Inferring Program Transformations From Singular Examples via Big Code

Jiajun Jiang†, Luyao Ren†, Yingfei Xiong†, Lingming Zhang‡
†Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)
†Department of Computer Science and Technology, EECS, Peking University, Beijing, China
‡University of Texas at Dallas, USA
{jiajun.jiang, rly, xiongyf}@pku.edu.cn, lingming.zhang@utdallas.edu

***Abstract*—Inferring program transformations from concrete program changes has many potential uses, such as applying systematic program edits, refactoring, and automated program repair. Existing work for inferring program transformations usually rely on statistical information over a potentially large set of program-change examples. However, in many practical scenarios we do not have such a large set of program-change examples.**

In this paper, we address the challenge of inferring a program transformation from one single example. Our core insight is that "big code" can provide effective guide for the generalization of a concrete change into a program transformation, i.e., code elements appearing in many files are general and should not be abstracted away. We first propose a framework for transformation inference, where programs are represented as hypergraphs to enable fine-grained generalization of transformations. We then design a transformation inference approach, GENPAT, that infers a program transformation based on code context and statistics from a big code corpus.

We have evaluated GENPAT under two distinct application scenarios, systematic editing and program repair. The evaluation on systematic editing shows that GENPAT significantly outperforms a state-of-the-art approach, SYDIT, with up to 5.5x correctly transformed cases. The evaluation on program repair suggests that GENPAT has the potential to be integrated in advanced program repair tools – GENPAT successfully repaired 19 real-world bugs in the Defects4J benchmark by simply applying transformations inferred from existing patches, where 4 bugs have never been repaired by any existing technique. Overall, the evaluation results suggest that GENPAT is effective for transformation inference and can potentially be adopted for many different applications.

***Index Terms*—Pattern generation, Program adaptation, Code abstraction**

I. INTRODUCTION

Modern program development is often repetitive, where the same changes are applied over and over again in different positions or in different projects, by the same or different developers. Inferring program transformations from change examples could automate the changes of the same type, and has many potential uses such as systematically editing many places in the source code [1], fixing bugs based on patches of recurring bugs [2]–[4], porting commits among forked projects [5], [6], adapting client code for incompatible API changes [7], [8], refactoring [9], [10], etc.

*Yingfei Xiong is the corresponding author. This work was partially done when Jiajun Jiang was a visiting student in UT Dallas.

A key challenge in transformation inference is to decide what can be generalized in the transformation. As an example, let us consider the following change:

```
f(a, b) ⟹ f(g(a),b)
```

A possible transformation could be the following one:

> Wrapping with `g` any element that
> - is a variable
> - has type integer
> - has identifier name `a`

We may also consider making the transformation more general such as the following one:

> Wrapping with `g` any element that
> - is a variable
> - has type integer

We may also consider the context of the change to make the transformation more specific such as the following one:

> Wrapping with `g` any element that
> - is a variable
> - has type integer
> - is the first argument of a call to f

Making the transformation too specific may decrease recall, i.e., missing cases that should be transformed. Making the transformation too general may decrease precision, i.e., transforming cases that should not be transformed. Therefore, selecting a suitable level of generalization is critical to the quality of the inferred program transformation.

A typical method adopted by many existing techniques [11]–[13] is to learn from many examples, where the statistical information from many examples is used to decide which part should be concrete in the transformation and which part should be abstracted away. In the above example, if there are many change examples that wrap the first arguments of `f` with `g` and the first arguments have many different names, we know that the last transformation should be the desirable one, and information such as variable name `a` should be abstracted away. However, such an approach requires many examples as the training set. In practice, we often do not have so many examples. For example, Genesis [13] uses hundreds of patches for the same type of bugs to generate transformations, while in practice the repetitiveness of patches tends to be tenuous [14],

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00033

and only one or a few patches can be located for many types of bugs.

On the other hand, a few approaches [12], [15] have tried to reduce the needed number of examples by using predefined rules to decide what part in the concrete changes should be abstracted away in the transformation, i.e., always ignore the variable names and allow to match variables with any name [15]. However, as shown in the next section, predefined rules cannot capture different situations and often fail to produce the desired transformation.

In this paper, we address the challenge of inferring a program transformation from one single example. Our core insight is to learn from "big code", utilizing the statistical information in a large code corpus to guide the generalization from a change example to a transformation. More concretely, the elements that appear in many files are potentially general and should be kept in the transformation in order to capture the transformation for all such instances. Along this line, we first propose a general framework for transformation inference from an example, where a hypergraph is used to represent source code and fine-grained transformation tuning is enabled by selecting elements and their attributes from the hypergraph. We then instantiate the framework with a transformation inference algorithm that fine-tunes the hypergraph information based on code statistics from a big code corpus. We have already implemented our approach as a tool, GENPAT, and evaluated GENPAT in two distinct application scenarios. In the first scenario, we employed GENPAT to perform systematic editing as studied by Meng et al. [15] but with a much larger dataset. The result shows that GENPAT significantly outperforms state-of-the-art SYDIT with an up to 5.5x improvement in terms of correctly generated transformations. In the second scenario, we explore the potential of using GENPAT to repair bugs by simply mining and applying fixing patterns from existing patches. Although not designed as a comprehensive and standalone repair technique, GENPAT successfully fixed 19 bugs in a subset of the commonly used Defects4J [16] benchmark. Particularly, 4 bugs have never been fixed by any existing technique as far as we know. The results suggest that GENPAT is potentially useful in both systematic editing and program repair and indicate a promising future for adopting GENPAT in practical systems with program transformations.

In summary, this paper makes the following contributions:

- A framework for transformation inference from a single example by representing code as a hypergraph to allow fine-grained generalization of the transformation.
- An algorithm to instantiate the framework by defining the rules for selection based on the code context and statistics in a large code corpus.
- An implementation of the proposed technique in Java language, called GENPAT, which can be publicly accessed at https://github.com/xgdsmileboy/GenPat.
- An evaluation with GENPAT on two distinct practical application scenarios, showing the effectiveness of the proposed framework and calling for future research to integrate GENPAT for advanced program-transformation-based systems, including systematic-editing and program-repair systems.

II. RELATED WORK

In this section, we introduce the most related works to this paper. Existing techniques have explored different strategies for transformation extraction and two categories of transformations have been proposed. The first one is *executable transformations*, which can be applied directly to modify a code snippet. The second one is *abstract transformations*, which cannot be applied directly but constrain a space of possible transformation results. In other words, executable transformations are functions, while abstract transformations are binary relations that are not functions. Abstract transformations are useful in guiding other technical processes, such as ranking candidates in automatic program repair [17]–[19]. In the following, we will introduce the most related approaches from these two categories in detail. Also, we will discuss few-shot learning problem in machine learning domain, of which the transformation inference problem can be seen as an instance.

A. Executable Transformation Generation

As explained in the introduction, the key challenge of transformation inference is to decide how to generalize a change into a transformation. To approach this challenge, existing techniques proposed to utilize different strategies, such as learning from many examples or employing predefined rules.

