Automated Patch Extraction via Syntax- and Semantics-Aware Delta Debugging on Source Code Changes

Masatomo Hashimoto
Chiba Institute of Technology
Narashino, Chiba, Japan
m.hashimoto@stair.center

Akira Mori
National Institute of Advanced Industrial Science and Technology
Ikeda, Osaka, Japan
a-mori@aist.go.jp

Tomonori Izumida
IIJ Innovation Institute
Chiyoda-ku, Tokyo, Japan
tizmd@iij.ad.jp

ABSTRACT

Delta debugging (DD) is an approach to automating debugging activities based on systematic testing. DD algorithms find the cause of a regression of a program by minimizing changes between a working version and a faulty version of the program. However, it is still an open problem to minimize a huge set of changes while avoiding any invalid subsets that do not result in testable programs, especially in case that no software configuration management system is available. In this paper, we propose a rule-based approach to syntactic and semantic decomposition of changes into independent components to facilitate DD on source code changes, and hence to extract patches automatically. For analyzing changes, we make use of tree differencing on abstract syntax trees instead of common differencing on plain texts. We have developed an experimental implementation for Java programs and applied it to 194 bug fixes from Defects4J and 8 real-life regression bugs from 6 open source Java projects. Compared to a DD tool based on plain text differencing, it extracted patches whose size is reduced by 50% at the cost of 5% more test executions for the former dataset and by 73% at the cost of 40% more test executions for the latter, both on average.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software reverse engineering*; *Software evolution*;

KEYWORDS

software regression, delta debugging, tree differencing

ACM Reference Format:
Masatomo Hashimoto, Akira Mori, and Tomonori Izumida. 2018. Automated Patch Extraction via Syntax- and Semantics-Aware Delta Debugging on Source Code Changes. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236047

1 INTRODUCTION

Debugging is still a demanding manual task that relies on experience and intuition in general. There have been a number of studies on automated debugging [4, 9, 13, 17, 24, 25, 31, 32, 34, 37–39, 41, 42, 44, 47, 48, 50, 52, 53, 55, 56, 63, 65, 69, 71, 72] focusing on activities such as identifying program statement(s) responsible for the failure, understanding the root cause of the failure, and determining how to modify the code to remove the root cause [45].

Delta debugging (DD) pioneered by Zeller [67, 69] is an approach to automating the debugging activities based on systematic testing. It resolves regression causes automatically and effectively. DD algorithms find the cause of a regression of a program by minimizing changes between a working version and a faulty version of the program. Zeller demonstrated DD for narrowing down 178,000 changed GDB lines to a single failure-inducing change relying on line-by-line differencing [67]. DD is distinguished from the prior approaches by its ability to handle interference, inconsistency, granularity, and non-monotony [67]; a combination of several individual changes may be responsible for a failure, while each of the changes is not (interference); there may also be combinations of changes that do not result in a testable program (inconsistency); a single logical change may affect several hundred or even thousand lines of code, but only a few lines may be responsible for the failure (granularity); a change set may not fail even if it contains a change that causes a failure (non-monotony).

However, it is still an open problem to minimize a huge number of changes while coping with inconsistency in case that no software configuration management system is available. Suppose that we naively perform DD on a huge number of hunks obtained by comparing program versions with extensive modificaton gaps in between by using an ordinary line-by-line diff tool. Then due to dependencies between the hunks, we would frequently fail to build the variants derived from the original program by applying the possible subsets of the hunks.

Zeller suggested, in his pioneering work [67], using syntactic or even semantic criteria in order to group changes based on the affected entities such as statements, functions, and modules. The basic idea was to group changes applied to a program by the following steps: construct a couple of program dependence graphs (PDGs) [19, 36] for a working version and for a faulty version of the program; for each change and each PDG, construct the forward static slice [5, 64] from the nodes affected by the change; group changes by the common nodes contained in their respective slices.

Unfortunately, however, static slices can be overly conservative and therefore tend to grow quickly [6, 68]. Thus, grouping changes based on their relevant static slices would result in a single large group in the worst case. On the other hand, dynamic slices [1, 35] can be precise since they are computed for specific program executions. However, we cannot rely on them to group changes since they can tell us nothing about the dependencies between portions that are not executed while producing all possible executions is infeasible in practice.

In order to overcome the problem, we propose a rule-based approach to syntactic and semantic decomposition of changes into independent components to facilitate DD for minimizing source code changes, and hence for extracting patches automatically. Given two versions V_1 and V_2 of a program, we model them as abstract syntax trees (ASTs) instead of plain text documents. Then the set of changes between the versions is decomposed through the following steps:

(1) comparing V_1 with V_2 by using an AST differencing algorithm,
(2) generating *AST delta* consisting of atomic *tree hunks* that operates on AST nodes based on the result of the AST differencing, and
(3) grouping tree hunks syntactically and semantically based on a set of rules that describe dependencies between tree hunks by exploiting syntactic and semantic information embedded in the ASTs.

Once changes are decomposed into individual components, we can apply DD algorithms on the components. Furthermore, we accelerate the DD process by way of *staging*, which is achieved by exploiting dependencies between tree hunks and the hierarchical structure of AST delta in a similar way to hierarchical delta debugging (HDD) [43].

We have developed an experimental implementation of a DD system for Java programs based on the proposed method and applied it to 194 bug fixes from Defects4J [33] dataset and 8 real-life regression bugs from 6 open source Java projects.

The remainder of the paper is organized as follows. Section 2 reviews an AST differencing method that we employ. Then, a method of generating decomposable AST delta is presented in Section 3. After an overview of syntactic and semantic decomposition of AST deltas is given in Section 4, Section 5 briefly explains staged DD. Section 6 details experiments conducted for evaluating our method of patch extraction. After related work and threats to validity are reviewed in Sections 7 and 8, Section 9 concludes the paper.

2 AST DIFFERENCING

Figure 1 illustrates an overview of our method of AST differencing. First, ASTs are obtained by parsing the source files *Prog1* and *Prog2*. Then the tree differencing engine compares the ASTs to generate an *AST delta* or *delta* for short. Finally, the delta is converted into *XML delta description (XDD)* format. The delta can be applied to *Prog1* to reproduce *Prog2*. Moreover, an arbitrary part of the delta can produce an intermediate version between *Prog1* and *Prog2*, which will be explained later.

Based on a structural change analysis tool, Diff/TS [26], we have designed and implemented the delta generation mechanism. We begin with reviewing Diff/TS's way of representing a difference between ASTs, that is, an *edit sequence*.

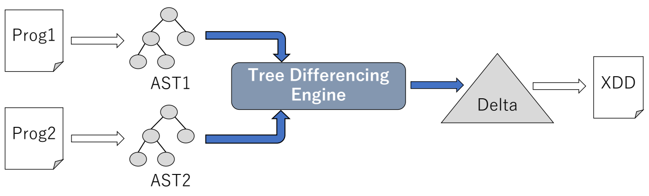


Figure 1: AST differencing.

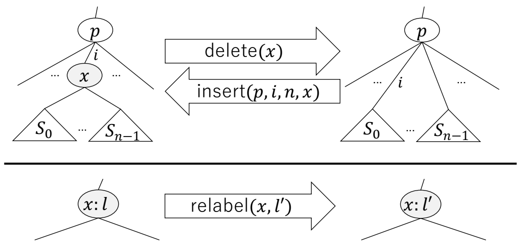


Figure 2: Tree edit operations.

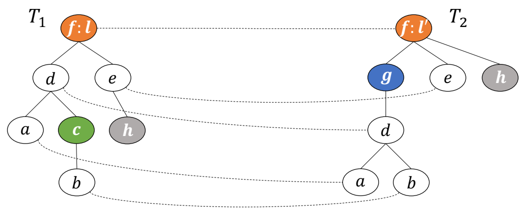


Figure 3: A tree mapping.

