Mining Repair Actions for Guiding Automated Program Fixing

Matias Martinez, Martin Monperrus

Mining Repair Actions for Guiding Automated Program Fixing

Matias Martinez
University of Lille & INRIA, France

Martin Monperrus
University of Lille & INRIA, France

Technical report, Inria, 2012

***Abstract*—Automated program fixing consists of generating source code in order to fix bugs in an automated manner. Our intuition is that automated program fixing can imitate human-based program fixing. Hence, we present a method to mine repair actions from software repositories. A repair action is a small semantic modification on code such as adding a method call. We then decorate repair actions with a probability distribution also learnt from software repositories. Our probabilistic repair models enable us to mathematically reason on the automated software repair process. By applying our method on 14 repositories of Java software and 89993 versioning transactions, we show that our probabilistic repair actions are able to guide the automated fixing process in the repair space, with a probabilistic focus on likely repair shapes first.**

I. Introduction

Automated program fixing consists of generating source code in order to fix bugs in an automated manner [1], [2], [3], [4], [5]. The generated fix is often an incremental modification (a "patch" or "diff") over the software version exhibiting the bug. The previous contributions in this new research field make different assumptions on what is required as input (e.g. good test suites [2], pre- and post- conditions [3], policy models [1]). The repair strategies also vary significantly. Examples of radically different models include genetic algorithms [2] and satisfiability models (SAT) [6].

However behind the diversity of approaches, there is a common denominator: automated program fixing deals with *repair actions* on software. A software repair action is a kind of modification on source code that is made to fix bugs. We can cite as examples: changing the initialization of a variable; adding a condition in an "if" statement; adding a method call, etc. In this paper, we use the term "repair model" to refer to a set of repair actions. For instance, the repair model of Weimer et al. [2] has three repair actions: deleting a statement, inserting a statement taken from another part of the software, swapping two statements

There is a key difference between a repair action and a repair: a repair action is a kind of repair, a repair is a concrete patch. In object-oriented terminology, a repair is an instance of a repair action. For instance, "adding a method call" is a repair action, "adding `x.foo()`" is a repair. Note that there is a clear difference between a repair action and a repair: a repair is not program-specific, it contains no domain-specific data such as variable names or literal values. A search-based automated program repair process often balances between exploring a new repair action or a new combination of repair actions and instantiating them. Again, in the repair model of Weimer et al. [2], at each step, the repair action is first chosen randomly (add, delete or swap), and then instantiated. We call those two phases the repair shaping and the repair synthesis. A repair shape is thus defined as a subset of repair actions of a repair model. In this paper, we try to understand the fundamentals of repair shaping.

First we present an approach to mine repair actions from patches written by developers. In other words, we mine repair models that imitate human-based program fixing. We find traces of human-based program fixing in software repositories (e.g. CVS, SVN or Git), where there are versioning transactions that only fix bugs. We use those "fix transactions" to mine semantic repair actions such as adding a method call, changing the condition of a "if", deleting a catch block. Semantic repair actions are extracted with the semantic differencing algorithm of Fluri et al. [7]. This results in a repair models that are much bigger (41 and 173 repair actions) compared to related work which considers at most a handful of repair actions.

Second, we propose to decorate the repair models with a probability distribution. Our intuition is that not all repair actions are equal and certain repair actions are more likely to fix bugs than others. We also take an empirical viewpoint to define those probability distributions: we learn them from software repositories. We show that those probability distributions are independent of the application domain.

Then, we demonstrate that our probabilistic repair models enable us to reason on the automated software repair process. We introduce a mathematical formula that predicts the logical time to find a given repair shape. We describe a novel algorithm, called MCShaper, that enables the realistic comparison of repair models and their probability distributions. The algorithm runs over thousands of bug fix transactions of software history to determine how fast it takes to shape a repair under a certain distribution.

We apply the whole approach on $89,993$ versioning transactions of the versioning repositories of 14 open-source Java projects. Our empirical results indicate that the probability distribution of repair actions over real bug fix data is indeed unbalanced. Our algorithm indicates that despite the huge theoretical size of all possible repair shapes, the extremely distorted probability distribution results in being able to find most repair shapes of real bug fixes in less than 1000 attempts. In other terms, our probabilistic repair actions are able to

guide the automated fixing process in the repair space, with a probabilistic focus on likely repair shape first.

Compared to previous work on automated repair, this paper takes a risk. To our knowledge, this is the first paper on automated program repair where no real bugs are fixed. To some extent, other papers in this field are "vertical" papers: they go from an idea until generating a couple of correct patches for a couple of examples. They make many assumptions on the way, and do not explore all theoretical and empirical reasons of their success. This is perfect for exploratory research. Our paper is of a fundamentally different kind, we take an horizontal point of view by focusing on a single step in the repair process: repair actions and repair shaping only. This focus enables us to 1) to introduce mathematical reasoning in the field of automated software repair and 2) to discuss the empirical roots of software repair in depth (which actions are most common in human-based fixes, to which extent repair actions are project independent, etc.).

II. Describing Versioning Transactions with a Change Model

In this section, we describe the contents of versioning transactions of 14 repositories of Java software. Previous empirical studies on versioning transactions [8], [9], [10], [11], [12] focus on metadata (e.f. authorship, commit text) or size metrics (number of changed files, number of hunks, etc.). On the contrary, we aim at describing versioning transactions in terms of contents: what kind of source code change do they contain: addition of method calls; modification of conditional statements; etc. To our knowledge, few empirical studies (e.g. [13], [14]) discuss this point, and only at a coarser grain compared to what we describe in this paper.

Note that other terms exist for referring to versioning transactions: "commits", "changesets", "revisions". Those terms reflect the competition between versioning tools (e.g. Git uses "changeset" while SVN "revision") and the difference between technical documentation and academic publications which often use "transaction". In this paper, we equate those terms and generally use the term "transaction", as most of previous work does.

Software versioning repositories (managed by version control systems such as CVS, SVN or Git) store the source code changes made by developers during the software lifecycle. Version control systems (VCS) enables developers to query versioning transactions based on revision number, authorship, etc. For a given transaction, VCS can produce a difference ("diff") view that is a line-based difference view of source code. For instance, let us consider the following diff:

```
1  while(i < MAX_VALUE){
2    op.createPanel(i);
3  - i=i+1;
4  + i=i+2;
5   }
```

The difference only shows one string replaced by another one. However, one could also observe the changes at a semantic level, rather than at the syntactic level. In this case, the semantic diff is a change of a literal inside an assignment. In this section, our research question is: *what are versioning transactions made of at the semantic level?*.