Learning from many examples. Multiple existing techniques learn practical transformations from a set of examples with similar code changes. The basic idea is that shared code elements across change examples are critical parts for the transformation and should be preserved, while the other parts tend to be specific to some individual examples and thus will be discarded. Andersen et al. [20], [21] proposed `spdiff`, which extracts a set of term replacement patches from each example, and then takes the longest common subpatches as a transformation pattern. Meng et al. [11] proposed LASE that learns edit scripts from multiple examples. LASE extracts a set of edit operations from each example, keeps the common operations, and extracts the context of the common operations to form a transformation. Reudismam et al. [12] proposed REFAZER, which learns syntactic code transformations from examples. REFAZER takes a program synthesis perspective and searches for a transformation program that is consistent with all the examples. Long et al. [13] proposed Genesis. It infers a template AST from existing patches, which can cover all mined examples. Bavishi et al. [22] proposed to mine repair strategies (or repair patterns) from examples for fixing bugs reported by static analyzers, which clusters similar edit examples for pattern abstraction (i.e., Synthesis of Context-Matchers) via leveraging a DSL for representation. Nguyen et al [23] proposed CPATMINER that aims to mine semantic code change patterns from code corpus and represents patterns as graphs. CPATMINER also depends on the repetitiveness of

code changes and leverages graph isomorphism for pattern clustering. Similarly, Molderez et al [24] leveraged frequent itemset mining algorithm to learn edit scripts of code changes from histories of open-source repositories, and employed them for code change recommendation.

As discussed in the introduction, to achieve a good level of generalizability, these approaches require a non-trivial number of examples, which are often difficult to obtain in practice. For example, in the scenario of program synthesis, these approaches have been successfully applied to only the most common bugs [13] or the bugs in student assignments [12], where a large number of patches can be found for the same type of bug. However, in practice, the repetitiveness of patches tend to be tenuous [14], and only one or a few patches can be located for many types of bugs.

Inferring transformations with predefined rules. Several approaches rely on predefined rules to infer a suitable transformation. A typical approach is SYDIT, which also infers a transformation from one example, and is similar to our goal. Given a change, SYDIT first selects all related statements that have dependencies with the changed statement, then abstracts away all names (variable name, type name, method name, etc) in the statements and leaves only the structure of the statements. Then the structure is used to match other places and perform the change. However, there are many cases that we may need to abstract away part of the structure or keep some names in the transformation, where SYDIT cannot extract the desirable transformation. As our evaluation will show later, our approach significantly outperforms SYDIT with an up to 5.5x improvement. Approaches relying on multiple examples may also use predefined rules to select the desired transformation if the examples are not enough to ensure the quality of the transformation. For example, REFAZER employs a set of rules to rank the transformations if the synthesizer found multiple possible transformations.

Defining transformation manually. There are some other approaches that perform code changes with manually defined transformations. For example, Kim et al. [25] manually defined a set of transformations for automatic program repair after analyzing a corpus of human patches. Similarly, Liu and Zhong [26] defined transformations (a.k.a. repair templates) with analyzing code samples from StackOverflow. Molderez and De Roover [27] proposed to refine a manually defined template with a suite of mutation operations, which recommends changes to the templates iteratively. Additionally, to ease the description of transformations, a set of DSLs have been proposed by previous studies [7], [8], [21], [28]–[32] for program migration or API updating. These techniques provide a way for developers to systematically update a current program with manually defined transformations. However, even with the help of DSL, manually defining transformations is not easy, and automatic transformation inference is desirable in many situations.

B. Abstract Transformation as Guidance

Recently, a number of existing techniques were proposed to extract transformations from a set of examples for guiding other technical processes. In particular, transformations are often used in automatic program repair to guide the patch generation as the complete search space can be too huge [33]. For example, Xuan et al. [17] and Wen et al. [19] leveraged transformations from historical bug fixes as program repair templates. Similarly, Jiang et al. [18] proposed to use such transformations to refine the search space of patch generation. Also, other researches proposed to use such transformations for patch prioritization [34]. The core insight behind these techniques is that frequently appeared transformations in history bug fixes have higher possibility to repair a bug, and thus can be utilized to refine the patch space. However, these transformations cannot be directly applied and can be much more abstract compared to those executable transformations.

C. Few-Shot Learning

Few-shot learning [35] attempts to train a machine-learning model with a very small set of training data, and is often considered a grand challenge in the machine learning domain. A typical approach to few-shot learning is to utilize data that are beyond the current task, and train a meta-level model with these data, which can be used as a basis for the few-shot learning task. Our problem is similar to few-shot learning as we try to generalize a transformation from just one example. Also, we learn meta-level information from big code for the transformation inference, where the idea is similar to the approach of few-shot learning. On the other hand, the current few-shot learning techniques are still designed for classic classification problem over feature vectors, and thus cannot be applied to the transformation inference problem since it is not a classification procedure.

III. MOTIVATING EXAMPLE

In this section, we motivate the problem of transformation inference with an example in the systematic editing scenario [15]. In a typical systematic editing scenario, the programmer would like to perform the same change on a series of places. She would first change one place, and ask the system to extract a transformation from the change, then navigate to the next place and invoke the transformation there. The process is similar to a copy-paste clipboard operation process except that only the transformation is “copied” and “pasted”.

Listing 1 shows an example requiring systematic editing. Here “-” denotes deleted code lines and “+” denotes newly introduced code lines. The grayed description on the top gives the detailed information related to the code changes, including the GitHub link of the corresponding commit, fix message and changed classes. Particularly, there are two separate code changes, where the first one at line 68 is the change from which a transformation would be inferred, and the second one at line 35 is the ideal change that we expect the transformation to produce.

```
Commit : github.com/junit-team/junit4/commit/75f7892
Message: Removed nascent category implementation
Source : src.main.java.org.junit.runner.Description
==========
// first case for pattern generation
67   Description createDescription(Class<?> testClass){
68 -   return new Description(testClass.getName(),null,
68 +   return new Description(testClass.getName(),
69                      testClass.getAnnotations());
70   }

// candidate place to apply the above pattern
// Sydit failed to apply the above pattern because the
// variable ``name'' cannot match the method ``getName()''
// while GenPat successfully applies it.
34   Description createDescription(String name,
                                   Annotation... annotations){
35 -   return new Description(name,null,annotations);
35 +   return new Description(name,annotations);
35   }
```

Listing 1. An example that SYDIT fails to apply pattern.

As we can analyze from the two examples, a desirable transformation should delete the second argument of a call to `Description` if it is `null`. In other words, the first argument `testClass.getName()` and the third argument `testClass.getAnnotations()` are specific to the local change and should not be considered as part of the context of the transformation. The challenge is to know which part should be kept in the transformation and which part should be abstracted away, i.e., deciding how to generalize the change.

