LASE: Locating and Applying Systematic Edits by Learning from Examples

Na Meng* Miryung Kim* Kathryn S. McKinley*†
The University of Texas at Austin* Microsoft Research†
mengna09@cs.utexas.edu, miryung@ece.utexas.edu, mckinley@microsoft.com

***Abstract*—Adding features and fixing bugs in software often require *systematic edits* which are similar, but not identical, changes to many code locations. Finding all relevant locations and making the correct edits is a tedious and error-prone process. This paper addresses both problems by using edit scripts learned from multiple examples. We design and implement a tool called LASE that (1) creates context-aware edit scripts from two or more examples, and uses these scripts to (2) automatically identify edit locations and to (3) transform the code.**

We evaluate LASE on an oracle test suite of systematic edits from Eclipse JDT and SWT. LASE finds edit locations with 95% precision and 88% recall, and transforms them with 91% accuracy. We also evaluate LASE on 37 example systematic edits from other open source programs and find LASE is accurate and effective. A key contribution of this paper is an algorithm that combines two or more edits to make a script. This feature of learning from multiple examples is critical to accuracy; edit scripts created from only one example produce too many false positives, false negatives, or both. These results illustrate that LASE should help developers in automating systematic editing. Whereas most prior work either suggests edit locations *or* performs simple edits, LASE is the first to do both for nontrivial program edits.

I. INTRODUCTION

To add features, fix bugs, refactor, and adapt to new APIs, developers often perform *systematic edits*—similar, but not identical, changes to many locations. Kim et al. observe that 75% of structural changes involve systematic change patterns [1]. Nguyen et al. find that 17% to 45% of bug fixes are systematic; 86% to 92% occur in methods that perform similar functions and/or object interactions [2]. When an API evolves, client applications must systematically adapt by constructing new objects, passing new arguments, or replacing API calls [3]. In the context of product-line software development, programmers often make similar changes to related products. In all these examples, programmers manually locate many code locations and then apply similar but not identical edits to them one by one. This work is tedious and error-prone.

Existing tools either suggest code locations or transform the code, but not both, except for specialized or trivial edits. For example, much prior work infers code patterns or takes them as input to find buggy code violating the patterns [4]–[8], or they identify code clones that may require similar edits [2], [9]. However, these tools do not fix programs by applying source transformations. Other tools apply code transformations, but the the user must specify the target locations [10]. The closest work locates and fixes violations of simple local invariants by applying simple edits [11]–[13], performing very stylized API usage corrections [14], or applying identical, lexical edits to similar text [15]. However, these tools do not perform nontrivial program transformations. For example, Andersen and Lawall [14]'s approach focuses on API changes and cannot always correctly position edits, because they compute positions without considering data and control dependence constraints that edits have on their surrounding context.

This paper introduces LASE (Locating and Applying Systematic Edits) to help developers evolve programs by performing systematic, general edits that add features and fix bugs. Developers specify two or more example methods that they edited by hand to LASE. LASE learns a context-aware edit script and uses it to find other edit locations and to apply a customized edit to each target location. From the exemplar changed methods, LASE infers the *most specific generalization*, resulting in a *partially abstract, context-aware* edit script.

Intuitively, LASE infers edit *operations* (insert, delete, update, and move) common to all examples and abstracts or omits edit operations that differ between the examples. For instance, if two examples delete the same `if` statement, but disagree on specific identifiers (variable, type, or method names), LASE creates a partially abstract delete statement in the edit script that abstracts the uncommon ones. It uses concrete identifiers if they are common among all examples. LASE next computes the *context* of the inferred edit operations, which determines the relative position of edits in the method. For instance, if the edit operation inserts a statement S and S is the first statement in a `while` loop in both edits, the context is the `while` loop, and the relative position is `while`'s first child. LASE determines the largest common context with a novel algorithm that combines clone detection [16], maximum common embedded subtree extraction [17], and dependence analysis. The result is an edit script that consists of partially-abstract edit operations and context.

LASE uses the script to find edit locations and transform the code. LASE relies on the assumption that similar contexts require similar code changes. LASE thus searches for code locations that match the context of the script. If it finds a new code location that matches the context, it customizes the script for the method. LASE replaces the abstract identifiers in the script with concrete identifiers used by the target method. It then applies this customized edit script and suggests the changed method for developers' review.

We perform a thorough evaluation of LASE and its features

on programmer applied systematic edits drawn from open-source programs. We use real-world repetitive bug fixes that required multiple check-ins in Eclipse JDT and SWT as an oracle. For these bugs, developers applied supplementary bug fixes because the initial patches were either incomplete or incorrect [18]. We evaluate LASE by learning edit scripts from the initial patches and by determining if LASE correctly derives the subsequent, supplementary patches. On average, LASE identifies edit locations with 95% precision and 88% recall. The accuracy of applied edits is 91%—the tool-generated version is 91% similar to the developers' version. In several confirmed cases, LASE identifies and performs edits on locations that the developers missed. We also evaluate LASE on a test suite of 37 systematic edits drawn from five Java open source projects. In these experiments, we find that LASE's approach to finding locations has significantly fewer false positives and negatives when compared to other approaches, such as learning edit scripts from a single example with fully abstract [10] or fully concrete identifier names. The partially abstract context and edits that LASE infers from multiple examples are critical to achieving high recall, precision, and accuracy.

To our knowledge, LASE is the first tool to learn nontrivial program edits from multiple changed methods, to use them to find other edit locations, and to perform customized program transformations at each location. LASE reduces the time and effort of specifying code changes, manually searching for methods that need a systematic edit, applying the edit, and thus eases the pain of tedious and error-prone hand editing.