2.1 Edit Sequence

An AST is naturally modeled by a rooted labeled ordered tree. In the remainder of the paper, we mean a rooted labeled ordered tree by a "tree" unless otherwise specified. For trees T_1 and T_2, we calculate an *edit sequence* that transforms T_1 into T_2. An edit sequence consists of four kinds of *edit operations*: delete, insert, relabel, and move, all of which operate on AST nodes. The first three operations are illustrated in Figure 2. By delete(x), x is removed and the children of x become the i-th, . . ., and $(i+n-1)$-th children of p respectively. By insert(p, i, n, x), the i-th,. . ., and $(i + n - 1)$-th children of p are removed from p to become the children of x, and x becomes the i-th child of p. By relabel(x, l'), the label of x is changed into l', where we denote that x has a label l by "$x : l$". The last operation move is the composition of delete and insert; move(p, i, n, x) means delete(x) then insert(p, i, n, x).

2.2 Tree Mapping

Let T_1 and T_2 be trees, and s be an edit sequence that transforms T_1 into T_2. By $\mathcal{N}(T)$ we denote the set of nodes contained in T. Then by removing deleted, inserted, or moved nodes by s, we obtain a *tree mapping* $M \subseteq \mathcal{N}(T_1) \times \mathcal{N}(T_2)$ which satisfies the following conditions:

- $v_1 = w_1$ iff $v_2 = w_2$,
- v_1 is an ancestor of w_1 iff v_2 is an ancestor of w_2, and
- v_1 is to the left of w_1 iff v_2 is to the left of w_2

for any $(v_1, v_2) \in M$ and $(w_1, w_2) \in M$. In other words, a tree mapping is a one-to-one partial mapping that preserves ancestor-descendant relationships and sibling relationships. Figure 3 depicts a tree mapping, where dashed lines designate the elements of the mapping. Note that the untouched or the relabeled nodes are contained in the mapping, while others are not; c is deleted, g is inserted, and h is moved. We call nodes in a tree mapping *mapped nodes*.

2.3 Optimizing Edit Sequences

For each pair of trees T_1 and T_2, there exist trivial edit sequences, which correspond to the empty tree mapping that removes all nodes in T_1 and then adds all nodes in T_2. To quantitatively evaluate an edit sequence, we compute the *edit cost* of it. Given a cost value for each edit operation, the edit cost of an edit sequence is obtained as the sum of the costs of the operations contained in the sequence. If we exclude the move operations, there are several tree differencing algorithms that find optimal edit sequences in terms of edit cost. However, such algorithms are quasi-quadratic at best in time complexity and quadratic in space complexity [7], hence are inefficient for large trees that contain tens of thousands of nodes such as ASTs derived from thousands of source lines of code. Note that finding optimal edit sequences containing move operations is known to be NP-hard [40].

While Diff/TS is based on an optimal tree differencing algorithm, it has achieved good processing speed and precision needed for investigating large-scale software projects by employing tree decomposition, subtree hash encoding, and post-processing mechanisms based on elaborated heuristics for source code. The core algorithm of Diff/TS accepts a couple of trees as input and decomposes the entire comparison task into manageable subtree comparisons by means of hash value matching and tree-flattening [26]. It applies an optimal algorithm [70] for subtree comparisons. An edit sequence is computed by integrating the results of the subtree comparisons, and then post-processed to yield the final edit sequence. The post-processing step performs generate-and-validate move generation and revises the resulting edit sequence based on heuristics specific to source code. For example, we can correct move operations and/or relabel operations based on def-use relationships [26].

2.4 Tree Hunks

The post-processing step of Diff/TS also aggregates edit operations of the same kind that operate on the connected nodes in the AST. We call an aggregated set of edit operations a *tree hunk*, or simply a *hunk* if there is no confusion. Figure 4 illustrates three kinds of hunks. By $\overline{\text{relabel}}$, $\overline{\text{delete}}$, $\overline{\text{insert}}$, and $\overline{\text{move}}$, we denote a *relabel hunk*, a *delete hunk*, an *insert hunk*, and a *move hunk*, respectively. Note that a relabel hunk is always a singleton since we perform relabeling one by one. Application of hunks other than relabel hunk takes extra arguments in addition to those of the original edit operations.

- $\overline{\text{delete}}(x, E)$ where E denotes a set of *excluded nodes*,
- $\overline{\text{insert}}(p, i, n, x, I)$ where I denotes a set of nodes with their positions, or *insertion points*, and
- $\overline{\text{move}}(p, i, n, x, E, I)$ where E and I denote excluded nodes and insertion points, respectively.

Then application of the hunks in Figure 4 are represented as follows:

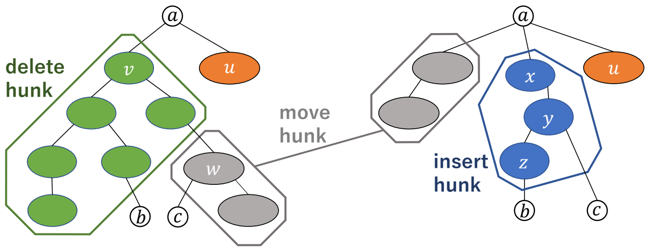


Figure 4: Tree hunks.

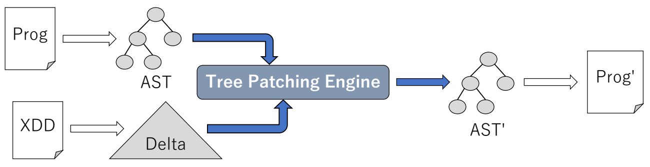


Figure 5: AST patching.

- $\overline{\text{delete}}(v, \{b\})$,
- $\overline{\text{insert}}(a, 1, 2, x, \{(y, 1), (z, 0)\})$, and
- $\overline{\text{move}}(a, 0, 0, w, \{c\}, \emptyset)$.

2.5 AST Delta

An AST delta is a set of augmented hunks; hunks are augmented with several annotations that make arbitrary combination of them applicable to ASTs to produce valid patched versions of the ASTs. Instead of detailing such annotations, we will explain our design of the delta application in the next sections.

3 AST PATCHING

Figure 5 depicts an overview of our method of AST patching. A delta obtained from an XDD file is applied to *AST* obtained from *Prog* to produce a patched version *AST′* of *AST*. Then we obtain *Prog′* by unparsing *AST′*. The definition of delta application above assumes a rough ordering of hunk application. Without such assumption, the definition would have been much more complicated since it has to take arbitrary partial delta application into account. In order to apply a hunk sequence s to a tree T, we can easily reorder s as follows:

$$s = \overbrace{e_0 \cdots e_i}^{\overline{\text{relabel}}} \overbrace{e_{i+1} \cdots e_j}^{\overline{\text{delete}}} \overbrace{e_{j+1} \cdots e_k}^{\overline{\text{insert}}},$$

where each move hunk is decomposed into a delete hunk and an insert hunk. Figure 6 illustrates application of a hunk sequence composed of the hunks shown in Figure 4. Firstly, a $\overline{\text{relabel}}$ is applied although not illustrated in the figure (Figure 6-1). Secondly, $\overline{\text{delete}}$ and deletion part of $\overline{\text{move}}$ are applied (Figure 6-2). Then $\overline{\text{insert}}$ is applied (Figure 6-3). Finally, insertion part of $\overline{\text{move}}$ is applied (Figure 6-4).