To answer this question, we have followed the following methodology. First, we have chosen a semantic difference algorithm from the literature. Then, we have constituted a dataset of software repositories to run the semantic difference algorithm on a large number of transactions. Finally, we have computed descriptive statistics on those semantic differences. Let us first discuss the dataset.

A. Dataset

CVS-Vintage is a dataset of 14 repositories of open-source Java software [15]. The inclusion criterion of CVS-Vintage is that the repository mostly contains Java code and has been used in previous published academic work on mining software repositories and software evolution. This dataset covers different domains: desktop applications, server applications, cross-cutting libraries such as logging, compilation, etc. It includes the repositories of the following projects: Argouml, Columba, Jboss, Jhotdraw, Log4j, org.eclipse.ui.workbench, Struts, Carol, Dnsjava, Jedit, Junit, org.eclipse.jdt.core, Scarab and Tomcat. In all, the dataset contains 89993 versioning transactions, 62179 of them have at least one modified Java file. Overtime, 259264 Java files have been revised (which makes a mean number of 4.2 Java files modified per transaction).

B. Semantic Differencing

There are different propositions of semantic differencing algorithms in the literature. Important ones include Raghavan et al.'s Dex [16], Neamtiu et al's AST matcher [17], Dallmeier and Zimmermann's iBugs tool [18] and Fluri et al's ChangeDistiller [7]. For our empirical study on the contents of versioning transactions, we have selected the latter.

ChangeDistiller [7] is a fine-grained semantic differencing tool, it provides detailed information on the difference between statements. It expresses fine grain source code changes using a taxonomy of 41 source changes types, such as "statement insertion" of "if conditional change". ChangeDistiller handles changes that are specific to object-oriented elements such as "field addition". Fluri and colleagues have published an open-source stable and reusable implementation of their algorithm for analyzing changes of Java code.

Formally, for each versioning transaction of Java files, ChangeDistiller produces a set of "semantic source code changes". Since ChangeDistiller is a semantic differencer, formatting transactions (such as changing the indentation) produce no semantic change at all. Each semantic source code change is a 3-value tuple: $scc = (ct, et, pt)$ where *ct* is one of the 41 change types, *et* (for entity type) is a finer grain, it refers to the source code entity related to the change (for instance, a statement update may change a method call or an assignment), and *pt* (for parent type) indicates the parent code entity where the change take place (such as a the top-level method body or inside an if block). For the short listing above, ChangeDistiller outputs one single semantic change that is a statement update

Change Action	#Changes α_i	Prob. χ_i
Statement insert	345548	28,9
Statement delete	276643	23,1
Statement update	177063	14,8
Statement parent change	69425	5,8
Statement ordering change	56953	4,8
Additional functionality	49192	4,1
Condition expression change	42702	3,6
Additional object state	29328	2,5
Removed functionality	26172	2,2
Alternative part insert	20227	1,7
Total	**1196385**	

TABLE I
THE TOP-10 SEMANTIC CHANGES OF CHANGE MODEL CT REPRESENTED AMONG 62179 VERSIONING TRANSACTIONS.

Change Action	α_i	Prob. χ_i
Statement insert of Method invocation	83046	6,9
Statement insert of If statement	79166	6,6
Statement update of Method invocation	76023	6,4
Statement delete of Method invocation	65357	5,5
Statement delete of If statement	59336	5
Statement insert of Variable declaration statement	54951	4,6
Statement insert of Assignment	49222	4,1
Additional functionality of Method	49192	4,1
Statement delete of Variable declaration statement	44519	3,7
Statement update of Variable declaration statement	41838	3,5
Total	**1196385**	

TABLE II
THE TOP-10 SEMANTIC CHANGES OF CHANGE MODEL CTET REPRESENTED AMONG 62179 VERSIONING TRANSACTIONS.

(ct) of an assignment (et) and the parent code entity is a while body (pt).

C. Change Models

All versioning transactions can be expressed within a "change model". We define a change model as a set of "change actions". For instance, the change model of standard Unix diff is composed of two change actions: line addition and line deletion. A change models represents a kind of feature space, and observations in that space can be valued. For instance, a standard Unix diff produces two integer values: the number of added lines and the number of deleted lines. ChangeDistiller enables us to define the following change models.

CT is composed of 41 features, the 41 change types of ChangeDistiller. For instance, one of this feature is "Statement Insertion" (we may use the shortened name "Stmt_Insert"). **CTET** is made of all valid combinations of the Cartesian product between change types and entity types. There are 104 entity types but many combinations are impossible by construction, as a result CTET contains 173 features. For instance, since there is one entity type representing assignments, one feature of CTET is "statement insertion of an assignment".

In the rest of this paper, we express versioning transactions within those two change models. There is no better change model per se: they describe versioning transactions at different grains. We will see later that depending on the perspective, both change models have pros and cons.

D. Measures for Change Actions

We define two measures for a change action i: α_i is the absolute number of change action i in a dataset; χ_i is the probability of observing a change action i as given by its frequency over real data. For instance, let us consider feature space CT and the change action "statement insertion" (StmtIns). If there is $\alpha_{StmtIns} = 12$ source code changes related to statement insertion among 100, the probability of observing a statement insertion is $\chi_{StmtIns} = 12\%$

E. Empirical Results

We have run ChangeDistiller over the $62,179$ Java transactions of our dataset, resulting in 1196385 semantic changes. Table I and II present the top 10 change actions and the associated measures for change models CT and CTET. The comprehensive tables for all change actions are given in companion technical report [19]. In Table I, one can see that inserting statements is the most common change type (41.3% of transactions contain at least one inserted statement), which makes sense for open-source software that is generally in constant growth. Statement update and statement deletion are #2 and #3 change types, followed by API documentation update. The distribution is rather unbalanced: those top-ten change actions (out of 41) account for 91.5% of the distribution.