II. MOTIVATING EXAMPLE

This section uses a motivating example drawn from revisions to org.eclipse.compare to show LASE's work flow and compare it to our prior work. Figure 1 shows two methods with similar changes: `mA` and `mB`. The unchanged code is in black, added code is in **blue** with '+', and deleted code is in **red** with '−'. The changes to method `mA` delete two print statements (lines 3 to 4), insert a local variable declaration `next` for each enumerated element (line 9 of `mA`), perform a type cast to `MVAction` on the variable (line 10), and then process it. In our prior work [10], SYDIT relied on the developer to specify this one exemplar change and also to specify *where* to apply the inferred edit transformation.

An edit script consists of edit operations (add, delete, move, and update) and *context*, other statements in the method on which the edit statements are control and/or data dependent that serve to position the edit. Figures 4 and 5 show example edit scripts. Gray bars represent edit context, red bars represent deleted code, and blue bars represent inserted code. Figure 4 shows the edit script inferred by SYDIT. Unfortunately in this example, SYDIT cannot apply it to either `mB` or `mC` because the script is too specific to `mA`. In general, learning a systematic edit from a single example has the following limitations:

- It may *over specify* the learnt edit to the example. In this case, learning only from method `mA` includes deleting two statements (lines 3 and 4), but these operations are specific to `mA` and prevent the script from being applied to other methods, such as `mB` and `mC`.
- It may *over generalize* the edit. Given only one example, it is unclear which concrete identifiers will be the same in other methods and thus abstracting all identifiers produces a more flexible script. For instance, abstracting `e` of type `Iterator` to `v$0` of type `t$0` enables `v$0` to match variables `e` in `mA` and `iter` in `mB`, both of which have type `Iterator`. However, full abstraction is so flexible that `v$0` matches `task` in `mD` (we omit `mD`'s code for brevity), which is not an instance of type `Iterator` and `mD` is not a location where the programmer wants to apply the learnt edit. Searching with this edit script produces many spurious, false positive matches.

This paper seeks an edit script that serves double duty, both finding edit locations and accurately transforming the code. We take the approach to learn from two or more example edits given by the developer to solve the problems of over generalization and over specification. Although developers may also want to directly create or modify a script, since they already make similar edits to more than one place, we think providing multiple examples is a natural interface. Developers may also use a tool like Repertoire [19] to identify multiple code examples that changed similarly.

Figure 3 shows the workflow for LASE where the developer specifies two exemplar changed methods, `mA` and `mB`. LASE then infers the edit script shown in Figure 5. LASE uses the edit script to find locations for which it is appropriate, specializes the script for each location, applies the script, and suggests the resulting code to the developer. To create edit scripts from multiple examples requires new algorithms that identify common changes and context which abstract or omit differences. None of these algorithms are necessary when learning from a single exemplar edit.

LASE first finds the longest common edit operation subsequence among exemplar edits to filter out operations specific to only a single example. Notice that LASE omits the deleted `print` statements from `mA` (lines 3 and 4) in Figure 5 because the edits are not common to `mA` and `mB`. LASE next extracts the context for each common edit and then it determines the largest common edit-relevant context by combining clone detection, maximum common embedded subtree extraction on the Abstract Syntax Tree (AST), and program dependence analysis. Finally, LASE abstracts operations and context in the script. For example, the script in Figure 5 uses type name `Iterator` because it is common to `mA` and `mB`. However since field access `fActions` in `mA` and method invocation `getActions()` in `mB` match, but differ, it generalizes them to an abstract identifier `u$0:FieldAccessOrMethodInvocation`.

The next few sections describe how LASE creates edit scripts, uses the edit context to search for additional locations, and applies the edit.

III. APPROACH

This section summarizes LASE's three phases and formalizes our terminology. The following sections then describe

mA_{old} to mA_{new}

```
1. public void textChanged (TEvent event) {
2.   Iterator e=fActions.values().iterator();
3. - print(event.getReplacedText());
4. - print(event.getText());
5.   while(e.hasNext()){
6. -  MVAction action = (MVAction)e.next();
7. -  if(action.isContentDependent())
8. -  action.update();
9. +  Object next = e.next();
10.+  if (next instanceof MVAction){
11.+  MVAction action =(MVAction)next;
12.+  if(action.isContentDependent())
13.+    action.update();
14.+ }
15.  }
16.  System.out.println(event + " is processed");
17.}
```

mB_{old} to mB_{new}

```
1. public void updateActions () {
2.   Iterator iter = getActions().values()
                        .iterator();
3.   while(iter.hasNext()){
4. -   print(this.getReplacedText());
5. -   MVAction action=(MVAction)iter.next();
6. -   if(action.isDependent())
7. -   action.update();
8. +   Object next = iter.next();
9. +   if (next instanceof MVAction){
10.+    MVAction action =(MVAction)next;
11.+    if(action.isDependent())
12.+      action.update();
13.+  }
14.+  if (next instanceof FRAction){
15.+    FRAction action = (FRAction)next;
16.+    if(action.isDependent())
17.+    action.update();
18.+  }
19.  }
20.  print(this.toString());
21.}
```

Fig. 1. A systematic edit to two methods based on revisions from 2007-04-16 and 2007-04-30 to org.eclipse.compare

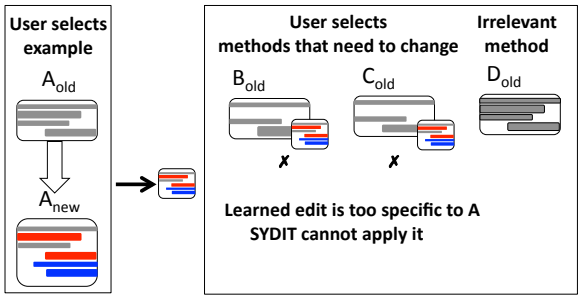

Fig. 2. SYDIT learns an edit from one example. A developer must locate and specify the other methods to change.