However, if we apply a subsequence of s to T, nodes to be inserted may not be inserted into T and nodes to be deleted may be left in T. To cope with such situations, we should be able to insert a node when its insertion target is not present yet and also be able to insert a node even when its insertion target has a node sitting at the

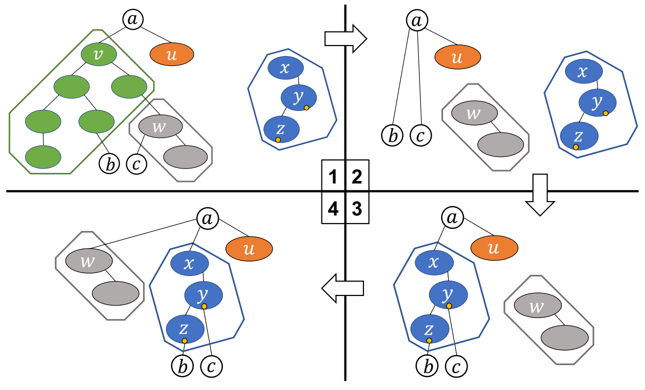


Figure 6: Delta application.

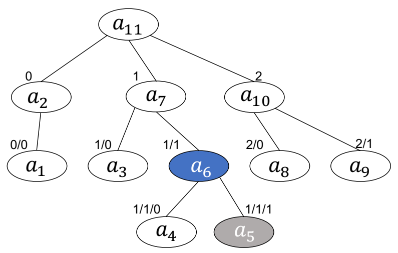


Figure 7: Node access paths.

position where the node is to be inserted. To remedy the both, we introduce notions of *node access path* and *virtual insertion target*.

3.1 Node Access Path

We locate a node in a tree by a *path* from the root. A path is essentially a list of ancestors represented by an ordered list of positions. Figure 7 shows examples of paths. Suppose that a_7 is a mapped node and that a_6 and a_5 are to be inserted and a_6 is the insertion target of a_5. If a_5 is inserted before a_6 is inserted, a_5's insertion target will be a_7, which is the latest present node in the path of a_6.

3.2 Virtual Insertion Target

Since we allow partial delta application, an insertion target a of an insert hunk rooted at x may have another node v (to be deleted) at the position where x is to be inserted, as seen in Figure 6-1. If the insertion hunk has no insertion points, x will be safely inserted next to v. Otherwise, we find a *virtual insertion target*, which is the latest common ancestor of the nodes to be inserted into the insertion points. For example, in Figure 8-1, b and c are inserted into insertion points at z and y in the insertion hunk, respectively. In this case, the virtual insertion target is v, which is the latest common ancestor of b and c. Inserting the insert hunk and the insertion part of the move of w yields an intermediate tree as shown in Figure 8-2. Note that performing deletion including the deletion part of the move of w on the intermediate tree yields the same result as that in Figure 6-4.

While the above works, there exists exceptional cases as shown in Figure 9, where the original program fragment and its AST are

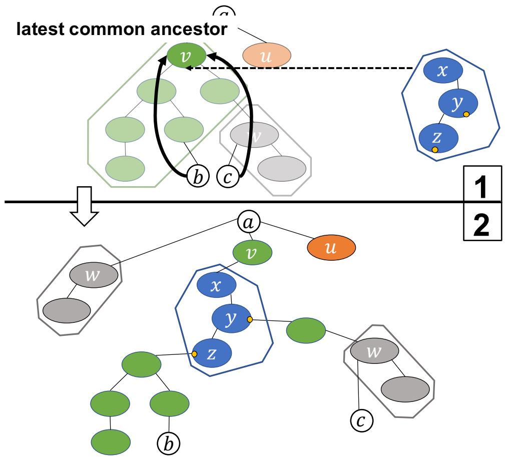
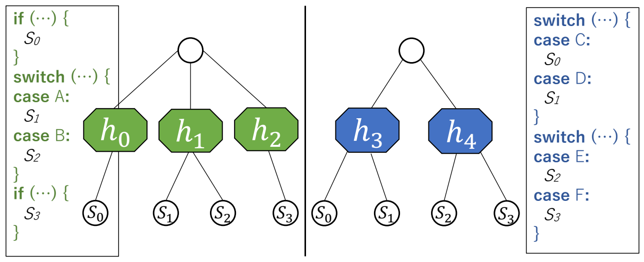


Figure 8: A virtual insertion target.



```
if (⋯) {
  S0
}
switch (⋯) {
case A:
  S1
case B:
  S2
}
if (⋯) {
  S3
}
```

```
switch (⋯) {
case C:
  S0
case D:
  S1
}
switch (⋯) {
case E:
  S2
case F:
  S3
}
```

Figure 9: An exceptional case.

on the left and the modified version of the program and its AST are on the right. Inserting a hunk h_3 into a virtual insertion target (the root), which is the latest common ancestor of s_0 and s_1, yields the tree shown at the upper right of Figure 10. Then inserting h_4 yields an intermediate tree shown at the center of Figure 10 since s_2 is already taken by h_3. If we delete h_1 later, the location of s_2 should be adjusted as shown in Figure 10. We have covered several such cases as they emerged. Now the tool can correctly generate patched programs for most cases.

4 SYNTACTIC AND SEMANTIC DECOMPOSITION OF DELTA

Our aim is to decompose an AST delta into independent *delta components*, each of which is composed of tree hunks, so that any subset of the set of components does not cause build failures when it is applied to a program. To achieve this, we introduce three kinds of logical rules for coupling tree hunks: syntactic rules, refactoring-based rules, and change-based rules. Syntactic rules are defined so that any delta component does not violate syntactic constraints. Refactoring-based rules are derived from the definitions of refactoring patterns. Change-based rules are elaborated based on dependencies between tree hunks to avoid build failures.

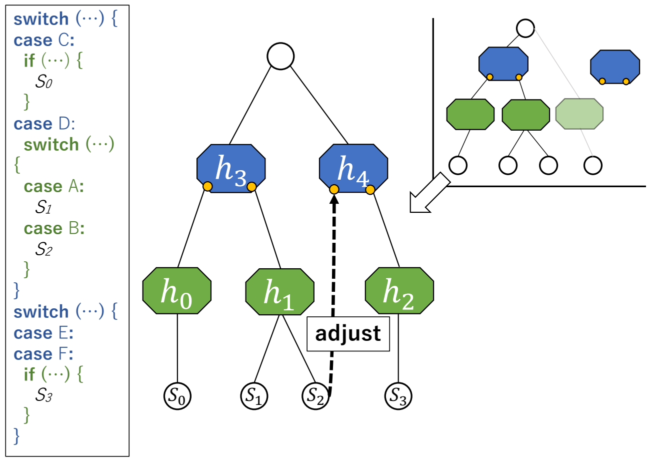


Figure 10: Node location adjustment.

4.1 Coupling by Syntactic Constraints

For an AST node, its children's labels and/or the number of the children are constrained by the syntax of the underlying programming language. For example, a node labeled with a constant value would have no children, while another node labeled with "`if`" would have two or three children for a condition expression and a "`then`" part, and possibly "`else`" part.

If a hunk contained in a delta incompletely changes a part of an AST and hence violates syntactic rules, the requisite hunks to complete the change should be included in the delta. For example, a hunk that deletes the condition of a "`if`" statement should be coupled with another hunk that inserts a condition into the statement.

4.2 Coupling by Refactoring Patterns

Hunks related to a refactoring pattern should be coupled together. For example, *Add Parameter* pattern couples hunks for adding parameters to a method with those for adding arguments of the corresponding method invocations. In order to find refactoring patterns, we employ a framework for analyzing fine-grained source code change patterns [28]. Since the framework requires a database query for specifying a change pattern, we translated descriptions of 38 refactoring patterns cataloged by Fowler [22] into database queries.

Let us consider the case of *Remove Parameter* pattern for instance. We regard a parameter p as removed when the following fine-grained conditions hold:

(1) m and m' are method declarations,
(2) q is a parameter list (corresponding to $(\cdots)$),
(3) q' is a parameter list,
(4) the parent nodes of q and q' are m and m', respectively,
(5) the version v' of m' is the immediate successor of the version v of m,
(6) q' is a new version of q, that is, q is mapped to q',
(7) m' is a new version of m, that is, m is mapped to m', and
(8) p is deleted from q.