Let us now compare the results over change models CT and CTET. One can see that statement insertion is mostly composing of inserting a method invocation (6.9%), insert an "if" conditionals (6.6%), and insert a new variable (4.6%). Since change model CTET is at a finer grain, there are less observations: both α_i and χ_i are lower. The probability distribution (χ_i) over the change model is less sharp (smaller values) since the feature space is bigger. High value of χ_i means that we have a change action that can frequently be found in real data: those change actions have of a high "coverage" of data. CTET features describe modifications of software at a finer grain. The differences between those two tables illustrate the tension between a high coverage and the analysis grain.

F. Project-independence of Change Models

An important question is whether the probability distribution (composed of all χ_i) of Table I and II is generalizable to Java software or not. That is, do developers evolve software in a similar manner over different projects? To answer this question, we have computed the metric values not for the whole dataset, but per project. In other terms, we have computed the frequency of change actions in 14 software repositories. We would like to see that the values do not vary between projects, which would mean that the probability distributions over change actions are project-independent. Since our dataset covers many different domains, having high correlation values would be a strong point towards generalization.

We computed the Spearman correlation values between the probability distributions of all pairs of project of our datasets (i.e. $\frac{14*13}{2} = 91$ combinations). For instance, the Spearman correlation between columba and dnsjava is 0.98. All values are given in the companion technical report [19]. They are high, the majority being higher than 0.9. *This shows that the likelihood of observing a change action is globally indepen-*

dent of the project used for computing it[1]. We also computed the correlation between projects within change model CTET (see companion technical report [19]). They lower than those of CT, but still high enough to be consider that the distribution over CTET are project-independent as well.

G. Link with Automated Program Repair

Let us now consider those results under the perspective of automated software repair: *our intuition is that automated software repair should concentrate more on fixes made of likely semantic changes rather than of those made of rare change actions*. This means that we think that a repair algorithm whether based on genetic algorithm, probabilistic call graphs or test case generation should explore more often frequent change actions (such as inserting an if statement in change model CTET).

III. From Change Actions to Repair Actions

This section presents how we can transform a "change model" into a "repair model" usable for automated software repair. As discussed in Section II, a change model describes all types of source code change that occur during software evolution. On the contrary, we define a "repair action" as a change action that often occurs for repairing software, i.e. often used for fixing bugs.

By construction, a repair model is equal to a subset of a change model in terms of features. But more than the number of features, our intuition is that the probability distribution over the feature space would vary between change models and repair models. For instance, on might expect that changing the initialization of a variable has a higher probability in a repair model. Hence, the difference between a change model and a repair model is matter of perspective. Since we are interested in automated program repair, we now concentrate on the "repair" perspective hence use the terms "repair model" and "repair action" in the rest of the paper.

A. Versioning Transaction Bags

In Section II, we have defined and discussed two measures per change action i: α_i and χ_i. For instance, $\chi_i StmtInsert$ gives the frequency of a statement insertion. Those measures implicitly depend on a transaction bag to be computed, so far we have considered all versioning transactions of the repository containing at least one modified Java file. We think that for defining a repair space, we need to apply those two measures on a transaction bag representative of software repair. What is such a transaction bag?

1) Based on Commit Texts: When committing source code changes, developers may write a comment/message explaining the changes they have made. For instance when a transaction is related to a bug fix, they may write a comment referencing the bug report or describing the fix.

To identify transaction bags related to bug fix, previous work focused on the content of the commit text: whether it contains a bug identifier, or whether it contains some keywords such as "fix" (see [20] for a discussion on those approaches). To identify bug fix patterns, Pan et al. [21] select transactions containing at least one occurrence of "bug", "fix" or "patch". We call such a transaction bag BFP. We will compute α_i and χ_i based on this definition.

Our intuition is that such a transaction bag makes a strong assumption on the development process and the developer's behavior: it assumes that developers generally put syntactic features in commit texts enabling to recognize repair transactions, which is not really true in practice [20], [22].

2) Based on Syntactic Features: A well-known kind of fix consists of adding, deleting or changing a single line of code in software. More generally, transaction bags can be defined as containing at most N lines changed (N being the sum of added, deleted and modified lines). Our intuition is small transactions are very likely to only contain a bug fix and unlikely to contain a new feature (along the same line as e.g. [13]). We call such transaction bags N-LC (for instance 1-LC consists of all transactions of at most one line changed).

3) Based on Semantic Features: Finally, we may define fixing transaction bags based on their semantic contents, i.e. based on the type and numbers of change actions that a versioning transaction contains. In particular, repair actions may be those that appear atomically in transactions (i.e. the transaction only contains one semantic source code change). This has the same inspiration as transaction bags built on one-line fixes. However, a one line-fix may actually contain several semantic changes (such as changing the static type declaration of a variable and changing its initialization value). This transaction bag is called N-SC (for N Semantic Changes, e.g. 1-LC represents the bag of transactions containing a single semantic source code change).

The main question we ask is whether those different definitions of "repair transactions" yield different topologies for repair models.

B. Methodology

We have applied the same methodology as in II. We have computed the probability distributions of repair model CT and CTET based on different definitions of fix transactions, i.e. we have computed α_i and χ_i based on different transactions bags. In this paper, we present six of them, that nicely span the scope of different results we have experienced: ALL transactions, 1-LC (one-line fixes), 1-SC (one single semantic change), BFP (Pan et al. [21] baseline: "bug", "fix" or "patch" in the commit text), 20-LC and 20-SC.