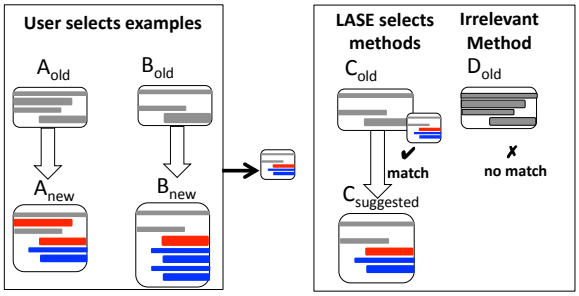

Fig. 3. LASE learns an edit from two or more examples. LASE locates other methods to change.

```
1. … … method_declaration(… …){
2.   T$0 v$0 = v$1.m$0().m$1();
3.   DELETE: m$2(v$2.m$3());
4.   DELETE: m$2(v$2.m$4());

3.   while(v$0.m$5()){
4.     UPDATE: T$1 v$3 = (T$1)v$0.m$6();
5.       TO: T$2 v$4 = v$0.m$6();
6.     if(v$3.m$7()){
7.       … …
8.     }
MOVE
9.     INSERT: if(v$4 instanceof T$1){
10.      INSERT: T$1 v$3 = (T$1)v$4;
11.      … …
12.          }
```

Fig. 4. Edit script from SYDIT abstracts all identifiers. Gray marks edit context, red marks deletions, and blue marks additions.

```
1. … … method_declaration(… …){
2.   Iterator v$0 = u$0:FieldAccessOrMethodInvocation
                        .values().iterator();
3.   while(v$0.hasNext()){
4.     UPDATE: MVAction action = (MVAction)v$0.next();
5.       TO: Object next = v$0.next();
6.     if(action.m$0()){
7.       … …
8.     }
MOVE
9.     INSERT: if(next instanceof MVAction){
10.      INSERT: MVAction action = (MVAction)next;
11.      … …
12.          }
```

Fig. 5. Edit script from LASE abstracts identifiers that differ in the examples and uses concrete identifiers for common ones. Gray marks edit context, red marks deletions, and blue marks additions.

each phase in detail. We analyze and transform an Abstract Syntax Tree (AST) representation of the program. We also represent edit operations and context using AST. Phase I takes as input multiple changed methods M and generates AST edits for each of them. It identifies the common edit operations E among all examples and extracts the largest common context C relevant to E to create an edit script Δ_P. Phase II identifies other edit locations M_f that match the context C elsewhere in the program. Phase III customizes the edit to each new location $m_f \in M_f$ and applies it.

Phase I: Generating an Edit Script. Generating an edit script from multiple examples has four steps.

1) *Generating Syntactic Edits.* For each changed method $m_i \in M$=$\{m_1, m_2, \ldots, m_n\}$, LASE compares the old and new versions of m_i and creates an edit script: $E_i = \{e_1, e_2, \ldots, e_k\}$ where e_i is a delete, insert, move, or update of AST statements.
2) *Identifying Common Edit Operations.* LASE identifies the longest common edit operation subsequence E_c such that $\forall 1 \leq i \leq n$, $E_c \subseteq E_i$, and E_c preserves the sequential order of operations in each E_i.
3) *Abstracting Identifier Names.* When common edit operations $e \in E_c$ in the examples use distinct identifier (variable, type, and method) names, LASE replaces the concrete identifier names with abstract names, resulting E. Otherwise, it uses the original concrete identifiers.

$$LCEOS(s(E_i,p),s(E_j,q)) = \begin{cases} 0 & \text{if } p=0 \text{ or } q=0 \\ LCEOS(s(E_i,p-1),s(E_j,q-1))+1 & \text{if } equivalent(e_p,e_q) \\ max(LCEOS(s(E_i,p)),s(E_j,q-1)),LCEOS(s(E_i,p-1),s(E_j,q))) & \text{if } !equivalent(e_p,e_q) \end{cases} \quad (1)$$

$s(E_*,i)$ represents the edit operation subsequence $e_1,...,e_i$ in E_*.

4) *Extracting Common Edit Context.* LASE finds the largest common context C relevant to E using code clone detection, maximum common embedded subtree extraction, and dependence analysis. LASE then abstracts identifiers in context C.

The result of this process is a *a partially abstract, context-aware edit script* Δ_P.

Phase II: Finding Edit Locations. LASE uses the edit script's context C to search for methods M_f that match C.

Phase III: Applying an Edit. For each method $\texttt{m}_f \in M_f$, LASE concretizes identifiers and code positions in Δ_P to m_f, producing Δ_f, applies it, and suggests a modified version $\texttt{m}_f\texttt{'}$.

IV. Phase I: Learning Edits from Multiple Examples

A. Generating Syntactic Edits

For each exemplar changed method $\texttt{m}_i \in M$, LASE compares the AST of $\texttt{m}_i$'s old and new versions and creates a sequence of node edit operations E_i consisting of:

- **insert (Node** u**, Node** v**, int** k**):** insert u and position it as the $(k+1)^{th}$ child of v.
- **delete (Node** u**):** delete u.
- **update (Node** u**, Node** v**):** replace u's label and AST type with v's while maintaining u's position in the tree.
- **move (Node** u**, Node** v**, int** k**):** delete u from its current position and insert it as the $(k+1)^{th}$ child of v.

This process uses Fluri et al.'s AST differencing algorithm [20] and results in a set of concrete edits $\{E_1, E_2, \ldots E_n\}$, where $n \geq 2$ is the number of exemplar methods

B. Identifying Common Edit Operations

LASE identifies common edit operations in $\{E_1, E_2, \ldots E_n\}$ by iteratively comparing the edits pairwise using a Longest Common Edit Operation Subsequence (LCEOS) algorithm, which is a modified version of the Longest Common Subsequence (LCS) algorithm [21], as shown in Equation (1). For instance, given $\{E_1, E_2, E_3\}$, LASE first computes $E_{c1} = LCEOS(E_1, E_2)$, $E_{c2} = LCEOS(E_1, E_3)$, and finally computes $E_c = LCEOS(E_{c1}, E_{c2})$. We do not require exact equivalence between edit operations because systematic edits are often similar, but not exactly the same. We define the comparison function $equivalent(e_p, e_q)$ for two edit operations to return equivalence for inexact matches as follows.