It is not difficult to check these conditions based on the information reported by Diff/TS.

4.3 Coupling by Other Changes

This kind of couplings are further divided into *directed* and *undirected* couplings. Suppose that a hunk h_0 depends on another h_1, while h_1 does not depend on h_0. Then h_0 is coupled with h_1 in a directed way. For example, insertion of an instance creation of a new class requires insertion of the class if there is any, while the class insertion does not. If h_0 and h_1 depend on each other, they are coupled in an undirected way. Informal descriptions of the directed coupling rules include the following:

DR1 inserting a statement requires inserting the enclosing method body,
DR2 inserting a use of a variable requires inserting an initialization of the variable before the use,
DR3 inserting an abstract method into an abstract class requires inserting a concrete overriding method into a concrete subclass of the abstract class,
DR4 deleting a method declaration requires deleting an invocation of the method,
DR5 deleting a single type import declaration requires deleting an occurrence of the type name, and
DR6 deleting a loop requires deleting a "`continue`" statement in the loop.

Informal descriptions of the undirected coupling rules include the following:

UR1 a field name changes together in the field declaration and in the field accesses,
UR2 return values of a method and the return type of the method change together,
UR3 "`throws`" and "`throw`" statements are inserted in a method together,
UR4 a parameter and a corresponding argument are inserted together into a method and into an invocation of the method,
UR5 "`throws`" of a method and "`throws`" of an overriding method are deleted together, and
UR6 deletion of an "`abstract`" modifier from a method and insertion of a body into the method are simultaneous.

Note that **DR2** requires control flow graphs, while **DR4** and **UR4** require call graphs.

Figure 11 shows a decomposed delta, namely grouped tree hunks, derived from bug #3517 of jEdit [61], where red circles represent hunks, blue rectangles represent, refactoring patterns, green rectangles represent other change patterns, lines between red circles represent syntactic couplings, lines from blue rectangles represent refactoring-based couplings, and lines from green rectangles represent change-based couplings.

5 STAGED DELTA DEBUGGING

5.1 Delta Debugging

Delta debugging (DD) is an approach to automated debugging based on systematic testing [67, 69]. We employ *ddmin* algorithm [69] to minimize failure-inducing or bug-fixing changes of a program. Unless otherwise noted, we mean *ddmin* by DD.

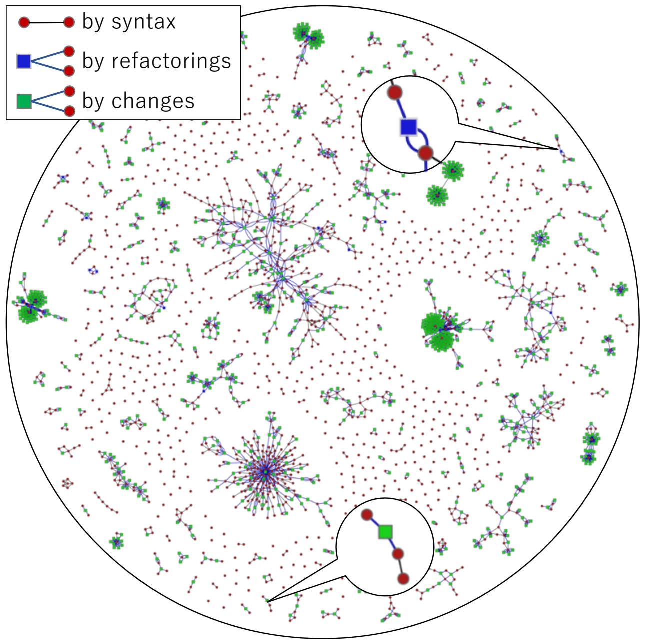


Figure 11: Grouped AST delta hunks.

Suppose that we have a program and a test for it. DD starts a minimization process with a set of changes between two versions V_{good} and V_{bad} of the program, where the test passes for V_{good} and fails for V_{bad}. The test is required to distinguish three test results: PASS, FAIL, and UNRESOLVED. DD divides the change set into two subsets in order to produce a virtual intermediate version of the program based on one of the subsets, and apply the test to the intermediate version. DD repeats bisecting and testing until it obtains *1-minimal* failure-inducing changes, which means that the changes no longer cause test failure if any of the changes is removed.

In this process, we apply coupling rules explained in Section 4 to avoid unwanted build failures. For example, if a change set is composed of method name changes in invocations, we can augment the set with method name changes in the corresponding method declarations to avoid "symbol not found" errors.

5.2 Staged Delta Debugging

By virtue of AST differencing that Diff/TS offers, we can effectively apply hierarchical delta debugging (HDD) [43] to our method. Like the original HDD, we intend to accelerate the DD step in our method by exploiting the hierarchical structure of an AST delta and dependencies between delta components. Suppose that we take an AST delta as an input of a DD algorithm. We divide the whole DD step into several *stages*. At the first stage, delta components are grouped together based on the files to which they belong. Note that a delta component may belong to multiple files since it is composed of multiple tree hunks possibly coupled across file boundaries. If a delta component that belongs to a set of files depends on another that belongs to another set of files, the dependency is naturally extended to the dependency between files. Then the file level DD, namely DD on the groups of the delta components, is performed. At the following stages, the method level DD and the statement level DD are performed similarly. At the final stage, the finest node level DD, namely DD on the original delta components, is performed.

It should be noted that we would have hierarchies consisting only of files and text hunks [3] if we based our method on a line-by-line text patch.

6 EXPERIMENTS

In order to evaluate our method, we have developed a couple of prototype systems by overloading `_test` method of `DD.py` [66] created by Zeller. DDJ is an implementation of our method for changes of Java programs, which was created based on Diff/TS and the query-based analysis framework used in several software engineering studies [27–29]. We prepared more than 500 database queries for specifying coupling rules. As mentioned in Section 4, some of the coupling rules require control flow graphs and call graphs. For this, we prepared several dozens of queries for implementing simple control flow analysis and simple call graph construction based on class hierarchy analysis (CHA) [16].

Note that DDJ also overloads `_resolve` method of `DD.py` in addition to `_test` method. The `_resolve` method takes care of resolving dependencies arising from a set c of components in two directions: by adding delta components to c and by removing delta components from c. DDJ resolves dependencies from c based on the directed couplings of tree hunks contained in c.

As a baseline system, we have also implemented programming-language-agnostic DD system, DDP, for changes of plain texts based on GNU diff. DDP decomposes a plain text patch into *patch components*, namely text hunks as opposed to DDJ decomposing an AST delta into delta components.

We conducted a couple of experiments that extract bug-inducing changes and bug-fixing patches from pairs of commits/revisions/versions. We evaluate the results by the number of build (test) executions, by *build success rate*, and by the size of the resulting patches.

In order to measure the size of an AST delta and that of a plain text patch uniformly, we convert them into token sequence differences. That is, an AST delta or a plain text patch is converted into a list of deleted or inserted token sequences (replacement of a token sequence s by s' is interpreted as deletion of s followed by insertion of s') obtained by tokenizing the original and the patched programs and then by applying a sequence matching algorithm to the tokenized programs. By tokenizeJ and tokenizeP, we denote conversions into token sequence differences, namely lists of deleted or inserted token sequence lists, from an AST delta and an plain text patch, respectively. We define the size of an AST delta or a text patch as the sum of the number of tokens occurring in the token sequence difference obtained by tokenizeJ or tokenizeP, respectively. We employed `javalang` [57] and `difflib` [21] for tokenizing Java programs and differencing token sequences, respectively.

6.1 Automated Isolation of Bugs for Defects4J

Defects4J, D4J for short, is a database of real-life bugs for reproducible studies in software engineering research [33]. At the time of writing, D4J (version 1.2.0) contains 395 bugs from 6 open source

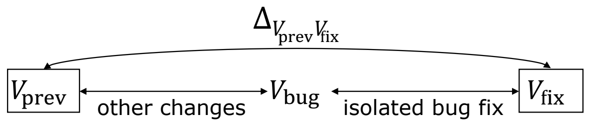


Figure 12: Isolating a bug fix.