As discussed previously, existing approaches rely on either multiple examples or predefined rules. However, providing multiple examples is often not desirable or feasible. For example, in systematic editing, the examples are provided by the user, and asking the user to provide multiple and preferably diverse examples significantly increases the cost of using this approach. In the scenario of bug repair, for many types of bugs, only one existing patch can be found, and we have to produce a transformation out of the patch. For example, Listing 2 shows a patch that inserts an equality check between two `Object` arguments into a method returning `boolean`. From this patch, our approach successfully inferred a transformation and fixed bug Mockito-22 in Defects4J [16] benchmark, which is shown in Listing 3 and has never been fixed by any previous technique. However, we found only one such change instance from more than 1 million historical code change examples of open-source Java projects on GitHub from 2011 to 2016.

```
Commit : github.com/clitnak/mcrailo/commit/8e76da8
Message: solved ticket #RAILO-2411
Source : railo-java.railo-core.src.railo.runtime.op.Operator
==========
526 boolean _equalsComplexEL(Object left,Object right,...){
527+  if(left==right){
528+     return true;
529+  }
530   if(Decision.isSimpV(left)&&Decision.isSimpV(right)){
```

Listing 2. Referenced history patch to fix Mockito-22.

```
12  public static boolean areEqual(Object o1,Object o2){
13 +   if(o1==o2){
14 +      return true;
15 +   }
16     if(o1==null||o2==null){
```

Listing 3. Patch of Mockito-22.

On the other hand, predefined rules hardly meet the divergent requirements of different situations. For example, SYDIT has a predefined rule to abstract away all variable/method/type names and keep only the structure. However, in this case, it would keep the structure of the first and the third arguments, requiring them to take the `o.m()` form. It would also discard the name of the method call `Description`. Both are not desirable.

Our approach decides how to generalize the change by analyzing the "big code". We count the number of files where an element appears in a large code corpus. If an element appears in many files, it is probably a general element and should be kept in the transformation to transform other sibling instances, otherwise it is probably specific to the current change and should be abstracted away. In this example, `testClass.getName()` and `testClass.getAnnotations()` can be seldomly found in the codebase and thus is abstracted away. On the other hand, `Description` and `null` are both frequent and thus are kept in the transformation.

While the general idea is simple, realizing the idea is not easy and faces multiple challenges:

- *Abstraction.* We need to have a flexible representation of the transformation, where the level of generalization can be adjusted at a fine-grained level.
- *Match.* The representation should be flexible to allow matching code with different attributes (such as the static value type) or different relations (such as data dependency).
- *Transformation.* The matched code pieces should be consistent with the transformation, i.e., when some code pieces are matched, the transformation must be able to be replayed on these code pieces.

In the next section, we will propose a framework for transformation inference to address the above challenges.

IV. FRAMEWORK OF TRANSFORMATION INFERENCE

In this section, we introduce the framework of transformation inference. Here we consider a transformation as first a pattern to match code pieces and a sequence of modification operations to change the code pieces. To address the challenges mentioned above, we make the following design decisions.

- *Match.* To ensure the code elements could be flexibly matched by their attributes and relations, we abstract source code into a hypergraph, where the nodes are AST nodes with their attributes (called *code elements*) and the hyperedges are relations among nodes.
- *Abstraction.* We further introduce a match relation between hypergraphs such that a graph can be matched by a more abstract graph with possibly fewer elements and attributes. In this way, we can abstract a hypergraph into a pattern at a fine-grained level by selecting elements and attributes that should be kept in the pattern.
- *Transformation.* To ensure the matched code elements are transformable, we use elements and attributes as the

interface between the pattern and the modifications. The modifications specify the elements and attributes that must be matched to make the transformation applicable, and the pattern ensures to match these elements and attributes.

Now we introduce the design in detail. We start by defining code elements. Intuitively, a code element captures a node in an AST, and the attributes of the AST node that we are interested in.

Definition 1 (Code Element)**.** A (code) element is a pair $\langle id, attrs \rangle$ where id is an element ID and $attrs$ is a set of attributes, where each attribute is a pair $\langle name, value \rangle$.

In our current implementation, we mainly consider three attributes, AST node type (such as `Statement` or `Variable`), content (such as `a+b` or `>=`, which is the string representation of the complete subtree), and static value type (such as `String` or `int`).

The code element captures a single AST node and its attributes, but not the relation between AST nodes. To capture the relations, we further define code hypergraph as a collection of code elements and their relations.

Definition 2 (Code Hypergraph)**.** A (code) hypergraph is a pair $\langle E, R \rangle$, where E is a set of elements and R is a set of hyperedges, where each hyperedge is a pair $\langle rname, r \rangle$ consisting of a relation name $rname$ and a relation $r \subseteq E^k$ for some k, where E^k denotes the k-ary Cartesian power of E.

The relation r can be either directed or undirected, but in our implementation, we consider mainly three directed relations, the parent relation in an AST, the ancestor relation which is the transitive closure of the parent relation, and the intra-procedural data dependency between l-values in the program. We only consider data dependencies (ignoring control-flow dependencies) to avoid over-approximations [36], [37] for lightweight analysis. Please also note that when the parent relation is included, a hypergraph subsumes an AST. Additionally, the ancestor relation is necessary as it may still guarantee the program structure match even when two nodes do not have direct parent-child relation.

For example, Figure 1 shows the code hypergraph of the two code snippets in Listing 1. Each node in the graph represents a code element, where their IDs and attributes are listed. Three types of relations are shown in the graph, the black lines represent the parent relation and the blue lines represent the data dependency relation. Note that there is no data dependency between p_3 and node p_6 while the omitted child node of p_3 has data dependency on p_6. For clarity, we ignore the ancestor relation in the figure, e.g., node p_1 is the ancestor of node p_3.

After we have a code hypergraph, we can define a pattern that matches elements in the graph. Here we treat a pattern uniformly also as a hypergraph. A pattern matches some code elements if both the attributes and the relations on the pattern hypergraph match those of the target code elements. Formally, we first define the match between elements.

Definition 3 (Element Match)**.** An element $\langle id, attrs \rangle$ is said to match another element $\langle id', attrs' \rangle$, if $\forall \langle name, value \rangle \in attrs,\ \langle name, value \rangle \in attrs'$.

Based on the match between code elements, we define the match between hypergraphs.

Definition 4 (Hypergraph Match)**.** A code hypergraph $\langle E, R \rangle$ matches another code hypergraph $\langle E', R' \rangle$ via a mapping $match : E \to E'$ such that $\forall e \in E,\ e$ matches $match(e)$ and $\forall \langle rname, r \rangle \in R,\ \exists \langle rname', r' \rangle \in R',\ rname = rname' \wedge r \subseteq r'$.

We say a code hypergraph g is more abstract than another code hypergraph g' if there exists a match from g to g'.