LASE first applies *concreteMatch*(e_i, e_j, t_s), which takes as input two concrete edit operations, e_i and e_j, and a threshold t_s. It compares e_i and e_j based on edit type and edited node u's AST *label*, i.e., a string representation of the AST node. If two operations are of the same type and the labels' bi-gram string similarity [22] is above the threshold t_s, the function returns true. By default, we set t_s to 0.6 to increase the number of matches between different but similar edits.

If two edit operations fail the *concreteMatch* test, LASE applies *abstractMatch*(e_i, e_j), which first converts all identifiers of types, methods, and variables in the edited node u's label to abstract identifiers `t$`, `m$`, and `v$`. If the edit type and their labels' abstract representation match, *abstractMatch* returns true. The result is the set of concrete edits that are *equivalent* and common to all exemplar methods, but their identifiers and AST types may not match.

C. Generalizing Identifiers in Edit Operations

LASE next generalizes identifier names as needed. When all the edits agree on an identifier, LASE uses the concrete name. If one or more edits use a different identifier name, LASE generalizes the name. For example, Figure 1 shows e_A = **delete(** `MVAction action=(MVAction)e.next()` **)** matches with e_B = **delete(**`MVAction action=(MVAction)iter.next()` **)**. When LASE detects the divergent identifiers `e` vs. `iter`, it generalizes them by creating a fresh abstract variable name `v$0`, substituting it for the original names, and creating e = **delete(**`MVAction action=(MVAction)v$0.next()` **)**. LASE records the pairs (`e`, `v$0`), (`iter`, `v$0`) in a map. LASE then substitutes `v$0` for all instances of `e` in $\texttt{m}_A$ and E_A and all instances of `iter` in $\texttt{m}_B$ and E_B to enforce a consistent naming for all edit operations and edit context. The result is a list of partially abstract edit operations E.

D. Extracting Common Edit Context

This section explains how LASE extracts the common edit context C for E from the exemplar methods with clone detection and then refines this context based on consistent identifiers usage, AST subtree extraction, and control and data dependences.

1) Finding Common Text with Clone Detection: For each $\{E_1, E_2, \ldots, E_n\} \in E$, LASE extracts unchanged context. It aligns each two changed methods' unchanged context based on the common edit operations they share and uses text-based clone detection to find clones in each aligned code segment pair. When two exemplar edits: E_1 and E_2, share a single edit operation, we use the edited node n_1 in $\texttt{m}_1$ and the edited node n_2 in $\texttt{m}_2$ to divide the unchanged code in each method into two parts, code before the edited node (S_1 in $\texttt{m}_1$, S_2 in $\texttt{m}_2$) and after the edited node (T_1 in $\texttt{m}_1$, T_2 in $\texttt{m}_2$). LASE compares each segment pair (S_1 vs. S_2, T_1 vs. T_2) using CCFinder [16] to detect any clones. This step reveals all possible common *text* shared between each two methods in sequence, C_{text}.

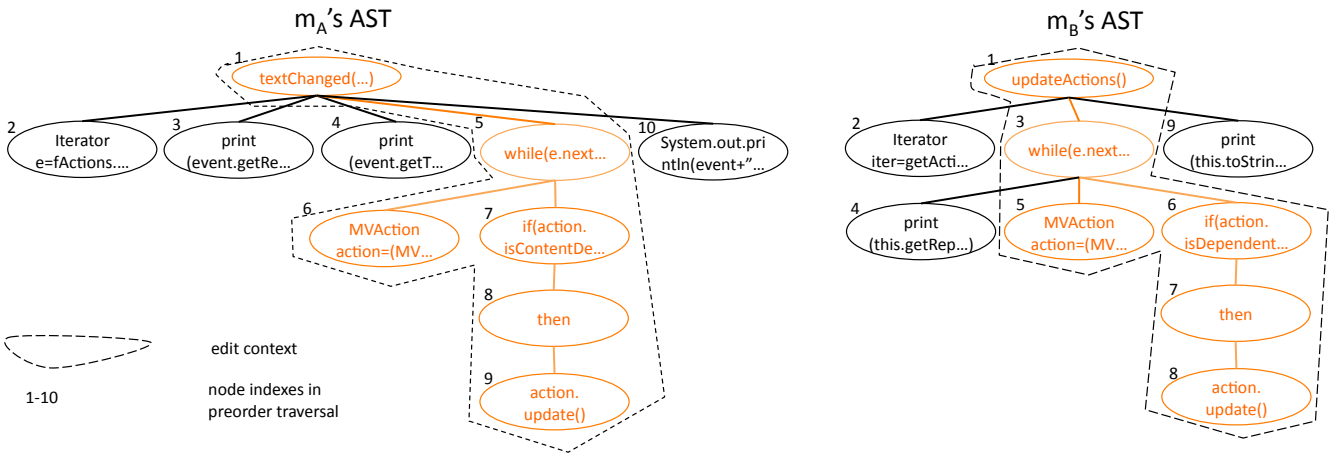

Fig. 6. $\mathtt{m}_A$'s and $\mathtt{m}_B$'s AST

2) Generalizing identifiers: Because clone detection uses text similarity, it does not guarantee that an identifier in one method is mapped consistently to identifiers in other methods. LASE collects all identifier mappings between each two methods' clone pairs. If there are conflicting identifier mappings, LASE only keeps the most frequent one, removing the rest and their corresponding clone pairs from the context. If two different concrete identifiers map consistently with each other, LASE generalizes them to a fresh abstract identifier and substitutes it for both identifiers in all statements in the respective methods and the context to create an abstract common context C_{abs}.