C. Empirical Results

Table III presents the top 10 change types of repair model CT associated with their probability χ_i for different versioning transaction bags. The first column is the row χ of table I, it is reproduced here for easy comparison. The complete table for all repair actions is given in the companion technical report [19]. *Overall, the distribution of repair actions over real bug fix data is very unbalanced, the probability of observing a*

[1] if the project history is large enough, it is out of scope of this paper to precisely define this "large enough"

ALL	1-LC	1-SC	BFP	20-SC	20-LC
Stmt_Insert-29%	Stmt_Upd-33%	Stmt_Upd-38%	Stmt_Insert-32%	Stmt_Insert-33%	Stmt_Insert-34%
Stmt_Del-23%	Stmt_Insert-24%	Add_Funct-14%	Stmt_Del-23%	Stmt_Del-16%	Stmt_Del-18%
Stmt_Upd-15%	Stmt_Del-14%	Cond_Change-13%	Stmt_Upd-12%	Stmt_Upd-16%	Stmt_Upd-15%
Param_Change-6%	Cond_Change-13%	Stmt_Insert-12%	Param_Change-7%	Param_Change-7%	Param_Change-10%
Order_Change-5%	Param_Change-6%	Stmt_Del-6%	Order_Change-6%	Add_Funct-7%	Cond_Change-6%
Add_Funct-4%	Order_Change-2%	Rem_Funct-5%	Add_Funct-4%	Cond_Change-5%	Order_Change-5%
Cond_Change-4%	Inc_Access_Change-1%	Add_Obj_St-3%	Cond_Change-3%	Add_Obj_St-3%	Alt_Part_Insert-2%
Add_Obj_St-2%	Rem_Obj_St-1%	Order_Change-2%	Add_Obj_St-2%	Order_Change-3%	Add_Funct-2%
Rem_Funct-2%	Add_Obj_St-1%	Rem_Obj_St-2%	Alt_Part_Insert-2%	Rem_Funct-2%	Add_Obj_St-2%
Alt_Part_Insert-2%	Decr_Access_Change-1%	Inc_Access_Change-1%	Rem_Funct-2%	Alt_Part_Insert-2%	Alt_Part_Del-1%

TABLE III
TOP 10 CT CHANGE TYPES AND THEIR PROBABILITY χ_i FOR DIFFERENT TRANSACTION BAGS. THE DIFFERENT HEURISTICS USED TO COMPUTE THE FIX TRANSACTIONS BAGS HAS A SIGNIFICANT IMPACT ON THE RANKING AND THE PROBABILITIES.

single repair actions goes from more than 30% to 0.000x%. We observe the Paretto effect: the top 10 repair actions account for more than 92% of the cumulative probability distribution.

Furthermore, we make the following observations. First, the order of repair actions (i.e. their likelihood of contributing to bug repair) varies significantly depending on the transaction bag used for computing the probability distribution. For instance: a statement insertion is #1 when we consider all transactions, but only #4 when considering transactions with a single semantic change (column 1-SC). In this case, the probability of observing a statement insertion varies from 34% to 12%. Second, even when the orders obtained from two different transaction bags resemble such as for ALL and 20-LC, the probability distribution still varies: for instance χ_{Stmt_Upd} is 29% for transaction bag ALL, but jumps to 34% for transaction bag 20-LC. Third, the difference between syntactic features (number of lines) and semantic features (number of semantic changes) has an impact for very small transactions: the columns of 1-LC and 1-SC are significantly different. On the contrary, for bigger transactions, the impact is less clear: the columns of 20-SC and 20-LC are similar. Al those observations also hold for repair model CTET, the complete table is given in the companion technical report [19].

Those results are a first answer to our question: different definitions of "repair transactions" yield different probability distribution over a repair model.

D. Link with Automated Program Repair

We have shown that one can base repair models on different strategies to extract repair transaction bags. There are certain analytical arguments against or for those different repair space topologies. For instance, selecting transactions based on the commit text makes a very strong assumption on the quality of software repository data, but ensures that the selected transactions contain at least one actual repair. Alternatively, small transactions (whether syntactically with the number of changed lines or semantically with the number of semantic code changes) indicate that transactions are dedicated to a single concern, that is likely to be a repair. However, small transactions may only see the tip of the fix iceberg, resulting in a distorted probability distribution over the repair space.

In all cases, the idea of building a repair model based on software history is meant to "learn" how human-developers fix programs. However, the results presented in this section are not enough to say whether one learning strategy is better than the other. We will provide first answers to this question in Section IV.

IV. SHAPING BUG FIXING WITH MCSHAPER

This section presents the concept of "repair shape", a search-based repair strategy called MCShaper that maximizes the likelihood of finding good shapes of repair based on probability distributions, and a way of comparing repair models and their probability distribution based on data from software repositories.

A. Decomposing The Automated Program Repair Process

We define the "repair shape" as a set of repair actions taken from a repair model. Informally, the shape of a patch is a kind of patch. For instance, the shape of adding an "if" throwing a exception signaling an incorrect input consists of two repair actions in repair space CTET: statement insertion of "if" and statement insertion of "throw". Wei et al. [3] call this a "fix schema", Weimer et al's equivalent [2] is "mutation operator".

An automated program repair process generally embeds repair shapes (either implicitly or explicitly). Actually, we consider that a repair approach can be sketched with three components: the *fault localizer* predicts where the repair is likely to be successful; the *shaper* determines which kind of repair may be applied (the repair actions); the *synthesizer* assigns concrete statements and values fitting into the repair shape (instantiating the repair action).

For instance, in Weimer [2]'s approach, the fault localizer uses test coverage data to concentrate on specific statements; the shaper randomly selects either a statement insertion, deletion or swap; the synthesizer picks concrete statements in the rest of the code. Wei et al.'s shaper [3] has three repair shapes (called fix schema).

Those three components may interact in many ways: for instance, the fault localization may depend on the repair to be tried and vice versa. Also, if the synthesizer fails to find concrete objects, values, or method calls for a given shape, this may trigger trying another shape.

B. The Monte Carlo Shaper Repair Algorithm

The Monte Carlo shaper repair algorithm (MCShaper) consists of predicting an unordered tuple of repair actions (from a set of repair actions called $\mathcal{R}$). With the

Input: C ▷ A bag of transactions
Output: The median number of attempts to find good repair shapes
begin
  $\Omega \leftarrow \{\}$ ▷ Result set
  $T, E \leftarrow split(C)$ ▷ Cross-validation: split C into **T**raining and **E**valuation data
  $M \leftarrow f(T)$ ▷ Train/compute a repair model (e.g. a probability distribution over repair actions)
  for $s \in E$ ▷ For all repairs observed in the repository
  do
    $n \leftarrow computeRepairability(s, M)$ ▷ How long to find this repair according to the repair model
    $\Omega \leftarrow R \cup n$ ▷ Store the "repairability" value of s
  return $median(\Omega)$ ▷ Returning the median number of attempts to find the repair shapes

Fig. 1. An Algorithm to Compare Fix Shaping Strategies. There may be different flavors of functions $split$, f and $computeRepairability$.

repair actions of repair model CT, a repair is for instance: $(StmtInsert, StmtDelete)$. Being a tuple, a repair shape may contain repeated repair actions (say $(StmtInsert, StmtInsert, StmtDelete)$, that's important to model the shape of real bug fixes where the same repair action is applied multiple times. Being unordered, $(StmtInsert, StmtDelete)$ and $(StmtDelete, StmtInsert)$ are logically and probabilistically equivalent.