Given a code hypergraph, we can abstract it into a pattern by removing elements and attributes from the hypergraph. The result is ensured to match the original hypergraph. In this way, we turn the generalization problem into a problem of selecting elements and attributes in a hypergraph, where the selected elements, their selected attributes, and their relations form a new hypergraph as a pattern. Please note that our framework also allows to select relations, but in this paper we only consider the selections of elements and attributes and keep all relations among the selected elements. For example, in Figure 1, the red elements, their red attributes, and their relations form a new hypergraph that would match both code snippets. The elements with solid frame are the matched elements while the elements in the dashed frame are not matched.

After an element is matched, we can apply the modification operations to the elements. Our framework does not enforce a particular set of modification operations and treats modifications as uninterpreted atomic elements, denoted by set M. Furthermore, we assume the existence of two functions, *preIDs* and *preAttrs*. Function *preIDs*(m) denotes the element IDs that must be matched to make the modification m feasible. Function *preAttrs*(m, id) returns the attribute name on element id that must be matched to make the modification m feasible, where $id \in$ *preIDs*(m).

In our current approach we consider the following types of modifications.

- $insert(id, id', i)$: inserts an AST subtree rooted at id' as the i^{th} child of the element id.
- $insert\_str(id, str, i)$: inserts the text str as the i^{th} child of the element id.
- $replace(id, id')$: replaces an AST subtree rooted at id with another AST rooted at id'.
- $replace\_str(id, str)$: replaces an AST subtree rooted at id with the text str.
- $delete(id, id')$: deletes an AST subtree rooted at id from its parent id'.

For any modification m of the above modification type, *preIDs*(m) returns the set of element IDs appearing as the argument, e.g., *preIDs*$(insert(id, id', i)) = \{id, id'\}$;

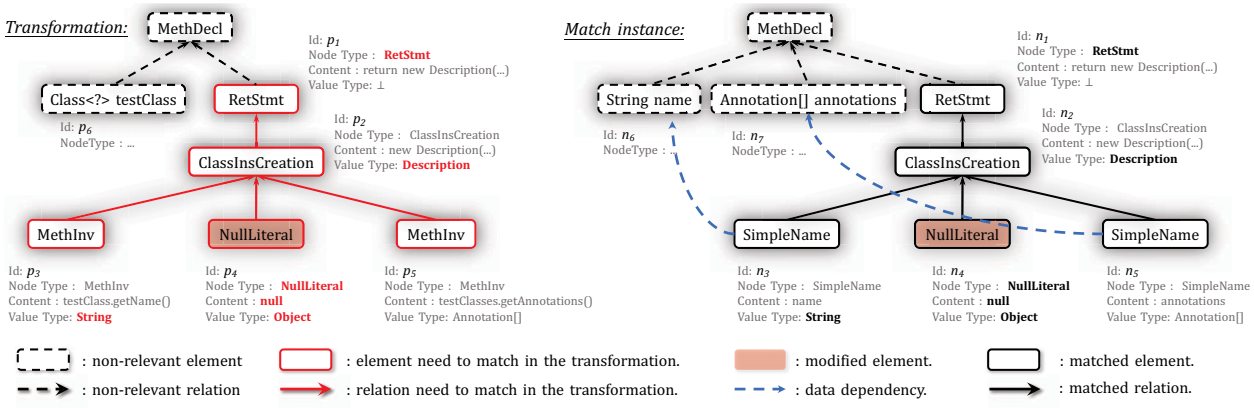

Fig. 1. Transformation instance inferred from the first case in Listing 1 and its matched instance. In the figure, we use "..." to represent omitted code content for simplicity. Besides, the ancestor relations are omitted as well.

preAttrs(m, id) always returns "AST node type" for any $id \in$ *preIDs*(m), as we need to keep consistency of the node type to ensure the AST is well-formed.

In our running example, the change can be captured by the modification $delete(p_4, p_2)$, while *preIDs* requires p_4 and p_2 to be matched, and *preAttrs* requires the matched elements have the same AST node types.

Finally, we give the definition of a transformation.

Definition 5 (Transformation)**.** A transformation is a pair $\langle g, \vec{m} \rangle$, where g is a code hypergraph and $\vec{m}$ is a sequence of modifications such that for any $m \in \vec{m}$, $id \in preIDs(m)$ and $attrName \in preAttrs(m, id)$, there exists an element $\langle id', attrs' \rangle$ in g such that $id = id'$ and $attrs'$ contains $attrName$.

Given a code hypergraph $g' = \langle E', R' \rangle$, a transformation $\langle g, \vec{m} \rangle$, and a match $match$ from g to g', applying the transformation generates a sequence of modifications $\vec{m}[id_0 \backslash id'_0, \ldots, id_n \backslash id'_n]$ where $\langle id_0, id'_0 \rangle, \ldots, \langle id_n, id'_n \rangle \in match$. In other words, the element IDs in original sequence of modifications are replaced by the matched element IDs. Then we apply the sequence of modifications to obtain the changed code.

V. The GenPat Approach

Based on the framework, we can now proceed to our approach. Given two code snippets before and after the change, our approach (1) extracts a code hypergraph from the snippet before the change, (2) extracts a sequence of modifications by comparing the two snippets, (3) infers a transformation by selecting elements and attributes from the hypergraph, and (4) matches and applies the transformation when given a new code snippet. In this section, we introduce how we implement the four components.

A. Extracting Hypergraphs

To extract the hypergraph, we need to extract the elements, their attributes, and their relations. In our current implementation we extract them as follows.

- *Elements.* We parse the code and extract the AST nodes.
- *AST node type, content, parent relation, ancestor relation.* We directly obtain them from the AST.
- *Value type.* We apply type analysis in Eclipse JDT to infer the value types of all expressions and parameters. For the rest of the elements (i.e., statements), we set its value type to $\perp$.
- *Data dependency.* We perform a simple flow-insensitive intra-procedural define-use analysis [38], [39] to extract data dependency relations. The variables are assumed to have no aliases during the analysis.

We assume the change occurs within a method, and consider only the code within the method body in our current implementation.

B. Extracting Modifications

In the current implementation, we employed the GumTree algorithm [40] to extract the modifications. Please note that the original GumTree algorithm also returns a "move" operation, which can be combined by a deletion and an insertion using our modification types.

C. Inferring Transformations

To infer a transformation, we select elements and attributes from the hypergraph. Please note that we do not select relations in this paper and consider all relations among the selected elements.

Element Selection. Since the definition of the transformation requires the elements in *preIDs* (i.e., elements corresponding to the modifications) to be included in the transformation, we first select these elements.

Next we add elements related to these elements as context. Here we follow the parent relation and the data dependency relation, both forwardly and backwardly, and include all elements that can be reached within k levels of the relations. In this study, we set $k = 1$ (the default configuration). In the future, we plan to conduct a more thorough investigation of different configurations.

For example, for the program in Figure 1, we first select p_2 and p_4 since they are modified. Then following the parent relation we include p_1, p_3, and p_5.

Attribute Selection. Same as elements, we first add the attributes required by *preAttrs* to form a well-formed transformation. In our example we would add the "AST node type" attributes of p_2 and p_4.