3) Extracting Common Subtree(s) with MCESE: Because the clone detection algorithm only uses text matching, the AST structure of two nodes may mismatch, e.g., two matching nodes may have different unmatched parent nodes. To solve this problem, LASE uses an off-the-shelf Maximum Common Embedded Subtree Extraction (MCESE) algorithm [17] to find the largest common forest structure between each two methods' ASTs, as shown in Equation 2. This algorithm traverses each tree in pre-order, indexes nodes, and encodes the tree structure into a node sequence. By computing the longest common subsequence between each two sequences and reconstructing trees from the subsequence, LASE finds the largest common embedded subtree(s), C_{sub}.

$$MCESE(s,t) = \begin{cases} 0 & \text{if } s \text{ or } t \text{ is empty} \\ max \begin{cases} MCESE(head(s), head(t)) \\ +MCESE(tail(s), tail(t)) + 1, \\ MCESE(head(s)tail(s), t), \\ MCESE(s, head(t)tail(t)) \end{cases} & \begin{array}{l} \text{if } equivalent(s[0], t[0]) \\ \\ \\ otherwise \end{array} \end{cases} \quad (2)$$

Consider $\mathtt{m}_A$'s and $\mathtt{m}_B$'s AST in Figure 6. LASE traverses $\mathtt{m}_A$'s AST in pre-order, indexes nodes, and encodes the tree into node sequence s = [1, 2, -2, 3, -3, 4, -4, 5, 6, -6, 7, 8, 9, -9, -8, -7, -5, 10, -10, -1], where notation "$-$" marks finishing the traversal of current node. Indexes X and $-X$ mark the boundaries of the subtree rooted at X's node. Similarly, LASE creates sequence t = [1, 2, -2, 3, 4, -4, 5, -5, 6, 7, 8, -8, -7, -6, -3, 9, -9, -1] for $\mathtt{m}_B$. We then use Equation (2) to find the longest common subsequence between them, which corresponds to subsequence $[1, 5, 6, -6, 7, 8, 9, -9, -8, -7, -5, -1]$ of s and $[1, 3, 5, -5, 6, 7, 8, -8, -7, -6, -3, -1]$ of t. The reconstructed trees out of these sequences are colored with orange and circled with dash lines.

In the equation, $head(s)$ returns the sequence of nodes sub-rooting at $s[0]$ (excluding $s[0]$), while $tail(s)$ returns the subsequence following $-s[0]$. For instance, given a sequence s=[1, 2, -2, -1, 3, -3], $head(s) = [2, -2]$, $tail(s) = [3, -3]$. The function $equivalent(i, j)$ checks string equality between the two nodes' labels.

4) Refining Edit Context with Dependence Analysis: The common text extracted between any two methods may include irrelevant code, i.e., code that does not have any control or data dependence relations with edited code. Blindly including them as edit context may put unnecessary constraints on potential edit locations in need of similar edits, causing false negatives in edit location search. LASE thus refines the extracted text using control and data dependence analysis to produce the edit context.

LASE performs control and data dependence analysis on each changed method to find context nodes relevant to any edited node involved in E. For both control and data dependence analysis, we include all nodes transitively depended on by an edited node as edit relevant context C_{dep}. $C = C_{sub} \bigcap C_{dep}$ is the final edit context extracted for E. If C_{dep} is empty, instead of an empty context, LASE simply uses C_{sub} as C in order to find edit locations. LASE combines E and C to create a *partially abstract, context-aware edit script*, Δ_P, where each edit operation in E is described with respect to context C.

V. Phase II: Finding Edit Locations

Given an edit script Δ_P, LASE searches for methods containing Δ_P's context C. Based on our assumption that methods containing similar edit contexts are more likely to experience similar changes, LASE suggests them as edit locations.

Because C is partially abstract, it contains both concrete and abstract identifiers. When LASE matches C with a method `m`, it matches concrete identifiers by name and matches abstract identifiers by identifier type or AST node type. For instance, `Iterator` in C only matches `Iterator` in `m`. An abstract name, such as `v$0`, matches any variable, while `u$0_FieldAccessOrMethodInvocation` only matches `FieldAccess` or `MethodInvocation` AST nodes. LASE reuses the MCESE algorithm described in Section IV-D to find the maximum common context between C and `m`, but redefines the $equivalent(i, j)$ function to compare concrete identifiers based on string equality and abstract identifiers based on identifier type and AST node type equality.

If all the common context between C and `m` matches each node in C, LASE suggests `m` as an edit location $\mathtt{m}_f$.

VI. Phase III: Applying the Edit

To apply the edit to a suggested location m_f, LASE first customizes Δ_P for $\mathtt{m}_f$ and then applies the edit. For this process, we slightly modify the edit customization and edit application algorithms that we introduced previously [10]. The customization algorithm replaces all abstract identifiers in Δ_P with corresponding concrete identifiers from $\mathtt{m}_f$ using the $equivalent(i, j)$ function defined above. In addition, it positions each edit operation concretely in the target method in terms of AST node positions. It uses data and control dependences in the edit script and target method to maintain correct and consistent relationships between identifiers and statements. The result is Δ_f, which fully specifies each edit operation as a modification of the AST with concrete identifiers and concrete node positions within each target AST. LASE then applies this customized concrete edit script and suggests a new version for developers to review.

VII. Evaluation

This section evaluates the precision and recall of LASE for finding the correct edit locations and the accuracy when LASE applies edits. We first use an oracle test suite with multiple systematic edits that fix the same bug in multiple commits, which are drawn from two open-source programs. We then explore the differences between using multiple examples and one example, between LASE's and other variants of edit scripts, on a test suite of 37 systematic edits from other five Java open-source programs.

A. Precision, recall, and accuracy with an oracle data set

To measure LASE's precision, recall, and accuracy, we use data from Park et al.'s study on supplementary bug fixes [18]. They find a group of bugs that were fixed in multiple commits to understand the characteristics of incomplete or incorrect bug fixes. They perform clone detection among bug patches labeled with the same bug ID to find repetitive, similar bug fixes [19]. If a bug is fixed more than once and there are clones of at least two lines between its bug patches checked in at different times, we manually examine these methods for systematic changes. We find 2 systematic edits in Eclipse JDT and 22 systematic edits in Eclipse SWT. In Table I, the first two rows are from JDT, while the rest are from SWT.