To predict the shape of a repair, MCShaper uses a a probability distribution $\mathcal{P}$ over the repair actions. MCShaper assumes that each repair actions are independent. As a result, to predict a repair of size 3, it consists of randomly selecting three times a repair action according to P. MCShaper runs with an estimated number of repair actions as input. For instance, one may tell MCShaper to find a repair shape of 3 repair actions. [2]

MCShaper is a kind of random search-based repair. A key characteristic of random search-based repair is that two runs may find the same repair in a considerably different amount of times. That's why the repair success has to be measured by an average over many runs. The experimental results of Weimer et al. [2] present such average in terms of time spent to find the repair.

Similarly, at the level of source code changes, MCShaper may find the exact set of repair actions in 10 attempts in a first run and in 230 attempts in a second one. However, our repair model makes it possible to know the exact median number of attempts N that is needed to find a given repair R (demonstration given in the companion technical report [19]):

$$N = k \text{ such that } \sum_{i=1}^{k} p(1-p)^{i-1} \geq 0.5 \quad (1)$$

$$\text{with } p = \frac{n!}{\Pi_j(e_j!)} \times \Pi_{r \in R} P_{\mathcal{P}}(r)$$

and $e_j$is the number of occurrences of $r_j$inside R

For instance, the repair of revision 1.2 of Eclipse's CheckedTreeSelectionDialog[3] aforementioned consists of two inserted statements. According to a certain probability distribution over the space of repair actions (e.g. the given by 1-SC), the formula tells us that MCShaper needs 12 attempts to find the correct repair shape (in average) for this real bug.

Let's assume that a fault localization algorithm has to pinpoint a particular method out of 2000 methods. A powerful fault localization algorithm would quickly concentrate on the faulty method, hence would cut the repair space by 2000. Similarly, if there is an average of 2000 possible repair shapes per method (which is approximately the number of possible combinations of two repair actions in repair model CT), a repair approach correctly predicting the repair shape dramatically cuts the repair space and decreases the repair time.

MCShaper provides likely repair shapes; let us now present a way to thoroughly evaluate its efficiency.

C. Comparing Probability Distributions over Repair Actions From Versioning History

We have seen in Section IV-B that MCShaper concentrates on finding likely repair shapes based on a probability distribution over repair actions. The probability distribution $\mathcal{P}$ guiding MCShaper is crucial for the its efficiency: since one randomly selects repairs, a good distribution P results in *concentrating on likely repairs first*, i.e. the repair space is traversed in a guided way, by first exploring the parts of the space that are likely to be more fruitful. This poses two important questions: first, how to set up a probability distribution over repair actions; second, how to compare the efficiency of different probability distributions to find good repair shapes.

To set up the probability distribution over repair actions, we propose to learn them from software repositories. For instance, if many bug fixes are made of inserted method calls, the probability of applying such a repair action should be

[2]In practice one never knows the actual size of the repair shape. We choose this simplification in order to only have one independent variable in the experiment (the probability distribution). Similarly to the decomposition of the repair process, one can stack different size prediction algorithms on top of MCShaper. For instance, with an additional probability distribution on the size or with a weighted round-robin on the size before drawing a shape.

[3]"Fix for 19346 integrating changes from Sebastian Davids" http://goo.gl/d4OSi

Repair Size	1	2	3	4	5	6	7	8
argouml	**5** (996)	**13** (638)	**131** (386)	**303** (362)	**2182** (254)	**5667** (234)	**11243** (197)	**16388** (166)
carol	**5** (30)	**6** (15)	**171** (10)	**529** (10)	**550** (7)	**3026** (13)	**2918** (6)	**8156** (9)
columba	**3** (382)	**12** (255)	**52** (144)	**194** (146)	**688** (113)	**904** (108)	**4441** (73)	**20888** (94)
dnsjava	**4** (165)	**12** (139)	**169** (71)	**438** (82)	**710** (54)	**1752** (50)	**4553** (33)	**74202** (44)
jEdit	**3** (115)	**12** (84)	**48** (53)	**150** (48)	**758** (32)	**917** (30)	**1300** (29)	**7377** (32)
jboss	**4** (514)	**12** (353)	**117** (208)	**293** (189)	**854** (147)	**6114** (150)	**5981** (86)	**21468** (113)
jhotdraw6	**5** (21)	**12** (21)	**95** (9)	**244** (10)	**4754** (10)	**189** (3)	**22260** (5)	∞ (2)
junit	**3** (40)	**33** (39)	**268** (18)	**95563** (11)	**10294** (7)	**50700** (11)	**9970** (9)	∞ (6)
log4j	**5** (223)	**12** (134)	**178** (68)	**1128** (69)	**12561** (64)	**5718** (42)	**19743** (41)	**55275** (47)
org.eclipse.jdt.core	**4** (1605)	**14** (1023)	**105** (658)	**286** (629)	**1208** (392)	**3987** (416)	**7401** (314)	**14574** (310)
eclipse.ui.workbench	**3** (1182)	**12** (783)	**126** (413)	**289** (464)	**742** (326)	**3940** (304)	**6892** (216)	**14348** (192)
scarab	**4** (653)	**12** (346)	**127** (202)	**435** (159)	**640** (113)	**1860** (137)	**14445** (89)	**8722** (77)
struts	**3** (221)	**23** (133)	**173** (86)	**191** (103)	**1094** (61)	**5166** (77)	**5759** (39)	**26125** (34)
tomcat	**3** (279)	**12** (167)	**124** (111)	**311** (119)	**749** (85)	**890** (87)	**2532** (61)	**15599** (51)

TABLE IV
THE MEDIAN NUMBER OF ATTEMPTS (IN BOLD) REQUIRED TO FIND THE CORRECT REPAIR SHAPE OF FIX TRANSACTIONS. THE VALUES IN BRACKETS INDICATE THE NUMBER OF FIX TRANSACTIONS TESTED PER PROJECT AND PER TRANSACTION SIZE FOR REPAIR MODEL CT. FOR SMALL TRANSACTIONS, FINDING THE CORRECT REPAIR SHAPE IS DONE IN LESS THAN 100 ATTEMPTS.

high. Consequently, we feed MCShaper with the probability distributions discussed in Section III.