Then we select from other attributes in the selected elements. For the attributes of content and value type, we compute the frequency for a given attribute. That is, we collect the element content and value types from a large code corpus, and then compute the cross-project frequency for a given attribute. If the frequency is larger than a threshold, we select the attribute. In current implementation, we use the following formula to compute the frequency of each attribute. In the experiment, we set the threshold as 0.5%.

$$freq(attr) = \frac{|\{f | attr \text{ exists in file } f\}|}{|\{all\ files\ in\ dataset\}|}$$

Finally, we select the attribute of AST node type when the corresponding code element is a statement. This is to avoid inconsistent matching such as matching a statement with a variable. In the example, we select the node type of p_1.

D. Matching and Applying Transformations

Now suppose we have a transformation $t = \langle g, \vec{m} \rangle$, and we would like to apply the transformation to a code snippet sp. We first transform sp to a hypergraph g'=$\langle E', R' \rangle$, and then find a match $match$ from g to g' to perform the transformation. In order to find the match, we proceed with the following two steps.

1) Greedily matching each element e in E with all elements in E' by considering only the attributes.
2) Exhaustively checking all possible matching combinations generated in the first step with the relations between elements.

In our running example, by considering only the attributes, we can obtain the following mapping.

$$match(p_1) \in \{n_1\}, match(p_2) \in \{n_2\}, match(p_3) \in \{n_3\}$$
$$match(p_4) \in \{n_4\}, match(p_5) \in \{n_2, n_3 \cdots n_7\}$$

Then further considering the relations between elements, we can filter out the extra elements for p_5, forming a valid match.

$$match(p_1) \in \{n_1\}, match(p_2) \in \{n_2\}$$
$$match(p_3) \in \{n_3\}, match(p_4) \in \{n_4\}, match(p_5) \in \{n_5\}$$

Based on this match, we can generate the following transformation on the target snippet.

$$delete(n_2, n_4)$$

It is possible that multiple matches exist for a target code snippet. In some applications, we would like to find only one match. For example, in program repair, we usually assume that there is only one fault for a failed test. As a result, we need to rank the matches to find the best one. In our current approach we use the similarity between the AST node type and the content attributes to rank the matches.

$$node\_sim = \frac{|\{e \mid e \in E \wedge sameNodeType(e, match(e))\}|}{|E|}$$

$$text\_sim = \frac{1}{|E|} \Sigma_{e \in E} \frac{LCS(tokenize(e), tokenize(match(e)))}{|tokenize(e)|}$$

$$score = node\_sim + text\_sim$$

In the formulas, $sameNodeType(e, e')$ is used to judge whether element e is with the same node type as e', $tokenize(e)$ is the tokenized sequence for the content of element e, and $LCS(s_1, s_2)$ computes longest common token sequence between two token lists s_1 and s_2. Finally, we use the sum of the two similarities for $match$ ranking. Our intuition for the ranking heuristic is that if the buggy code has more common parts with the pattern code, more confidence can be gained to apply the transformation. As a result, in the formulas we consider both the node-type and token-sequence similarity information since they correspond to code syntax and semantics, respectively.

VI. Evaluation

To evaluate the effectiveness of GenPat, we choose two application scenarios—systematic editing [1], [15] (Section VI-A), and automatic program repair [17]–[19], [41]–[46] (Section VI-B).

A. Systematic Editing

1) Subjects: We employ two datasets in our evaluation, both collected in existing studies for evaluating systematic editing. The first one is the Sydit dataset collected by Meng et al. [15]. The second one is the dataset collected by Kreutzer et al. [47], which we call $\mathcal{C}3$.

Both datasets contain similar changes collected from commits in open-source projects, where all modifications within a method in a commit are considered as a change. The difference is how they measure similarity: Sydit uses ChangeDistiller [48] to extract changes for method pairs and requires they share at least one common syntactic edit and their content is at least 40% similar, and $\mathcal{C}3$ represents a code change as a list of edit operations and then clusters the changes by calculating distances over the feature vector of code changes.

The Sydit dataset consists of 56 pairs of similar changes. For each pair, one change is used for pattern extraction and the other one is used to test the extracted transformation. The $\mathcal{C}3$ dataset consists of 218,437 clusters of similar changes, where each cluster may have multiple changes. To unify the format of the two datasets, we randomly select a pair from each cluster of the $\mathcal{C}3$ dataset. We summarize the detailed information of the subjects in Table I.

2) Procedure: In this experiment we use Sydit as a baseline for comparison, which is a state-of-the-art technique that uses predefined rules for inferring program transformations. For each pair of code changes $(v_a \rightarrow v_{a'}, v_b \rightarrow v_{b'})$ in the dataset, we apply GenPat and Sydit to extract the transformation from $v_a \rightarrow v_{a'}$, and apply the transformation

TABLE I
EVALUATION DATASET FOR *Systematic Editing*.

Dataset Source	Project	#Pairs
SYDIT [15]	-	56
$C3$ [47]	junit	3,904
	cobertura	2,570
	jgrapht	2,490
	checkstyle	13,263
	ant	25,063
	fitlibrary	3,199
	drjava	31,393
	eclipsejdt	73,109
	eclipseswt	63,446
Total		218,493

to v_b. If the transformation can be applied and produces v_{b^*}, we compare v_{b^*} with $v_{b'}$. Since the complete dataset is large, in this experiment, the adapted code v_{b^*} is considered correct only if it is syntactically identical with the ground truth $v_{b'}$. We also sample a small proportion of the programs that are not syntactically identical to the ground truth and check its semantic equivalence manually. For each pair, we set the timeout as 1 minute.

We also need a code corpus for calculating the frequencies of attributes. For simplicity, we use the same corpus of patches as in the second program-repair experiment (Section VI-B1). Please note while this is not an ideal choice for systematic editing, as we will see later, we already achieved significantly better performance than the state-of-the-art technique.

TABLE II
GENPAT ON COMPLETE EXPERIMENT DATASET FOR *Systematic Editing*.

Projects	Total Pairs	#Adapted	#Syn-Eq
SYDIT	56	49 (87.5%)	27 (48.2%)
junit	3,904	1,088 (27.9%)	412 (10.6%)
cobertura	2,570	769 (29.9%)	305 (11.9%)
jgapht	2,490	547 (22.0%)	226 (9.1%)
checkstyle	13,263	5,918 (44.6%)	1,679 (12.7%)
ant	25,063	10,428 (41.6%)	4,398 (17.5%)
fitlibrary	3,199	922 (28.8%)	374 (11.7%)
drjava	31,393	11,391 (36.3%)	4,151 (13.2%)
eclipsejdt	73,109	32,037 (43.8%)	14,150 (19.4%)
eclipseswt	63,446	22,218 (35.0%)	9,206 (14.5%)
Total	218,493	85,367 (39.1%)	34,928 (16.0%)

NOTE, the ratio in the table denotes the portion of **Total Pairs**.
In the table, SYDIT represents the corresponding dataset.