Java projects: JFreeChart (Chart), Closure Compiler (Closure), Apache Commons Lang (Lang), Apache Commons Math (Math), Mockito (Mockito), and Joda-Time (Time). Each bug is accompanied by a comprehensive test suite that contains at least one failing test that triggers the bug.

A bug is stored in D4J as a pair of a faulty and a fixed source code versions, V_{bug} and V_{fix}, that differ only by the bug fix. For each bug, they first identified V_{fix}, and then isolated the fix from the source code difference $\Delta_{V_{\text{prev}}V_{\text{fix}}}$ between two consecutive versions V_{prev} and V_{fix}, where V_{prev} is the previous faulty version of V_{fix} in the repository. Note that $\Delta_{V_{\text{prev}}V_{\text{fix}}}$ may include irrelevant changes such as refactorings. A virtual intermediate version V_{bug} is synthesized in a way that the difference between V_{bug} and V_{fix} becomes equivalent to the isolated bug fix obtained by removing unrelated changes to the bug fix from $\Delta_{V_{\text{prev}}V_{\text{fix}}}$. Figure 12 illustrates the relationship between V_{prev}, V_{fix}, and V_{bug}.

6.1.1 Interpretation of the Original Test Results. For each bug, D4J provides pairs of triggering tests and their expected exceptions $(t_0, e_0), \ldots, (t_n, e_n)$ when they fail. The test script test$_{\text{DD}}$ for both DDJ and DDP checks whether the same set of tests as that of D4J's triggering tests fails on the patched source code, and also checks whether the thrown exceptions are consistent with those in D4J. For each bug in D4J, test$_{\text{DD}}$ outputs PASS when any of the triggering tests $T = \{t_0, \ldots, t_n\}$ succeeds, FAIL when one or more t_i in T produce failure with exceptions consistent with e_i, and UNRESOLVED otherwise.

Note that test$_{\text{DD}}$ cannot exactly match exceptions against those provided by D4J since exceptions may also change according to the source code changes and/or the execution environment. For example, a test of Math triggers bug 38 with an unhandled exception "`BOBYQAOptimizer$PathIsExploredException`" in 38b, namely V_{bug} of bug 38. However, changes between 38p, V_{prev} of bug 38, and 38f, V_{fix} of bug 38, contain *Move Class* refactoring that moves the exception into class "`BOBYQAOptimizer`", hence the test triggers bug 38 with another "`PathIsExploredException`" without class prefix "`BOBYQAOptimizer$`" in 38p. Other examples include several tests of Lang that trigger bugs with `AssertionFailedError` messages that contain the date and/or time of the test executions.

6.1.2 DD on Defects4J Dataset. By reversing the chronological order of V_{prev} and V_{fix}, we regard the D4J bugs as regressions. Then, we performed DD, with DDP and DDJ, on changes between V_{fix} and V_{prev} of D4J projects that do contain unrelated changes to the bug fixes. We dropped trivial cases where there was no need to isolate bug-fixing changes. As mentioned above, the bugs in D4J are derived from real-life bug-fix commits; some commits involve changes unrelated to bug fixes, and others do not. Note that we turned off staging of DDJ in order to avoid fruitless overhead since

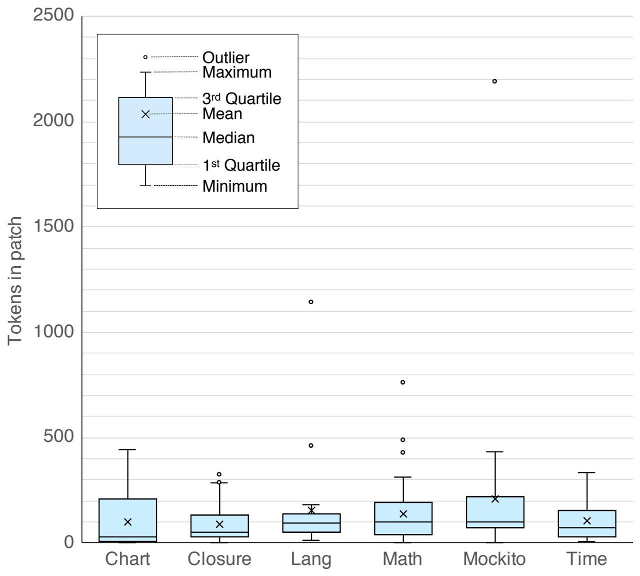


Figure 13: Amount of difference between V_{fix} and V_{prev}.

the differences between V_{fix} and V_{prev} of D4J projects are relatively small as shown in Figure 13.

The results of DDP and DDJ are summarized in Table 1. From 194 pairs of versions, DDP and DDJ extracted patches consisting of 15485 tokens and 9719 tokens in total, respectively. Thus, DDJ extracted 37% smaller patch (in tokens) than DDP did at the cost of 5% more build executions than DDP required. By virtue of syntax- and semantics-awareness, DDJ caused far less build failures than DDP did although DDJ decomposed the difference into 1.6 times as many components as DDP did (see C$_{\text{DDP}}$, C$_{\text{DDJ}}$, R$_{\text{DDP}}$, and R$_{\text{DDJ}}$).

6.1.3 Validity of the Resulting Patches. Although any of the resulting patches is valid in the sense that it causes the triggering tests to fail with consistent exceptions, we further examine whether it overlaps with the corresponding human extracted D4J patch. Table 2 (OP$_{\text{DDP}}$ and OP$_{\text{DDJ}}$) shows that half of the tokens in token differences D_{DDP}, namely pairs of deleted/inserted token sequence lists derived from DDP patches by tokenizeP, match those in D_{D4J} from D4J's manually extracted patches, while 61% of the tokens in D_{DDJ} derived from DDJ patches by tokenizeJ match those in D_{D4J}.

It should be noted that both DDP and DDJ extracted patches that do not overlap with D4J patches for 11 and 19 bugs, respectively (see NO$_{\text{DDP}}$ and NO$_{\text{DDJ}}$). The reasons behind this are the following.

- **Semantic equivalence of patches** The overlaps shown in Table 1 are the numbers of *syntactic* tokens. By manually inspecting the resulting patches, we found that 6 out of the 11 DDP patches and 13 out of the 19 DDJ patches are *semantically* equivalent to those of D4J.
- **Ambiguity of tests** Some triggering tests are ambiguous in that multiple independent subsets of the input delta or patch components cause the tests to fail with expected exceptions. Moreover, test$_{\text{DD}}$ cannot expect exact matching of the thrown exceptions as mentioned in Section 6.1.1.

Table 1: Minimizing Reverse Bug Fixing Changes for Defects4J

B: number of bugs, P: tokens in patch, OP: tokens in common with D4J patch / tokens in patch,
NO: number of patches overlapping with no D4J patch, C: number of patch (delta) components, R: build success rate

Project	B	$P_{V_{\text{fix}}V_{\text{prev}}}$	P_{D4J}	OP_{DDP}	OP_{DDJ}	NO_{DDP}	NO_{DDJ}	C_{DDP}	C_{DDJ}	R_{DDP}	R_{DDJ}
Chart	11	1127	753	403/878	556/748	0	0	88	71	0.930 (107/115)	1.000 (125/125)
Closure	69	6057	3075	1640/3449	1189/1907	7	10	260	373	0.779 (441/566)	0.991 (541/546)
Lang	22	3384	2278	1658/2993	754/1265	1	4	64	173	0.807 (155/192)	0.986 (272/276)
Math	53	7288	4035	2383/5119	2007/3762	3	4	203	446	0.844 (356/422)	0.953 (546/573)
Mockito	30	6275	1747	1297/2280	1073/1659	0	1	225	280	0.668 (308/461)	0.965 (304/315)
Time	9	973	632	467/766	332/378	0	0	37	53	0.950 (76/80)	1.000 (89/89)
Total	194	25104	12520	7848/**15485**	5911/**9719**	11	19	877	1396	**0.786** (1443/**1836**)	**0.976** (1877/**1924**)

Single output of DD algorithm The original DD.py, on which DDP and DDJ are based, looks for the first failing subset and hence reports only a single subset. We have added capability to obtain other subsets by rerunning DD on the complement of the resulting subset as long as it causes failure and/or by shuffling input sets in order to randomize bisection. In fact, by rerunning DDJ, we could extract other patches that syntactically overlap with D4J patches for all of the remaining 6 bugs, while DDP could do the same for the remaining 5 bugs by shuffling the inputs.