Despite our single method (learning the probability distributions from software repositories), we have shown that there is no single way to compute them (they involve different heuristics). To compare different distributions against each other, we set up the following process.

One first selects bug repair transactions in the versioning history. Then, for each bug repair transaction, one extracts its repair shape (as a set of repair actions of a repair model). Then one computes the average time that MCShaper needs to find this repair shape thanks to equation 1.

Let us assume two probability distributions $\mathcal{P}_1$ and $\mathcal{P}_2$ over a repair model and four fixes $(F_1 \dots F_4)$ consisting of two repair actions and observed in a repository. Let us assume that the time (in number of attempts) to find the exact shape of $F_1 \dots F_4$ with MCShaper according to $\mathcal{P}_1$ is $(5, 26, 9, 12)$ and according to $\mathcal{P}_2$ $(25, 137, 31, 45)$. In this case, it's clear that the probability distribution $\mathcal{P}_1$ enables us to find the correct repair shapes faster (the shaping time for $\mathcal{P}_1$ are lower). Beyond this example, by applying the same process over real bug repairs found in a software repository, our process enables us to select the best probability distributions for MCShaper (given a repair model).

Since MCShaper is parametrized by a number of repair actions, we instantiate this process for all bug repair transactions of a certain size (in terms of semantic changes). This means that our process determines the best probability distribution for a given bug fix shape size.

D. Cross-Validation

We need a last bit of method before running the whole comparison process between the probability distribution discussed in Section III. We compute different probability distributions $\mathcal{P}_x$ from transaction bags found in repositories. We evaluate the time to find the shape of real fixes, that are also found in repositories. It means that we use the same data to estimate the model parameters and to evaluate it.

Logically, we use cross-validation to solve this potential bias: we always use different sets of transactions to estimate $\mathcal{P}$ and to calculate the average number of attempts required to find a correct repair shape.

Since we have a dataset of 14 independent software repositories, we use this dataset structure for cross-validation. We take one repository for extracting repair shapes and the remaining 13 projects to calibrate the repair model (i.e. to compute the probability distributions). We repeat the process 14 times, by testing each of the 14 projects separately. In other terms, we try to predict real repair shapes found in one repository from data learned on other software projects, meaning that our mined repair knowledge is meant to be project- and domain-independent

Figure 1 sums up this algorithm to compare fix shaping strategies. From a bag of transactions C, function $split$ creates a set of testing transactions and a set of evaluation transactions. Then, one trains a repair model (with function f), in the case of MCShaper it means computing a probability distribution on a specific bag of transactions. Finally, for each repair of the testing data, one computes its "repairability" according to the repair model (in the case of MCShaper with Equation 1). The algorithm returns the median repairability, i.e. the median number of attempts required to repair the test data.

E. Empirical Results

We run our fix shaping process on our dataset of 14 repositories of Java software considering two repair models: CT and CTET (see Section II-C). We remind that CT consists of 41 repair actions and CTET of 173 repair actions. For both repair models, we have tested the different heuristics of III-B to compute the median repair time: all transactions (ALL); one-line transactions (1-LC); one semantic change (1-CS); 20 lines transactions (20-LC); 20 semantic source code changes (20-SC); transactions with commit text containing "bug", "fix", "patch" (BFP); a baseline of a uniform distribution over the repair model (EQP for equally-distributed probability).

We extracted all bug fix transactions transactions with less than 8 semantic changes from our dataset. For instance, the

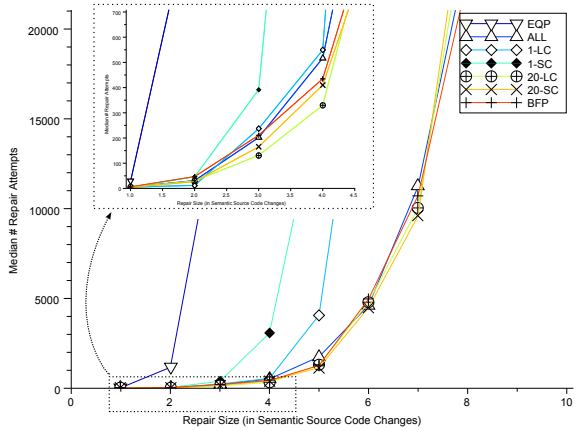

Fig. 2. The repairability of small transactions in repair model CT. Certain probability distributions yield a median repair time that is much lower than others.

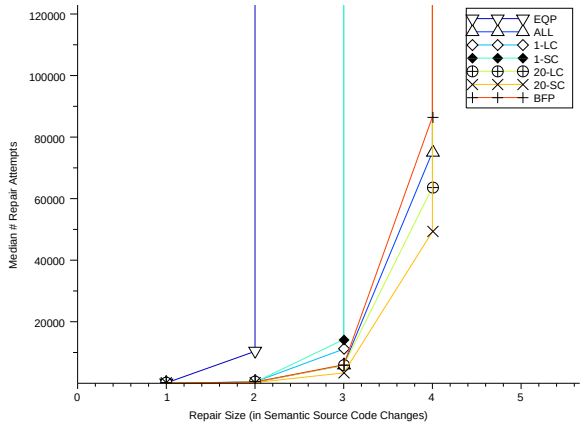

Fig. 3. The repairability of small transactions in repair space CTET. There is no way to find the repair shapes of transactions larger than 4 semantic code changes.

versioning repository of DNSJava contains 165 transactions of 1 repair action, 139 transactions of size 2, 71 transactions of size 3, etc. The biggest number of available repair tests are in jdt.core (1605 fixes consist of one semantic change), while Jhotdraw has almost only 2 transactions of 8 semantic changes. We then compute the median number of attempts to find the correct shape of those $23,048$ fix transactions. Since this number highly depends on the probability distributions $\mathcal{P}_x$, we compute the median repair time for all combinations of fix size transactions, project, and heuristics discussed above.