3) Results: First, we evaluate GENPAT on the complete dataset as shown in Table I, and the experimental results are listed in Table II. In the table, the second column shows the total number of cases for transformation in each project, and the last column (**#Syn-Eq**) denotes the number (ratio) that GENPAT makes a syntactically identical adaptation among all the test cases. We also report the number of cases that GENPAT can successfully match the generated transformation to the target code shown in the third column (**#Adapted**). In total, GENPAT can successfully match and produce a result on 39.1% cases, while on 16.0% cases, the result is syntactically identical to the ground truth.

Then we further compare the result of GENPAT with state-of-the-art SYDIT on the same dataset. Note that we directly borrow the experimental result of SYDIT on the SYDIT dataset as reported in the original paper [15]. For the other projects, we successfully ran SYDIT on three projects (SYDIT reported errors on other projects, such as missing dependencies as it requires the projects compilable, encountering exceptions like `NullPointerException` and `IndexOutOfBoundException`, etc.). Therefore, we compare the results of GENPAT and SYDIT on the subset of our experiment dataset where they both apply. The details of the experimental results are listed in Table III. Please also note that SYDIT requires code change pairs for transformation extraction and application coming from the same versions (*ref.* Section VI-A2: v_a and v_b should come from the same project version). To satisfy this constraint, we select only those pairs in this experiment.

In Table III, for each technique, we report both the number (ratio) of cases adapted and the number (ratio) of cases that are transformed with syntactically identical editing (Columns 4-7). Particularly, since the result of SYDIT on the SYDIT dataset is based on the semantic equivalence between the adapted code and the ground truth. For a fair comparison, we also perform a manual inspection on the results of GENPAT on the SYDIT dataset. However, for the other projects, we compare their results based on syntactic equivalence. From the table we can see that GENPAT significantly outperforms SYDIT on the numbers of both adapted and (syntactically or semantically) correctly transformed cases. Overall, GENPAT produces 2.0x (1079/541) the adapted cases and 5.5x (570/103) the correctly transformed cases as SYDIT. If we consider the ratio of false positives, i.e., the cases where a transformation result is produced but not identical to the ground truth, GENPAT ((1079-570)/1079=47.2%) still significantly outperforms SYDIT ((541-103)/541=81.0%). Moreover, we found that GENPAT can still achieve a much better result (117 *vs* 64) even only on the cases where SYDIT can find a match (**SYDIT #Adapted**). To conclude, GENPAT significantly outperforms state-of-the-art SYDIT. The results suggest that using predefined rules may produce undesired transformations in many cases, which either cannot match, or incorrectly match the target code.

Considering the performance of the tools on different datasets, we can find that on the SYDIT dataset GENPAT only slightly outperforms SYDIT (49 *vs* 46 adapted and 40 *vs* 39 correctly transformed), while on the $C3$ dataset, GENPAT significantly outperforms SYDIT (1030 *vs* 495 adapted and 530 *vs* 64 correctly transformed). The reason is that the SYDIT dataset has stricter requirements on the similarity of the changes, and thus predefined rules already achieve good performance. On the other hand, $C3$ contains more diverse pairs such that better transformation inference is needed.

Please note that syntactical equivalence may not be a precise measurement as two changes may be syntactically different but

TABLE III
COMPARING GENPAT WITH SYDIT ON *Systematic Editing*.

Dataset	Projects	#Total Pairs	#Adapted		#Syn-Eq/Sem-Eq		#Syn-Eq of GENPAT in SYDIT #Adapted
			GENPAT	SYDIT	GENPAT	SYDIT	
SYDIT	-	56	49(87.5%)	46(82.1%)	-/40(71.4%)	-/39(69.6%)	-
$C3$	jgrapht	1,314	354(26.9%)	20(1.5%)	211(16.1%)	6(0.5%)	7
	junit	1,208	383(31.7%)	240(19.9%)	206(17.1%)	57(4.7%)	110
	cobertura	1,021	293(28.7%)	235(23.0%)	113(11.1%)	1(0.1%)	0
	Total	3,543	1,030(29.1%)	495(14.0%)	530(15.0%)	64(1.8%)	117
Overall		**3,599**	**1,079(30.0%)**	**541(15.0%)**	**570(15.8%)**	**103(2.9%)**	-

In the table, the ratio denotes the portion of **Total Pairs**, and we use "-" to denote missing data or directly omit "-".

semantically equivalent. To further understand how much of the syntactically different cases can be semantically equivalent, we perform a manual inspection on the transformed results. Since we do not have the detailed result of SYDIT on its own dataset, we randomly choose 20 cases in each project from $C3$ dataset, where the transformed code is not syntactically identical with the ground truth. As a result, we choose 60 cases for GENPAT and 54 cases for SYDIT (only 14 cases in project jgrapht, *cf.* Table III). The results are 11.7% (7/60) semantically correct cases for GENPAT, while 9.3% (5/54) semantically correct cases for SYDIT. The results suggest that the number of semantically equivalent cases would be slightly higher than the syntactically equivalent cases, and GENPAT would probably still significantly outperform SYDIT.

We further investigate the reasons why GENPAT do not produce syntactically or semantically equivalent cases. We randomly sampled 100 cases that are not equivalent to the ground truth. By manually analyzing these cases, we found the following four main reasons. (i) The dominating reason is that the dataset contains noise, where the given code change examples do not conform to the target code. In other words, we cannot obtain the desired code after applying the transformation inferred from the corresponding example. For example, the given code change example is updating a variable `runners` to `fRunners`, while the desired change is updating `fRunners` to `runners`. It is impossible to infer the latter transformation from the former example. In total, 64% incorrect cases are due to this reason. (ii) Some types of changes are not supported by our implementation. For example, some cases change the method signature, and some cases change two methods at the same time. Both situations are not supported by our current implementation. In total, 27% cases are due to this reason. (iii) Our current modification types do not allow some transformations. For example, the desired transformation should insert a statement after some other statements, while our modification operation only allows inserting at an absolute position, i.e., the ith child of the parent, rather than a relative position. In total, 3% cases are due to this reason. (iv) Our inference algorithm does not infer the correct transformation. For example, we may extract a too strong context that cannot match the target code. In total, 6% cases are due to this reason. Note that first two reasons are not directly related to our approach. The latter two reasons point out future directions to further develop the approach. In other words, with a better implementation our approach may show even better results.

B. Automated Program Repair

Our second experiment aims to explore the capability of repairing real-world bugs using GENPAT. In this experiment, we infer transformations from a large dataset of existing patches, and then apply these transformations to repair new bugs.

1) Subjects: We prepare two datasets, one of which is used as a training set for transformation extraction, while the other one is used as the dataset for program repair.