Index	Bug (patches)		methods	Edit Location Σ	✓	P(%)	R(%)	A(%)	Operations E	C	A_E(%)
1	73784	(2)	4	4	4	100	100	53	7	2	29
2	82429	(2)	16	13	12	92	75	81	9	9	100
3	14344	(2)	4	4	4	100	100	100	6	6	100
4	139329	(3)	6	2	2	100	33	74	6	3	50
5	142947	(6)	12	12	12	100	100	100	1	1	100
6	91937	(2)	3	3	3	100	100	95	5	3	60
7	103863	(4)	7	7	7	100	100	100	34	34	100
8	129314	(3)	4	4	4	100	100	100	2	2	100
9	134091	(3)	4	4	4	100	100	73	24	24	100
10	139329	(3)	3	4	3	75	100	100	1	1	100
11	139329	(3)	3	3	3	100	100	88	12	12	100
12	142947	(6)	7	7	6	86	86	76	6	6	100
13	76182	(2)	6	6	6	100	100	90	6	6	100
14	77194	(3)	3	3	3	100	100	97	13	13	100
15	86079	(3)	3	3	3	100	100	100	25	25	100
16	95409	(2)	8	8	8	100	100	78	4	4	100
17	97981	(2)	4	3	3	100	75	100	3	3	100
Average			**6**	**5**	**5**	**97**	**92**	**89**	**10**	**9**	**91**
18	74139	(2)	3	5	3	60	100	100	1	1	100
19	76391	(2)	6	3	3	100	50	100	3	3	100
20	89785	(3)	5	5	5	100	100	95	5	3	60
21	79107	(2)	3	2	2	100	67	92	4	4	100
22	86079	(4)	4	2	2	100	50	100	8	8	100
23	95116	(4)	5	4	4	100	80	100	3	3	100
24	98198	(2)	9	15	9	60	100	95	3	3	100
Average			**5**	**5**	**4**	**89**	**78**	**97**	**4**	**4**	**94**
Total Average			**6**	**5**	**5**	**95**	**88**	**91**	**8**	**8**	**92**

TABLE I
LASE'S EFFECTIVENESS ON REPETITIVE BUG PATCHES TO ECLIPSE

We view the patches as an oracle of a correct systematic edit and test if LASE can produce the same results as the developers from the first two fixes in this set of systematic fixes. Since the developers may not be perfect, there may be incorrect edits or some missing edits (indeed, we found a couple), and we cannot control for this potential problem. If LASE however produces the same results as developers did in multiple patches, this indicates that LASE can help programmers detect edit locations earlier, reduce errors of omissions, and make systematic edits.

We give LASE as input two random changed methods in the first patch. If there is only one changed method checked in the first patch, we randomly select the second one from the next patch. LASE generates an edit script from these two examples, finds edit locations, customizes the edit for each location, and applies it to suggest a new version.

Table I shows the results on this data set. The table numbers each edit, lists the **Bug** identifier, the number of **patches**, and **methods** that developers changed. For each **Edit Location**, we present Σ: the number of methods that LASE identifies as change locations; ✓: the number of methods correctly identified; precision **P**: the percent of identified edit locations compared to the correct locations out of all found locations; recall **R**: the percentage of correct locations out of all expected locations; accuracy **A**: the similarity between the tool-suggested version and the expected version, only for

edited methods. For each edit **Operation**, we present **E**: the number of edit operations shared among repetitive fixes for the same bug, i.e., operations we expect LASE to infer; **C**: the number of lines of context inferred by LASE; and $\mathbf{A}_E$: the percentage of operations actually inferred over expected operations. LASE locates edit positions with respect to the oracle data set with 95% precision, 88% recall, and performs edits with 91% accuracy. We determine precision, recall, and accuracy by visual inspection, compilation, and testing.

In the table, examples are grouped into two sets based on whether LASE refined the context with program dependence analysis or not (see Section IV-D4). The first 17 edits have non-empty C_{dep}, and thus $C = C_{sub} \bigcap C_{dep}$. The last 7 edits have empty C_{dep} and thus $C = C_{sub}$. Most of the inferred edits are nontrivial and LASE handles these cases well. For instance, edit 7 requires 34 operations. LASE correctly infers all 34 of them, correctly suggests 5 edit locations, and correctly applies customized edits with 100% accuracy. Finally, although not listed in this table, LASE also suggested some omitted edits which developers *missed*. These missing edits include one customized edit for edit 8, two customized edits for edit 16, and six customized edits for edit 24. We confirmed these omissions with the developers and they *validated* our suggestions, which means that LASE will help developers make systematic edits consistently and reduce errors of omission.

LASE cannot guarantee 100% edit application accuracy for four reasons. First, the inferred edit is sometimes a subset of the exemplar edited methods, so LASE cannot suggest extra edits specific to a single location. Second, when abstract identifiers do not have corresponding concrete identifiers in an edit location (e.g., identifiers only used in inserted statements), LASE cannot decide how to concretize them. Third, based on string similarity, LASE's AST differencing algorithm cannot always infer edits correctly when two strings are quite different. Fourth, LASE's LCEOS algorithm cannot always find the best longest common edit operation subsequence between two sequences because it does not enumerate or compare all possible longest common subsequences between the two to decide the best one. Although each of these problems occurred, none occurs frequently.

The number of exemplar edits from which LASE learns a systematic edit affects its precision, recall, and accuracy. To determine how sensitive LASE is to different numbers of exemplar edits, we randomly pick 5 cases in the oracle data set. For each case, LASE enumerates all possible ways to construct an exemplar edit set, evaluates the precision, recall, and accuracy for each set separately, and calculates an average for exemplar edit sets with the same cardinality to determine how sensitive LASE is to choice of exemplar edits and the number of input examples. Table II shows the results.