Table IV presents the results of this evaluation for repair space CT. For each project, the bracketed values give the number of transactions per transaction size (size in number of semantic changes) and per project (for instance there are 996 transactions of one semantic change in the history of argouml). Then the bold values give the median repairability in terms of number of attempts for MCShaper to find the correct repair shape. One has six repairability values that depend on the six heuristics; since the table would be overloaded with all values, we present only one: the best one (i.e. the minimum since we are interested in minimizing the repair time). For instance, over 996 fix transactions of size 1 in the argouml repository, it takes an average of 5 attempts to find the correct repair shape. On the contrary, for the 51 transactions of size 8 in the tomcat repository, it takes an average of 15599 attempts to find the correct repair shape.

Those results are encouraging: for small transactions, MCShaper is able to find the correct repair shape in only a handful of attempts. The probability distribution over the repair model seems to drive the search efficiently.

Furthermore, finding the correct repair shapes of larger transactions (up to 8 semantic changes) has an order of magnitude of 10^4 and not more. Theoretically, for a given fix shape of n semantic changes, the size of the repair space is the number of repair actions of the model at the power of n (e.g. $|CT|^n$). For CT and $n = 4$, this results in a space of $41^4 = 2,825,761$ possible shapes (approx 10^6). In practice, overall all projects, MCShaper can find the correct small shape (i.e. less or equal than 3 changes) in a median time lower than 200 attempts. This again show that the probability distribution over the repair model is so unbalanced that the likelihood of possible shapes is concentrated on less than 10^4 shapes (i.e. that the probability density over $|CT|^n$ is really sparse).

Table IV willingly hides the difference between the different heuristics used to compute the distributions of probability over the repair model. What is the best heuristic to feed MCShaper?

1) The Best Heuristic for Computing Repair Actions: For each repair shape size of Table IV and heuristic, we computed the median repairability over all projects of the dataset (a median of median number of attempts). We also compute the median repairability for a baseline of a uniform distribution (EQP) over the repair model (i.e. $\forall i, P(r_i) = 1/|CT|$)). Figure 2 presents this data for repair model CT. It shows the median number of attempts required to identify correct repair shapes as Y-axis. The X-axis is the number of repair actions in the repair test (the size). Each line represents probability estimation heuristics.

Figure 2 gives us important pieces of information. First, the heuristics yield different repair time. For instance, the repair time for heuristic 1-SC is generally higher than for 20-SC. Overall, there is a clear order between the repairability time, and heuristic 20-LC gives the best results. Interestingly, certain heuristics are inappropriate for MCShaper: the resulting distributions of probability results in a repair time that explodes even for small shape (this is the case for a uniform distribution EQP even for shape of size 3). Also, all median repair times tend toward infinity for shape of size larger than 9. While for small shapes, 20-LC is slightly better (see zoom in the upper left-hand side), as of 5 repair actions, BFP, ALL, 20-SC and 20-LC are equivalent indicating that there is no fundamental differences between syntactic and semantic heuristics. Finally, although 1-SC and 1-LC are not good over

many shape size, we note that that for small shape of size 1 and 2, there are better. This is explained by the empirical setup (where we decompose transactions by shape size). Regarding the arbitrariness of 20 lines, note that we have tested all N-LC from 10 changed lines (10-LC) to 100 lines (100-LC), although there N for which the repair time is smaller or any shape size, 20-LC is better for shape sizes of less than 4 changes.

To sum up, *with repair to probabilistic shaping, some heuristics are bad (EQP, 1-SC, 1-LC), but there is no clearly better ones*.

Do those findings hold for repair model CTET, which has a finer grain?

2) The Difference between Repair Models: We have also run the whole evaluation with the repair model CTET (see II-C). The empirical results are given in appendix (in the same form as Table IV).

Figure 3 is the sibling of figure 2 for repair model CTET. They look rather different. The main striking point is that when MCShaper works with repair model CTET, it is able to find the correct repair shape for fixes that are no larger than 4 semantic changes. After that, the arithmetic of very low probabilities results in virtually infinite time to find the correct repair shape. On the contrary, in the repair model CT, even for fixes of 7 changes, MCShaper could find the correct shape in a finite number of attempts. Finally, in this repair model the average time to find a correct repair shape is several times larger than in CT (in CT, the shape of fixes of size 3 can be find in approx. 200 attempts, in CTET, it's more around 6000).

This major difference illustrates the tension between the richness of the repair model and the ease of fixing bugs automatically. When MCShaper works in CT, it finds likely repair shapes quickly (less than 5000 attempts), even for large repair: but then the synthesis phase (finding a concrete instance of a repair action, see IV-A) is harder. When MCShaper works in CTET, the "synthesis space" (see IV-A) is much smaller, because one has much more information for finding the exact values to instantiate the repair action. For instance, if the predicted repair action in CTET consists of inserting a method call, it just remains to predict the target object, the method and its parameters.

In other words, there is a balance between finding correct repair actions and finding concrete repair actions. When the repair actions are more abstract, it results in a larger synthesis space, when repair actions are more concrete, it hampers the likelihood of being able to concentrate on likely repair shapes first. It is similar to the difference between prescribing aspirin (it has a high likelihood to contribute to healing, but only partially) and prescribing a specific medicine (one can try many medicines before finding the perfect one).

The key insight behind the efficiency of Weimer's repair model [2] is to bet on two competing horses at the same time: to maximize the applicability of repair actions, they only have three repair actions (statement insertion, deletion and swapping), and to minimize the synthesis space, they only reuse existing statements.

Even if there is no definitive answer at this point, we tend to think that the profile of CT is better, because it is better from two viewpoints: it enables us to find bigger correct repair shapes (good) in a smaller amount of time (good).

V. Related Work

a) Empirical Studies of Versioning Transactions: Purushothaman and Perry [12] studied small commits (in terms of number of lines of code) of proprietary software at Lucent Technology. They showed the impact of small commits with respect to introducing new bugs, and whether they are oriented toward corrective, perfective or adaptive maintenance. German [9] ask different research questions on what he calls "modification requests" (small improvements or bug fix), in particular with respect to authorship and change coupling (files that are often changed together). Alali and colleagues [11] discussed the relations between different size metrics for commits (# of files, LOC and # of hunks), along the same line as Hattori and Lanza [10] who also consider the relationship between commit keywords and engineering activities. Finally, Hindle et al. [8], [23] focus on large commits, to determine whether they reflect specific engineering activities such as license modifications. Compared to these studies on commits that mostly focus, on metadata (e.f. authorship, commit text) or size metrics (number of changer files, number of hunks, etc.), we discuss the content of commits and the kind of source code change they contain. Fluri et al. [24] and Vaucher et al. [25] studied the versioning history to find patterns of change, i.e. groups of similar versioning transactions.