For the first dataset, we downloaded more than 2 million code change examples from all open-source Java projects on GitHub corresponding to all their commits from 2011 to 2016. In this process, we leverage a set of keywords for filtering, such as "fix", "repair", "bug", "issue", "problem", "error", etc. Following previous studies [17], [49], we further filter out code change examples involving more than five java files or six lines of source code since they may include benign changes. Moreover, we remove commits in the projects to repair (i.e., Defects4J projects) or their forked projects to avoid using their own patches. As a result, we build a training set consisting of more than 1 million bug-fixing examples, which will be used to extract transformations for program repair. Besides, this dataset is used as the big code corpus for attribute selection as well, where each changed file in each commit is treated as a code file.

For the second dataset, we employ a commonly used benchmark Defects4J [16] (v1.4), which consists of 395 real-world bugs from six open-source projects. We select 113 bugs from Defects4J for our experiment. The reason is that GENPAT is not designed to be a comprehensive and standalone repair tool and is not possible to fix many specific types of bugs (e.g., bugs requiring additional invocations of specific methods only from the current projects). To save experiment time, we filtered these bugs that cannot be fixed, and used the remaining 113 bugs. The details of the benchmark are listed in Table IV.

2) Procedure: As suggested by existing studies [2], [3], [50] that same bug fixes may recursively exist among the historical bug fixes. To avoid repetitive computation, we first

TABLE IV
EVALUATION BENCHMARK FOR *Program Repair*.

Project	Bugs	kLoC	Tests
JFreechart (Chart)	12	96	2,205
Closure compiler (Closure)	22	90	7,927
Apache commons-math (Math)	34	85	3,602
Apache commons-lang (Lang)	33	22	2,245
Joda-Time (Time)	6	28	4,130
Mockito (Mockito)	6	45	1,457
Total	113	366	21,566

In the table, column"**Bugs**" denotes the total number of bugs used in our evaluation, column"**kLoC**" denotes the number of thousands of lines of code, and column "**Tests**" denotes the total number of test cases for each project.

perform a transformation clustering, which collects the same transformations together to form a cluster. In this process, two transformations belong to the same cluster only if they can match each other, and they have the same modifications. As a result, after clustering 689,546 unique transformation clusters are left, which are finally employed for patch generation.

Following existing APR techniques [43]–[45], [51], we first leverage an existing fault localization framework [52] to obtain a ranked list of candidate faulty locations. Particularly, we employ the Ochiai [53] spectrum-based fault localization to compute suspicious scores. However, the fault localization result is at the statement level, while GENPAT matches a code snippet rather than a single line. Therefore, we further apply Method-Level Aggregation [54]–[57] to obtain a ranked list of faulty methods from statement-level results since it has been demonstrated to outperform direct method-level fault localization [54].

Given a faulty method, GENPAT locates a set of transformations whose attributes can be found in the faulty method. Then transformations will be ranked according to the size of corresponding clusters. Thereafter, GENPAT tries to apply each transformation to a given faulty method and generates patches. In the matching process, we discard matches that involve no elements in a faulty line in the method. In our experiment, we collect at most 10,000 compilable patches for each faulty method and then rank them with the ranking method introduced in the approach (Section V-D). Finally, we validate each candidate patch with the test suites and set a timeout of 5 hours to repair one bug. In this paper, following recent repair work [17]–[19], [44], [45], [58]–[60], we consider a patch as correct only if it is semantically equivalent to the developer's patch in Defects4J with manual check.

3) Results: In this section, we present the experimental result of GENPAT on repairing real-world bugs and compare it with state-of-the-art APR techniques that are recently published on SE conferences. The results are shown in Table V. In the table, we listed the number of bugs correctly fixed by each technique when considering top-k ($k \in \{1, 10\}$) plausible patches. We use "-" to represent those missed data. From the table we can observe that, surprisingly, although GENPAT is not designed as a comprehensive and standalone repair technique, it still successfully repairs 16 bugs when only considering top-1 plausible patch, even outperforming some recent approaches, such as SketchFix and JAID. When considering top-10 plausible patches, GENPAT can successfully repair 19 bugs. Moreover, among all the bugs fixed by GENPAT, 4 bugs have never been fixed by any existing technique as far as we know, such as the example shown in Listing 3. The results demonstrate that it is possible to repair real-world bugs by learning executable program transformations from historical bug fixes directly. Furthermore, the results also suggest that it would be promising to consider integrating GENPAT when designing advanced APR techniques to repair more bugs, which calls for future research in this direction.

TABLE V
COMPARING GENPAT WITH STATE-OF-THE-ART APR TECHNIQUES.

Conf.	Tech.	#Top-1 Pos.	#Top-10 Pos.
ISSTA'19	PraPR [60]	30	39
ISSTA'18	SimFix [18]	34	-
ICSE'18	SketchFix [61]	9	-
ICSE'18	CapGen [19]	21	22
ASE'17	JAID [59]	9	15
ICSE'17	ACS [45]	18	-
SANER'16	HD-Repair [17]	10	-
	GENPAT	16	19

In an investigation of the patches we found that the fixed bugs are often non-trivial and may not be easily fixed by approaches with a predefined search space. For example, Listing 4 shows the patch for Lang-21, which is successfully generated by applying the transformation extracted from the example in Listing 5. It is not easy to predefine a search space to include this constant replacement.

```
264   cal1.get(Calendar.MINUTE)==cal2.get(Calendar.MINUTE)&&
265 -cal1.get(Calendar.HOUR)==cal2.get(Calendar.HOUR)&&
265 +cal1.get(Calendar.HOUR_OF_DAY)==cal2.get(Calendar.
    HOUR_OF_DAY)&&
266   cal1.get(Calendar.YEAR)==cal2.get(Calendar.YEAR)&&
```

Listing 4. Patch of Lang-21.

```
Commit : github.com/Cbsoftware/PressureNet/commit/9d00742
Message: Fixing time display bugs, #113
Source : src.ca.cumulonimbus.barometernetwork.BarxxActivity
==========
2459 - if(start.get(Calendar.HOUR)==0&&end.get(Calendar.
    HOUR)==0){
2459 + if(start.get(Calendar.HOUR_OF_DAY)==0&&end.get(
    Calendar.HOUR_OF_DAY)==0){
```

Listing 5. Referenced history patch to fix Lang-21.

Meanwhile, though GENPAT is promising to repair real-world bugs, it still faces challenges. In our experiment, GENPAT generates plausible but incorrect patches for other 23 bugs among all 113 bugs. Compared with some state-of-the-art techniques, such as SimFix and CapGen, the repair precision of GENPAT is slightly lower. By analyzing those incorrect patches, we found that the reasons for its low precision are mainly threefold. First, though we have already preprocessed the training dataset for transformation extraction,

there still exist code changes that are not relevant to bug fixes, which may produce incorrect patches. Second, the inferred transformation is too general and can be applied frequently, such as inserting a `return` statement in an `if` body. Since GENPAT only expands one level dependency relation, the generated transformation can be applied wherever there is an `if` statement, and can easily introduce incorrect patches. Third, GENPAT is not designed to be a standalone repair tool and thus does not include the patch-correctness checking mechanisms that mature tools use. In the future, recent advanced patch-correctness checking techniques [62], [63] can also be further integrated with GENPAT to mitigate this issue.