The table shows that precision **P** does not change as a function of the number of exemplar edits. However, recall **R** does increase with the number of exemplar edits. The more exemplar edits provided, the less common context is likely to be shared among them, the more methods may match the context and are suggested as edit locations. In theory, **P** can go down when more diverse examples are given, but this case did not occur in our tests.

	# of exemplar edits	P(%)	R(%)	A(%)
Case 1	2	100	51	73
	3	100	82	72
	4	100	96	72
	5	100	100	71
Case 2	2	100	88	100
	3	100	94	100
Case 3	2	100	66	100
	3	100	94	100
	4	100	100	100
	5	100	100	100
Case 4	2	100	75	89
	3	100	75	93
Case 5	2	100	72	100
	3	100	88	100
	4	100	96	100

TABLE II
LASE'S EFFECTIVENESS WHEN LEARNING FROM MULTIPLE EXAMPLES

The table shows that accuracy **A** varies inconsistently with the number of exemplar edits, because it strictly depends on the similarity between methods. For instance, when exemplar methods are diverse, the fewer common edit operations extracted, and the lower accuracy of LASE. On the other hand, when exemplar methods are similar, adding exemplar methods may not decrease the number of common edit operations, but it may induce more identifier abstraction and result in a more flexible edit script, which will help LASE increase accuracy.

B. Comparing Settings for Edit Script Learning

This section compares edits learned from single examples to edits learned from multiple examples. We use LASE to generate edit scripts from example pairs and our prior tool, SYDIT, to generate edit scripts from single examples. We use LASE to find matching locations and apply the resulting edits. (SYDIT does not find edit locations, but instead relies on developers to choose locations.) These experiments show that edits learned from single examples are not as useful for finding other locations, and motivate using two or more examples.

We measure precision and recall of edit location suggestion and accuracy of edit application on a test suite we developed for SYDIT [10] from five open-source programs: jEdit, Eclipse jdt.core, Eclipse compare, Eclipse core.runtime, and Eclipse debug. This suite contains 56 pairs of exemplar changed methods. Each pair of methods demonstrates at least one similar edit operation and the two methods are at least 40% similar according to the syntactic program differencing results (see Section IV-A). We remove the simplest cases, e.g., edits on initially empty methods or only one statement, resulting in 37 pairs of exemplar edits. For each pair, we extend the oracle set of exemplar edits as follows. We first apply LASE to infer the systematic edit demonstrated by both methods and search for edit locations in the program's original version. Then we manually examine these locations. If a suggested location is

ID	methods	Two Examples					One Example				
		Σ	✓	P(%)	R(%)	A(%)	Σ	✓	P(%)	R(%)	A(%)
1	5	6	5	83	100	100	10	5	50	100	100
2	2	3	2	67	100	80	7	2	29	100	100
3	5	7	5	71	100	100	277	1	0	25	100
4	2	3	2	67	100	96	596	2	0	100	97
5	5	6	5	83	100	100	5	3	60	60	100
6	2	73	2	3	100	100	3354	2	0	100	100
Average	**4**	**18**	**4**	**62**	**100**	**94**	**708**	**3**	**23**	**81**	**100**
7	2	2	2	100	100	92	24	2	8	100	83
8	2	2	2	100	100	100	76	2	3	100	100
9	3	3	3	100	100	100	4	2	50	67	100
10	2	2	2	100	100	100	8	2	25	100	100
11	2	2	2	100	100	100	3	2	67	100	100
12	2	2	2	100	100	100	2	1	50	50	100
13	2	2	2	100	100	96	5	1	20	50	100
14	2	2	2	100	100	99	3	2	67	100	100
Average	**2**	**2**	**2**	**100**	**100**	**98**	**16**	**2**	**36**	**83**	**98**
15	2	2	2	100	100	100	2	2	100	100	100
16	2	2	2	100	100	100	2	2	100	100	100
17	2	2	2	100	100	100	1	1	100	50	100
18	2	2	2	100	100	96	1	1	100	50	100
19	2	2	2	100	100	100	2	2	100	100	100
20	2	2	2	100	100	100	2	2	100	100	100
21	2	2	2	100	100	100	2	2	100	100	100
22	2	2	2	100	100	75	1	1	100	50	100
23	4	4	4	100	100	100	4	4	100	100	100
24	2	2	2	100	100	100	2	2	100	100	100
25	2	2	2	100	100	86	1	1	100	50	100
26	2	2	2	100	100	87	1	1	100	50	100
27	5	5	5	100	100	100	5	5	100	100	100
28	2	2	2	100	100	100	1	1	100	50	100
29	2	2	2	100	100	74	2	2	100	100	99
30	2	2	2	100	100	88	1	1	100	50	100
31	2	2	2	100	100	100	2	2	100	100	100
32	2	2	2	100	100	100	2	2	100	100	100
33	2	2	2	100	100	84	1	1	100	50	100
34	6	6	6	100	100	100	6	6	100	100	100
35	6	6	6	100	100	100	6	6	100	100	100
36	6	6	6	100	100	100	6	6	100	100	100
37	6	6	6	100	100	100	6	6	100	100	100
Average	**3**	**3**	**3**	**100**	**100**	**95**	**3**	**3**	**100**	**83**	**100**

TABLE III
LEARNING FROM ONE EXAMPLE VERSUS MULTIPLE EXAMPLES

indeed edited similarly in the next version but not included in the known method pair, we extend the oracle set to include it.

Table III shows the results. On average, learning from one example has lower precision and recall when looking for edit locations as compared to learning from two examples, but has higher accuracy when suggesting edits for correctly identified locations. Several reasons explain these results.

- Inferring a common edit context from two examples results in a mix of concrete and abstract identifiers that generalizes identifiers only when necessary. Using the partially abstract context to search is more precise than using a fully abstract context, which matches more contexts.
- Using two examples reduces the edit to a common subset, so the derived edit is more likely not to be 100% accurate for any target location, where some extra specific edits are needed.
- LASE includes all nodes transitively depended on by any edited node in the inferred context, so it derives a more precise context as compared to SYDIT's context, which is based on direct dependence relations.
- LASE matches context differently than SYDIT.