6.2 Automated Extraction of Regression Fixes

We collected 8 real-life regression bugs from 6 open source Java projects. ANTLR is a parser generator for reading, processing, executing, or translating structured text such as source code or binary files. We chose a regression ANTLR-1543 [46], "type mismatch between left and right labels in left-recursive rule". Apache Commons Math is a mathematics and statistics library addressing the most common problems that are not available in the Java programming language. We chose MATH-805 [15], "Percentile calculation is very slow when input data are constants". HSQLDB is an SQL relational database software. We chose HSQLDB-933 [2], "UPDATE statement with IN criterion recently broken" and HSQLDB-1454 [58], "Null Pointer in CASEWHEN and CASE... WHEN use case". HSQLDB-933 appeared also in experiments conducted by Artho [3]. jEdit is a text editor designed for programmers. We chose JEDIT-3517 [61], "NPE when scrolling to top after multiline delete (soft wrap)". jsoup is a Java library that provides an API for extracting and manipulating data from HTML files. We chose JSOUP-920 [8], "Regression: Wrong parsing for clippath" and JSOUP-926 [54], "Regression: CSS attribute selector". Logback is one of the Java logging frameworks. We chose LOGBACK-1183 [30], "Message formatting regression".

For each regression, we searched the log backward from a faulty version V_{bad} for a regression-free version V_{good}. With DDP and DDJ, we performed DD on changes from V_{bad} through V_{good} instead of from V_{good} through V_{bad}. We detect a regression-fixing change set instead of a failure-inducing change set by modifying the test scripts to invert the test results. The results are summarized in Table 2. For each regression, V_{bad} and V_{good} indicate a version number prefixed by 'v', a revision number prefixed by 'r', or a commit ID with no prefix.

The largest input change set was the one taken from MATH-805, while the smallest from JSOUP-926. Note that even the smallest input change set is much larger than that of each bug in the D4J dataset. The DD runs for minimizing MATH-805 and JSOUP-926 regression fixes are shown in Figures 14 and 15, respectively. Performing DDJ on MATH-805 resulted in a patch which is 7 times as small as that by DDP at the cost of 15 times as many test executions as DDP required. The three worst build success rates were set for MATH-805, HSQLDB-933, and HSQLDB-1454 by DDP, mainly

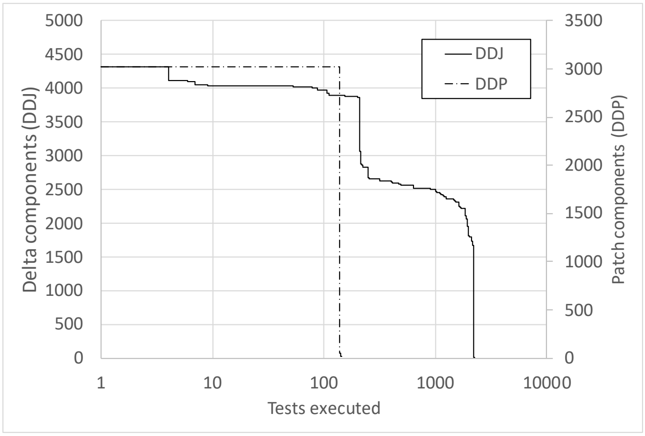


Figure 14: DD run for minimizing MATH-805 fix patch.

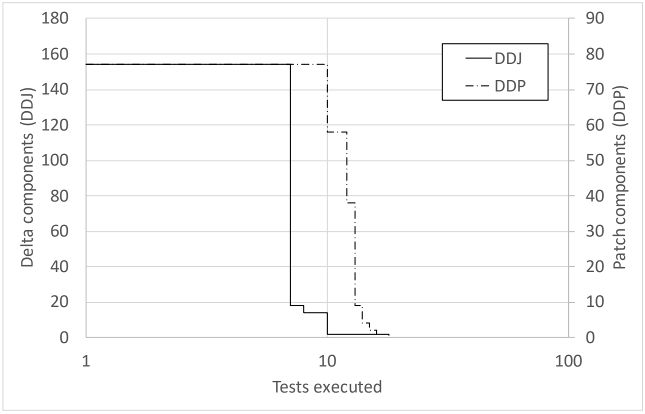


Figure 15: DD run for minimizing JSOUP-926 fix patch.

Table 2: Minimizing Regression Fixing Changes

P: tokens in patch, C: number of patch (delta) components, R: build success rate

Project	V_{good} (SLOC)	V_{bad} (SLOC)	$P_{V_{bad}V_{good}}$	P_{DDP}	P_{DDJ}	C_{DDP}	C_{DDJ}	R_{DDP}	R_{DDJ}
ANTLR-1543	14f05bb (15305)	aacd2a2 (14202)	9947	91	91	325	230	0.500 (20/40)	0.905 (19/21)
MATH-805	v2.0 (48495)	v2.2 (38234)	150195	728	101	3017	4320	0.149 (22/148)	0.949 (2148/2264)
HSQLDB-933	r3227 (137619)	r3262 (135802)	35037	105	27	952	1317	0.127 (105/827)	1.000 (91/91)
HSQLDB-1454	v2.3.2 (167638)	v2.3.3 (168563)	75907	183	75	1767	2650	0.179 (15/84)	0.676 (25/37)
JEDIT-3517	r12114 (93047)	r12710 (91556)	18462	86	2	454	634	0.555 (15/27)	1.000 (27/27)
JSOUP-920	b033535 (11146)	v1.10.3 (10565)	6792	2	2	251	451	0.230 (138/600)	0.893 (50/56)
JSOUP-926	v1.10.2 (11146)	v1.10.3 (10989)	1426	23	15	77	154	0.765 (13/17)	0.895 (17/19)
LOGBACK-1183	v1.0.13 (14905)	v1.1.0 (14700)	3072	34	21	101	165	1.000 (86/86)	1.000 (35/35)
Total	n/a (499301)	n/a (484611)	300838	**1252**	**334**	6944	9921	**0.226** (414/**1829**)	**0.946** (2412/**2550**)

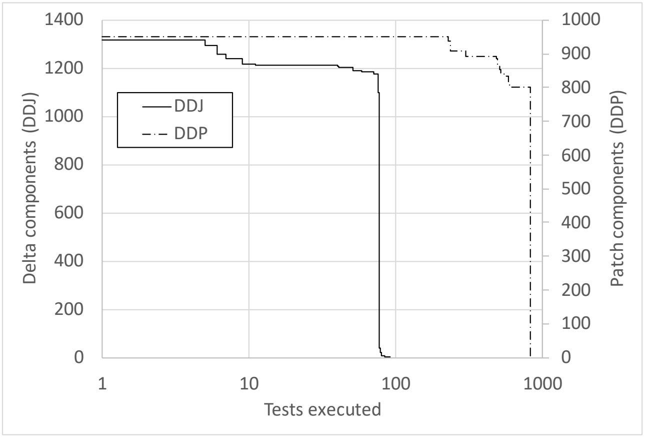


Figure 16: DD run for minimizing HSQLDB-933 fix patch.

due to increasing numbers of dependencies between text hunks in accordance with the sizes of the change sets as well as the lack of coupling rules for the text hunks. On the other hand, the best build success rate 1.00 was set for HSQLDB-933 and JEDIT-3517 by DDJ, and for LOGBACK-1183 by both DDP and DDJ. The DD run for minimizing the HSQLDB-933 regression fix is shown in Figure 16. Overall, DDJ extracted patches 3.7 times as small as DDP did at the cost of test executions 1.4 times as many as DDP required, both on average.