VII. THREATS TO VALIDITY

In this section, we discuss the threats to validity of GENPAT.

First, the external threats to the validity fall into the data collection in our evaluation. We employed a subset of the $C3$ data set, i.e., we choose one pair of similar code changes from each cluster for the experiment, which may cause data selection bias. However, to mitigate this threat, we employed all 218,441 clusters in the data set shown in Table I with a random sample, which leaves us 218,441 pairs of examples. We believe that this big dataset can alleviate the threats. On the other hand, since the dataset is constructed automatically by previous research, which may involve noises as discussed in the previous section. As a consequence, we employed the manually-constructed dataset [15] as well in our evaluation, which can mitigate this issue to some extent.

Second, the internal threats to validity are related to the implementation of GENPAT. To ensure the correctness of its implementation, two authors of the paper collaborate with code review to make sure all functions are properly implemented. However, it is still possible to unintentionally get some implementation bugs involved. To further reduce this threat, we have also released both the source and test code of GENPAT, as well as the replication package, and invite other researchers to contribute to this promising direction.

VIII. CONCLUSION

In this paper, we propose a framework for transformation inference from a single example by representing code as a hypergraph, which allows fine-grained generalization of transformations with big code. Based on this framework, we further propose a transformation inference algorithm and implement it in a tool called GENPAT. Finally, we evaluated the effectiveness of GENPAT in two distinct application scenarios, i.e., systematic editing and automatic program repair. The experimental results show that GENPAT significantly outperforms the state-of-the-art SYDIT with up to 5.5x correctly transformed cases in the first application. Additionally, although not designed as a comprehensive and standalone repair technique, GENPAT already shows potentialities in automatic program repair – it successfully fixed 19 bugs in the Defects4J benchmark, 4 of which have never been repaired by any existing technique. In all, the evaluation results suggest that GENPAT is effective and potentially can be adopted in many different applications. On the other hand, in the current implementation, we do not consider the context information while computing the attribute frequencies, which potentially can further improve the quality of the inferred transformations. Also, there are also other attributes and relations besides those considered in our current implementation, such as the node-position attributes in AST or control-dependency relations, both of which may impact the quality of inferred program transformations. We leave a more thorough investigation to these variations to our future study.

ACKNOWLEDGMENT

This work was partially supported by the National Key Research and Development Program of China under Grant No.2017YFB1001803, National Natural Science Foundation of China under Grant Nos. 61672045 and 61529201, and National Science Foundation under Grant Nos. CCF-1566589 and CCF-1763906, and Amazon. Special thanks should go to Xia Li (UT Dallas) who shared the big code base with us, making it possible to conduct our large-scale evaluation.

REFERENCES

[1] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 309–319.
[2] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *FSE*, 2006, pp. 35–45.
[3] Q. Gao, H. Zhang, J. Wang, and Y. Xiong, "Fixing recurring crash bugs via analyzing Q&A sites," in *ASE*, 2015, pp. 307–318.
[4] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," ser. ICSE, 2010, pp. 315–324.
[5] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 53:1–53:11.
[6] B. Ray, C. Wiley, and M. Kim, "Repertoire: A cross-system porting analysis tool for forked software projects," in *FSE*. New York, NY, USA: ACM, 2012, pp. 8:1–8:4.
[7] J. Li, C. Wang, Y. Xiong, and Z. Hu, "SWIN: towards type-safe java program adaptation between apis," in *PEPM*, 2015, pp. 91–102.
[8] C. Wang, J. Jiang, J. Li, Y. Xiong, X. Luo, L. Zhang, and Z. Hu, "Transforming programs between apis with many-to-many mappings," in *ECOOP*, 2016, pp. 25:1–25:26.
[9] W. F. Opdyke, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.
[10] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
[11] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," ser. ICSE '13, 2013, pp. 502–511.
[12] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *ICSE*, 2017, pp. 404–415.
[13] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017, pp. 727–739.
[14] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 428–439.
[15] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. ACM, 2011, pp. 329–342.

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[17] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[18] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018.

[19] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[20] J. Andersen and J. L. Lawall, "Generic patch inference," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 337–346.

[21] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. Khoo, "Semantic patch inference," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2012, pp. 382–385.

[22] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *ESEC/FSE*. New York, NY, USA: ACM, 2019, pp. 613–624.

[23] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *ICSE*. IEEE Press, 2019, pp. 819–830.

[24] T. Molderez, R. Stevens, and C. De Roover, "Mining change histories for unknown systematic edits," in *MSR*, May 2017, pp. 248–256.

[25] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[26] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 118–129.

[27] T. Molderez and C. De Roover, "Search-based generalization and refinement of code templates," in *Search Based Software Engineering*. Cham: Springer International Publishing, 2016, pp. 192–208.

[28] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. ACM, 2008, pp. 247–260.

[29] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, pp. 205–214.

[30] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Science of Computer Programming*, vol. 72, no. 1, pp. 52 – 70, 2008.

[31] M. Erwig and D. Ren, "An update calculus for expressing type-safe program updates," *Science of Computer Programming*, vol. 67, no. 2, pp. 199 – 222, 2007.

[32] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190 – 210, 2006, special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 04).

[33] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Softw. Engg.*, pp. 176–205, 2015.

[34] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[35] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 4077–4087.

[36] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 112–122, 2007.

[37] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage." in *OSDI*, 2016, pp. 619–634.

[38] A. Hajnal and I. Forgacs, "A precise demand-driven definition-use chaining algorithm," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, March 2002, pp. 77–86.

[39] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Trans. Program. Lang. Syst.*, no. 2, pp. 175–204, 1994.

[40] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014, pp. 313–324.

[41] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: How far are we?" in *ISSTA*. New York, NY, USA: ACM, 2019, pp. 43–54.

[42] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 364–374.

[43] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," ser. ASE, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155644

[44] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *ASE*. IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155643

[45] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE*, 2017.

[46] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Science China Information Sciences*, vol. 62, p. 200102, Sep 2019.

[47] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. ACM, 2016, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901749

[48] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, pp. 725–743, Nov 2007.

[49] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, 2018, pp. 832–837.

[50] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 180–190.

[51] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, 2017.

[52] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," ser. ICSE '17, 2017, pp. 609–620.

[53] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," ser. PRDC. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46.

[54] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," ser. ISSTA, New York, NY, USA, 2017, pp. 273–283.

[55] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[56] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *ISSTA*. New York, NY, USA: ACM, 2019, pp. 169–180.

[57] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *ESEC/FSE*. New York, NY, USA: ACM, 2019, pp. 223–234.

[58] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the Defects4J dataset," *Empirical Software Engineering*, pp. 1–29, 2016.

[59] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE*, 2017.

[60] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*. New York, NY, USA: ACM, 2019, pp. 19–30.

[61] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *ICSE*, 2018.

[62] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ICSE*, 2018.

[63] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016.