Emerging, 2007.

[8] A. Y. Yao, —Cvssearch: Searching through source code using cvs comments,in Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01). IEEE Computer Society, 2001, p. 364.

[9] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, —Analyzing the co-evolution of comments and source code, ‖ Software Quality Control, vol. 17, no. 4, pp. 367–394, Dec. 2009.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, —When do changes induce fixes?‖ in Proceedings of the 2005 International Workshop on Mining Software Repositories, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

[11] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr, —Automatic identification of bug-introducing changes,in Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on. IEEE, 2006, pp. 81–90.

[12] V. S. Sinha, S. Sinha, and S. Rao, —Buginnings: Identifying the origins of a bug,in Proceedings of the 3rd India Software Engineering Conference, ser. ISEC '10. New York, NY, USA: ACM, 2010, pp. 3–12.

[13] A. Mockus and L. G. Votta, —Identifying reasons for software changes using historic databases, ‖ in Software Maintenance, 2000. Proceedings. International Conference on. IEEE, 2000, pp. 120–130.

[14] M. Fischer, M. Pinzger, and H. Gall, —Populating a release history database from version control and bug tracking systems,in Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on., 2003, pp. 23–32.

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, —Fair and balanced?: bias in bug-fix datasets,in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2009, pp. 121–130.

[16] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, —The missing links: bugs and bug-fix commits,in Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010, pp. 97–106.

[17] P. Bourque, R. Dupuis, A. Abran, J. W. Moore, and L. Tripp, —The guide to the software engineering body of knowledge,IEEE software, vol. 16, no. 6, p. 35, 1999.

[18] J. A. McCall, P. K. Richards, and G. F. Walters, —Factors in software quality. volume i. concepts and definitions of software quality,DTIC Document, Tech. Rep., 1977.

[19] ISO/IEC, ―Tr 9126-2:2003 software engineering – product quality.

[20] K. Mordal, ―Analyse et conception d un modele de qualité logiciel,Ph.D. dissertation, Université de Paris 8, 2012.

[21]M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, ―Learning natural coding conventions,in Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293.

[22] P. W. Oman and C. R. Cook, ―Typographic style is more than cosmetic,Commun. ACM, vol. 33, no. 5, pp. 506–520, May 1990.

[23] M. Broy, F. Deißenböck, and M. Pizka, ―A holistic approach to software quality at work,in Proc. 3rd World Congress for Software Quality (3WCSQ), 2005.

[24] S. Haiduc and A. Marcus, ―On the use of domain terms in source code,in Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, June 2008, pp. 113–122.

[25] B. Webster, Pitfalls of Object-Oriented Development. M T Books(MIS PRess Inc.), 1995.

[26] R. Arthur, Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[27] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[28] F. Martin, Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[29] M. Mäntylä, J. Vanhanen, and C. Lassenius, ―A taxonomy and an initial empirical study of bad smells in code,in Proceedings of the International Conference on Software Maintenance, ser. ICSM ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 381–.

[30] V. Dallmeier, ―Mining and checking object behavior,Ph.D. dissertation, Universitat des Saarlandes, Aug. 2010.

[31] A. Kesäniemi and N. Oy, ―Automatic vs. manual code analysis,2009.

[32] A. C. Hora, ―Assessing and improving rules to support software evolution,Ph.D. dissertation, Université Lille 1-Sciences et Technologies, 2014.

[33] C. Ebert and C. Jones, ―Embedded software: Facts, figures, and future,Computer, vol. 42, no. 4, pp. 42–52, Apr. 2009.

[34] T. Kremenek and D. Engler, ―Z-ranking: Using statistical analysis to counter the impact of static analysis approximations,in Proceedings of the 10th International Conference on Static Analysis, ser. SAS’03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 295–315.

[35] C. Boogerd and L. Moonen, ―Prioritizing software inspection results using static profiling,in Source Code Analysis and Manipulation, 2006. SCAM ’06. Sixth IEEE International Workshop on, Sept 2006, pp. 149–160.

[36] S. Kim and M. D. Ernst, ―Which warnings should i fix first?‖ in Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on

The Foundations of Software Engineering, ser. ESEC-FSE 'New York, NY, USA: ACM, 2007, pp. 45–54.

[37] S. Heckman and L. Williams, —A systematic literature review of actionable alert identification techniques for automated static code analysis,Inf. Softw. Technol., vol. 53, no. 4, pp. 363–387, Apr. 2011.

[38] H. Shen, J. Fang, and J. Zhao, —Efindbugs: Effective error ranking for findbugs,in Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 299–308.

[39] G. J. Myers, C. Sandler, and T. Badgett, The art of software testing. John Wiley & Sons, 2011.

[40] E. Jaeger, —Remarques relatives a l emploi des méthodes formelles (déductives) en sécurité des systémes d information,Secrétariat général de la défense nationale 51 boulevard de la Tour Maubourg 75700 Paris SP 07, Tech. Rep., 2007.

[41] G. R. Hird, —Formal methods in software engineering,in Digital Avionics Systems Conference, 1990. Proceedings., IEEE/AIAA/NASA 9th, Oct 1990, pp. 230–234.

[42] L. Casset, —Construction correcte de logiciels pour carte à puce,Ph.D. dissertation, Citeseer, 2002.

[43] F. Wiedijk, —The fifteen provers of the world,2002.

[45] T. Zimmermann and P. Weißgerber, —Preprocessing cvs data for fine-grained analysis,2004.

[46] J. K. Lenstra, Local search in combinatorial optimization. Princeton University Press, 2003.

[47] B. Selman and C. P. Gomes, —Hill-climbing search‖ Encyclopedia of Cognitive Science.

[48] D. Papadias, —Hill climbing algorithms for content-based retrieval of similar configurations, in Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2000, pp. 240–247.

[49] M. U. Bhatti, N. Anquetil, and S. Ducasse, —An environment for dedicated software analysis tools, ERCIM News, vol. 88, pp. 12–13, Jan. 2012.

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

[51] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, and K. Taiwe, —Identifying the exact fixing actions of static rule violation,in Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, March 2015, pp. 371–379.

[52] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, and K. Taiwe, —Identifying the exact bug fixing actions,in Empirical Software Engineering in Practice (IWESEP), 2016 7th International Workshop on, March 2016, pp. 51–56.

[53] A. Lozano, G. Arévalo, and K. Mens, —Co-occurring code critics,in SATToSE 2014 Pre-proceedings, V. Zaytsev, Ed., July 2014, pp. 10–13,

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

AUTHORS

Hayatou Oumarou, Ph.D. currently works as Assistant Lecturer with the Department of Computer Science, Higher Teachers' Training College, The University of Maroua, Cameroon. His research interests include Software Engineering, Software Testing, Quality Engineering, and Reuse. Many research stays has brought him to INRIA Lille, Nord Europe.

Prof. Dr.-Ing. habil. Kolyang, is Professor and Head of Department of Computer Science, Higher Teachers' Training College, The University of Maroua, Cameroon. His research interests include Software Engineering, Computer Networks, Cryptographie, and Formal methods. He has published more than 40 papers in these areas. He is associated with various universities i.e. University of Bremen, University of Ngaoundere. He is a reasearch ambasador for the University of Bremen.