Note that the resulting patches are no more than first-aid patches that undo minimal failure-inducing changes. As opposed to the D4J experiment that is based on the actual bug fixes, the patches are qualitatively different from the actual fixing patches.

7 RELATED WORK

AST differencing and patching. Although several AST differencing methods and their implementations have been developed [18, 20, 26, 51], to the best of the authors' knowledge, no study has so far been made on AST patching that directly operates on AST nodes and even allows partial application of AST deltas like ours. Nevertheless, a few attempts have been made at tree patching for other tree structures such as XML [62] and JSON [10]. Unfortunately, however, we cannot employ XML/JSON patch tools for our method by converting AST into XML or JSON since their definitions of tree delta are not compatible with ours.

Refactoring reconstruction. Prete and others conducted an extensive study on refactoring reconstruction methods [49], where they presented a tool called Ref-Finder for detecting refactoring patterns as cataloged by Fowler [22]. Ref-Finder takes as input a pair of revisions from Java projects and reports refactoring instances as output. Ref-Finder first extracts facts from the given revisions using logical predicates such as method(methodName, typeName) denoting that a method named methodName is defined in a class or an interface named typeName. Then the difference between the revisions is obtained by taking the set difference of logical assertions in a simple case. For example, deletion of a method is easily identified by a disappearing assertion of method($\cdots$). For a complex change such as refactoring, Ref-Finder applies predefined template logic rules that cover 63 patterns from the Fowler's book. Compared to our approach, Ref-Finder is weak in identifying renamed entities since the set difference of logical assertions tells nothing about the correspondence between nameless entities. Our method is based on a fine-grained AST differencing method that provides mappings even between nameless AST nodes.

Recently, Tsantalis and others proposed a similar tool, RMiner [60], for identifying refactoring patterns occurring within a single GitHub commit by employing AST-based statement matching. Our method is based on full-scale AST differencing as opposed to RMiner's limited AST differencing up to statements. Thus DDJ can directly handle fine-grained patterns such as *Extract Variable* and *Inline Temp* that RMiner cannot. Moreover, our tool is capable of identifying 38 refactoring patterns between any given pairs of program versions, compared to RMiner's 15 patterns within a commit.

We plan to work on a detailed comparative study for these tools.

Delta debugging on code changes. Surprisingly few studies have ever tried DD on source code changes although DD has been applied to various failure-inducing circumstances such as program inputs [69], thread schedules [11], and program states [12]. Zeller demonstrated DD for narrowing down 178,000 changed GDB lines to a single failure-inducing change by making use of line-by-line differencing [67]. To reduce the number of tests executed, he grouped changes based on a file system hierarchy that involves directories/ files and also on common usage of identifiers. He also added a *failure resolution loop* to the DD step. In the loop body, if a group of changes causes a build failure, the error messages are scanned for identifiers, and then all changes that refer those identifiers are added to the group for another trial. This is repeated until the build succeeds or until there are no more changes to be added. Instead of this somewhat ad-hoc resolution, our method resolves such build failures based on the dependencies between changes described by a set of syntactic and semantic rules supported by AST differencing.

A distributed version control system called Git [59] provides a command `git-bisect` that identifies failure-inducing *commits* between good and bad commits based on a consistent test script. The command would be a good tool for fighting regressions if any commit contains relatively small set of changes [14]. DDJ does not require any development history. It only requires a pair of good and bad versions to isolate failure-inducing sets of delta components.

Automated patch extraction. Over the past several years, a considerable number of studies have been made on automatic software repair [9, 13, 17, 25, 31, 34, 37–39, 41, 42, 44, 47, 48, 50, 53, 55, 56, 63, 65]. The basic idea behind them is to generate fault fixing patch candidates that can be validated later. A number of approaches have been proposed for repairing different classes of faults under different conditions and hypotheses [23]. Although our method focuses on fixing regression bugs by minimizing the difference between good and faulty versions, it can also be used for extracting minimal patches from machine generated patches that are often indirect and lengthy.

8 THREATS TO VALIDITY

Granularity of delta component. An AST delta component is composed of tree hunks each of which is a set of edit operations aggregated as much as possible. For example, removal of a whole subtree is not interpreted as a set of node deletions but as a single deletion of the subtree. Such implicit grouping of changes may impact the result of DD on AST deltas. It would reduce the number of test executions when an implicit group of changes coincide with a group of syntactically and/or semantically coupled changes. On the other hand, it would needlessly increase the size of resulting patches when changes related to a target failure depend only on a single change in a large implicit group of changes.

Partial application of delta. Our implementation of partial application of AST deltas is just one of the possibilities. There might be a completely different design that better achieves the goal of the partial application: any sub-delta, or subset of delta components, of a delta is applicable to any source code where the full delta is applicable.

Adequacy of test. As Martinez and others pointed out, automatic program repair methods relying on test suites suffer from under-specified bugs, for which trivial or incorrect patches still pass the test suites [41]. Since DD relies heavily on tests as well, our method of extracting patches is affected similarly as suggested by the results of our experiment on the same dataset, namely Defects4J, as theirs. Even the task of minimizing a set of changes including a bug-fixing subset may depend on adequacy of tests.

Experiments. Our experiments are based on a proof-of-concept implementation for Java programs, which consist of 194 examples of bug fixes taken from Defects4J and 8 examples of regression bugs taken from 6 open source Java projects. The setting may not generalize to other programming languages, other real-life regressions, and/or bug fixes.

As for refactoring identification, we did not measure precision and recall in the experiments. Although we cannot precisely estimate the impact of low precision/recall on the results, we have not yet experienced negative effects caused by the false positives at least for both of the datasets.

The scalability of our method depends on the number of test executions, which is further broken down into the performance of each test execution: the size of source code to be built for a test and the complexity of the test itself. In the experiments performed on a machine with a 3.0GHz CPU and 8GB of memory, each individual DD run required several minutes to a day according to the number of test executions. The total amount of time spent on AST differencing and database construction was relatively small compared to that on the test executions. Each AST differencing task required a few to a dozen minutes according to the size of the source code and the gap between versions, while each database construction task required at most several dozen minutes.

9 CONCLUSION

Delta debugging is an approach to automating debugging activities based on systematic testing. It finds the cause of a regression of a program by minimizing changes between a working and a faulty versions of the program. In this study, we tackled an open problem of minimizing a huge amount of changes while avoiding combinations of changes that do not result in a testable program, especially in case that no software configuration management system is available.

Our solution to the problem is a rule-based method of decomposing changes into independent components both syntactically and semantically. A key technique in the method is to make use of tree differencing on abstract syntax trees (ASTs) instead of traditional differencing on plain texts for analyzing changes.

To evaluate the proposed method, we have developed DDJ, which is an implementation of DD on node-by-node AST changes of Java programs, and DDP, which is a language-agnostic implementation of DD on line-by-line source code changes. A couple of experiments have been conducted. One is on minimizing differences for each of the 194 pairs of good and faulty versions contained in the Defects4J dataset [33]. The other is on extracting patches for 8 real-life regression bugs from 6 open source Java projects. Compared to DDP, DDJ extracted patches of half the size at the cost of 5% more test executions for the former dataset and patches of 27% size at the cost of 40% more test executions for the latter, both on average.