Pan et al. [21] manually identified 27 bug fix patterns on Java software. Those patterns are precise enough to be automatically extractable from software repositories. They provide and discuss the frequencies of the occurrence of those patterns in 7 open source projects. This work is closely related to ours: we both identify automatically extractable repair actions of software. The main difference is that our repair actions are discovered fully automatically based on semantic differencing (there is no prior manual analysis to find them). Furthermore, since our repair actions are meant to be used in an automated program repair setup, they are smaller and more atomic.

Kim and et al. [26] use versioning history to mine project-specific bug fix patterns. Williams and Hollingsworth [27] also learn some repair knowledge from versioning history. They mine how to statically recognize where checks on return values should be inserted. Livshits and Zimmermann [13] mine co-changed method calls. The difference with those close pieces of research is that we enlarge the scope of mined knowledge: from project-specific knowledge [26] to domain-independant repair actions, and from one single repair action [27], [13] to 41 and 173 repair actions.

b) Semantic Differencing: The evaluation of semantic differencing tools often gives hints about common change actions of software. For instance, Raghavan et al. [16] showed the six most common types of changes for the Apache web server and the GCC compiler, the number one being "Altering existing function bodies". This example clearly shows the

difference with our work: we provide change and repair actions at a very fine grain. Similarly, Neamtiu et al. [17] gives interesting numerical findings about software evolution such as the evolution of added functions and global variables of C code. It also remains at grain that is coarser compared to our analysis. Fluri et al. [7] gives some frequency numbers of their change types in order to validate the accuracy and the runtime performance of their distilling algorithm. Those numbers were not — and not meant to be — representative of the overall abundance of change types.

c) Automated Software Repair: We've already mentioned many pieces of work on automated software repair (incl. [1], [2], [3], [4], [5], [28]). Those pieces of research are fundamental sources of inspiration of our work. As stated and explained all along the paper, we have built on their insights and results to get ours.

VI. Conclusion

In this paper, we have presented the idea that one can mine repair actions from software repositories. In other terms, one can learn from past bug fixes the main repair actions (e.g. adding a method call). Those repair actions are meant to be generic enough to be independent of bug types and software domain. We have discussed and applied a methodology to mine the repair actions of 62179 versioning transactions extracted from 14 repositories of 14 open-source projects.

We have largely discussed the rationales and consequences of adding a probability distribution on top of a repair model. We have shown that certain distributions over repair actions can result in an infinite time (in average) to find a repair shape while other fine-tuned distributions enable us to find a repair shape in hundreds of repair attempts.

We have presented our views on a decomposed repair process, such that, for instance, one can plug a shaping repair algorithm into an existing fault localization system. We are currently working on coupling MCShaper with fault localization algorithms and on instantiating repair actions (repair synthesis) in order to fix actual bugs. We envision many pluggable approaches for repair synthesis, starting from the fascinating "seed" strategy of Weimer and colleagues: cherry picking pieces of code from the rest of the code base.

References

[1] W. Weimer, "Patches as better bug reports," in *Proceedings of the International Conference on Generative Programming and Component Engineering*, 2006.

[2] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the International Conference on Software Engineering*, 2009.

[3] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the International Symposium on Software Testing and Analysis*, AC, 2010.

[4] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the International Conference on Automated Software Engineering*, 2009.

[5] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494–3514, 2011.

[6] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

[7] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725 –743, nov. 2007.

[8] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the International Working Conference on Mining Software Repositories*, 2008.

[9] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Softw. Engineering*, vol. 11, pp. 369–393, Sept. 2006.

[10] L. Hattori and M. Lanza, "On the nature of commits," in *Automated Software Engineering - Workshops*, pp. 63 –71, sept. 2008.

[11] A. Alali, H. Kagdi, and J. Maletic, "What's a typical commit? a characterization of open source software repositories," in *IEEE International Conference on Program Comprehension*, 2008.

[12] R. Purushothaman and D. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, pp. 511 – 526, june 2005.

[13] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," in *Proceedings of the European software engineering conference held jointly with International Symposium on Foundations of Software Engineering*, 2005.

[14] E. Giger, M. Pinzger, H. Gall, T. Xie, and T. Zimmermann, "Comparing fine-grained source code changes and code churn for bug prediction," in *Working Conference on Mining Software Repositories*, 2011.

[15] M. Monperrus and M. Martinez, "Conservation and replication with cvs-vintage: A dataset of cvs repositories of java software," tech. rep., INRIA, 2012, to appear.

[16] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: a semantic-graph differencing tool for studying changes in large code bases," in *20th IEEE International Conference on Software Maintenance*, 2004.

[17] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the International Workshop on Mining Software Repositories*, 2005.

[18] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," *Proceedings of the IEEE/ACM international conference on Automated software engineering*, p. 433, 2007.

[19] M. Martinez and M. Monperrus, "Appendix of "mining repair actions for automated program fixing"," tech. rep., INRIA, 2012 (to appear). Available at http://goo.gl/vCqQH.

[20] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on cvs," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2010.

[21] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, pp. 286–315, Aug. 2008.

[22] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *SIGSOFT FSE*, pp. 15–25, 2011.

[23] A. Hindle, D. German, M. Godfrey, and R. Holt, "Automatic classication of large changes into maintenance categories," in *International Conference on Program Comprehension*, 2009.

[24] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Proceedings of the International Conference on Automated Software Engineering*, 2008.

[25] S. Vaucher, H. Sahraoui, and J. Vaucher, "Discovering new change patterns in object-oriented systems," in *Proceedings of the Working Conference on Reverse Engineering*, 2008.

[26] S. Kim, K. Pan, and E. J. Whitehead, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.

[27] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.

[28] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*, (New York, NY, USA), pp. 237–246, ACM, 2010.