We have created a Docker image containing DDJ and DDP in order to reproduce the results of the experiments. The image will be available at https://hub.docker.com/r/codecontinuum/.

ACKNOWLEDGMENT

This work was supported in part by JSPS KAKENHI Grant Number JP26280025. The second author was supported by NEDO's "Development of a highly dependable humanoid robot system that can work in unstructured environments" project.

REFERENCES

[1] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/93542.93576

[2] Anonymous. 2009. [HSQLDB-933] UPDATE statement with IN criterion recently broken. Retrieved August 4, 2018 from https://sourceforge.net/p/hsqldb/bugs/933/

[3] Cyrille Artho. 2011. Iterative Delta Debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246. https://doi.org/10.1007/s10009-010-0139-9

[4] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 97–105. https://doi.org/10.1145/604131.604140

[5] Jean-Francois Bergeretti and Bernard A. Carré. 1985. Information-flow and Data-flow Analysis of While-programs. *ACM Transactions on Programming Languages and Systems* 7, 1 (Jan. 1985), 37–61. https://doi.org/10.1145/2363.2366

[6] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. 2002. Union Slices for Program Maintenance. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM '02)*. IEEE Computer Society, Washington, DC, USA, 12–21. https://doi.org/10.1109/ICSM.2002.1167743

[7] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1-3 (June 2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[8] François Billioud. 2017. [JSOUP-920] Regression: Wrong parsing for clippath. Retrieved August 4, 2018 from https://github.com/jhy/jsoup/issues/920

[9] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/3106237.3106255

[10] Paul C. Bryan. 2008. JavaScript Object Notation (JSON) Patch. Retrieved August 4, 2018 from https://tools.ietf.org/html/rfc6902

[11] Jong-Deok Choi and Andreas Zeller. 2002. Isolating Failure-inducing Thread Schedules. In *Proceedings oxof the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 210–220. https://doi.org/10.1145/566172.566211

[12] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 342–351. https://doi.org/10.1145/1062455.1062522

[13] Zack Coker and Munawar Hafiz. 2013. Program Transformations to Fix C Integers. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Computer Society, Washington, DC, USA, 792–801. https://doi.org/10.1109/ICSE.2013.6606625

[14] Christian Couder. 2009. Fighting regressions with git bisect. Retrieved August 4, 2018 from https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html

[15] Benoit de Rancourt. 2012. [MATH-805] Percentile calculation is very slow when input data are constants. Retrieved August 4, 2018 from https://issues.apache.org/jira/browse/MATH-805

[16] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[17] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, Washington, DC, USA, 65–74. https://doi.org/10.1109/ICST.2010.66

[18] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[20] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[21] Python Software Foundation. 2001. difflib: helpers for computing deltas. https://docs.python.org/2/library/difflib.html

[22] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code (Addison-Wesley Object Technology Series)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[23] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* Early Access (2017), 36. https://doi.org/10.1109/TSE.2017.2755013

[24] Alex Groce, Daniel Kroening, and Flavio Lerda. 2004. Understanding Counterexamples with explain. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*. Springer, Berlin, Heidelberg, 453–456. https://doi.org/10.1007/978-3-540-27813-9_35

[25] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI-17)*. AAAI Press, Palo Alto, CA, USA, 1345–1351. https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603/13921

[26] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*. IEEE Computer Society, Washington, DC, USA, 279–288. https://doi.org/10.1109/WCRE.2008.44

[27] Masatomo Hashimoto and Akira Mori. 2012. Enhancing History-Based Concern Mining With Fine-Grained Change Analysis. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. IEEE Computer Society, Washington, DC, USA, 75–84. https://doi.org/10.1109/CSMR.2012.18

[28] Masatomo Hashimoto, Akira Mori, and Tomonori Izumida. 2015. A Comprehensive and Scalable Method for Analyzing Fine-Grained Source Code Change Patterns. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER '15)*. IEEE Computer Society, Washington, DC, USA, 351–360. https://doi.org/10.1109/SANER.2015.7081845

[29] Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. 2015. Extracting Facts from Performance Tuning History of Scientific Applications for Predicting Effective Optimization Patterns. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR '15)*. IEEE Computer Society, Washington, DC, USA, 13–23. https://doi.org/10.1109/MSR.2015.9

[30] Joern Huxhorn. 2016. [LOGBACK-1183] Message formatting regression. Retrieved August 4, 2018 from https://jira.qos.ch/browse/LOGBACK-1183

[31] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. 2009. BugFix: A Learning-Based Tool to Assist Developers in Fixing Bugs. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '09)*. IEEE Computer Society, Washington, DC, USA, 70–79. https://doi.org/10.1109/ICPC.2009.5090029

[32] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

[33] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[34] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[35] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Inform. Process. Lett.* 29, 3 (1988), 155 – 163. https://doi.org/10.1016/0020-0190(88)90054-3

[36] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/567532.567555

[37] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17))*. ACM, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[39] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[40] Frédéric Magniez and Michel de Rougemont. 2007. Property Testing of Regular Tree Languages. *Algorithmica* 49, 2 (2007), 127–146. https://doi.org/10.1007/s00453-007-9028-3

[41] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[42] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York,

NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[43] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[44] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[45] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 199–209. https://doi.org/10.1145/2001420.2001445

[46] Terence Parr. 2016. [ANTLR-1543] Type mismatch between left and right labels in left-recursive rule. Retrieved August 4, 2018 from https://github.com/antlr/antlr4/issues/1543

[47] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[48] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. 2011. Code-Based Automated Program Fixing. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 392–395. https://doi.org/10.1109/ASE.2011.6100080

[49] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based Reconstruction of Complex Refactorings. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[50] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[51] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. 2004. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*. IEEE Computer Society, Washington, DC, USA, 188–197. https://doi.org/10.1109/ICSM.2004.1357803

[52] Manos Renieris and Steven P. Reiss. 2003. Fault Localization with Nearest Neighbor Queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03)*. IEEE Computer Society, Washington, DC, USA, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[53] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

[54] Felipe Matos Santana. 2017. [JSOUP-926] Regression: CSS attribute selector. Retrieved August 4, 2018 from https://github.com/jhy/jsoup/issues/926

[55] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/2737924.2737988

[56] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *37th IEEE/ACM International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 471–482. https://doi.org/10.1109/ICSE.2015.65

[57] Chris Thunes. 2013. javalang: pure Python Java parser and tools. https://github.com/c2nes/javalang

[58] toro. 2016. [HSQLDB-1454] Null Pointer in CASEWHEN and CASE... WHEN use case. Retrieved August 4, 2018 from https://sourceforge.net/p/hsqldb/bugs/1454/

[59] Linus Torvalds. 2005. Git: fast, scalable, distributed revision control system. https://git-scm.com/

[60] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[61] tvojeho. 2011. [JEDIT-3517] NPE when scrolling to top after multiline delete (soft wrap). Retrieved August 4, 2018 from https://sourceforge.net/p/jedit/bugs/3517/

[62] Jari Urpalainen. 2008. An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. Retrieved August 4, 2018 from https://www.ietf.org/rfc/rfc5261.txt

[63] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[64] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* 10, 4 (July 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[65] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[66] Andreas Zeller. 1999. DD.py: Enhanced Delta Debugging class. Retrieved August 4, 2018 from https://www.st.cs.uni-saarland.de/dd/DD.py

[67] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99)*. Springer-Verlag, Berlin, Heidelberg, 253–267. https://doi.org/10.1007/3-540-48166-4_16

[68] Andreas Zeller. 2002. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[69] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (February 2002), 183–200. https://doi.org/10.1109/32.988498

[70] K. Zhang and D. Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262. https://doi.org/10.1137/0218082

[71] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning Dynamic Slices with Confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 169–180. https://doi.org/10.1145/1133981.1134002

[72] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise Dynamic Slicing Algorithms. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 319–329. https://doi.org/10.1109/ICSE.2003.1201211