	P(%)	R(%)	A(%)
LASE	94	100	96
LASE $_{AbsAll}$	75	100	96
LASE $_{SigCon}$	98	60	100
LASE $_{SigAbs}$	78	88	97
LASE $_{DirDep}$	94	100	96
LASE $_{Sydit}$	74	82	99

TABLE IV
COMPARISON BETWEEN LASE AND ITS VARIANTS

In order to explore how each factor affects LASE's effectiveness, we explored variations of LASE, and evaluated their average precision, recall, and accuracy on all 37 edits, as summarized in Table IV.

LASE $_{AbsAll}$ differs from LASE by abstracting all identifiers instead of only abstracting identifiers when necessary. Therefore, LASE $_{AbsAll}$'s inferred context is more general than LASE's and it matches more methods, causing more false positives and lower precision.

LASE $_{SigCon}$ differs from LASE by learning from a single example and using all concrete identifiers. Therefore, LASE $_{SigCon}$'s inferred edit is very specific to the example and thus has higher precision and accuracy. However, LASE $_{SigCon}$'s derived context is too specific to reveal more unknown edit locations. Its average recall is just 60% with many false negatives. In many cases, LASE $_{SigCon}$'s context can only find the method from which it is inferred but cannot detect any other possible edit location. In contrast, LASE has 100% recall on these examples.

LASE $_{SigAbs}$ differs from LASE $_{SigCon}$ by abstracting all identifiers. Unsurprisingly, it derives a more general context which matches more methods, suggesting edit locations with lower precision and higher recall, and applying edits with lower accuracy.

LASE $_{DirDep}$ differs from LASE by only using direct dependence relations to include unchanged nodes for edit context, instead of using transitive closure of dependence relations. The comparison shows that excluding the extra dependences in edit context does not affect precision, recall, or accuracy.

LASE $_{Sydit}$ differs from LASE $_{SigAbs}$ by using SYDIT's context matching algorithm to search for locations instead of MCESE. The comparison shows that LASE $_{SigAbs}$ has a higher precision, higher recall, and lower accuracy, indicating that MCESE is more flexible when matching contexts.

To sum up, compared with learning from one example, LASE's new algorithms for deriving a common edit and context from multiple examples and generalizing identifiers only when necessary are critical to accurately suggesting correct edit locations with high precision and recall. Although LASE sometimes sacrifices edit application accuracy for that improvement, detecting missed edit locations will improve code quality and save developers time and effort.

VIII. RELATED WORK

This section describes related work on programming by demonstration, code search, edit location suggestion, and automated code repair.

Example-based Program Migration and Correction. The most closely related work automates API migration [14]. This work detects client differences in API usage from multiple instances, creates an edit script (called a semantic patch in their work) for the correct usage, and transforms programs to use updated APIs. This approach focuses on performing stylized API usage correction and cannot always correctly position edits in target contexts, because they compute edit positions without considering control or data dependence constraints that the edits have on their surrounding contexts [23]. In comparison, LASE supports much more expressive, customizable transformations and uses program dependence analysis to correctly position the edits. Their evaluation is limited to understanding a handful of API usage changes and inferring the reasons for certain bug fixes, whereas our evaluation actually *uses* the inferred systematic edits to search for edit locations, applies a customized edit to each location, and measures LASE's effectiveness systematically in terms of precision, recall, and accuracy.

Our prior work, SYDIT, produces code transformation from a single example only. It does not search for edit locations and requires developers to supply target edit locations. Furthermore, we find one example is not sufficient for creating an edit script that can find other edit locations; multiple examples significantly reduce false positives and negatives.

Simultaneous text editing automates repetitive editing [24]. Users interactively demonstrate edits in one context and the tool replicates *identical lexical* edits on pre-selected code fragments. In contrast, LASE performs *similar yet different* edits using a *syntactic* context-aware, abstract transformation.

Edit Location Suggestion. Sophisticated code search takes as input queries, like def-use and method-call sequences, and may identify locations missing similar edits. Wang et al. propose a dependence query language to find code snippets that require similar edits [25]. Using PQL, developers write declarative rule-based queries to look for matching code fragments at runtime and to correct an erroneous execution on the fly [26]. LibSync helps client applications migrate library API usages by learning migration patterns [9] with respect to a partial AST with containment and data dependences. Although it suggests example API updates, it is *unable* to transform code. All these tools suggest edit locations, but developers must manually apply edits, whereas LASE automatically finds locations and applies edits.

Program Synthesis. Recent work on synthesis learns from examples or specifications and then automatically synthesizes a program in a domain-specific language [27]. Researchers have applied this approach to string manipulation macros, table transformation in Excel spreadsheets, geometry construction, and programs. However, none of the synthesis approaches apply to editing general purpose programs such as Java.

Automated Code Repair. Automatic program repair generates candidate patches and checks correctness using compilation and testing [13]. For example, Weimer et al. generate candidate patches by replicating, mutating, or deleting code *randomly* from the existing program. They do not infer edits from multiple edit examples, nor do they systematically apply an edit to multiple places. Specification-based program repair such as AutoFix-E [28] generates simple bug fixes from manually prescribed contracts. Since LASE automatically infers program transformations from multiple edit examples, it handles a much larger space of systematic edits than these tools.

IX. Discussion and Conclusions

LASE is the first tool to learn nontrivial partially abstract, data and control context-aware edits from *multiple* edit examples, to automatically search for edit locations, and to apply customized edits to the locations. Other tools either are limited to much simpler changes, or only suggest locations, or only perform edits. To learn and apply a systematic edit, users must provide a set of examples whose common edit operations capture the skeleton of portable edits. For example, a library component developer could use LASE to automate API usage updates in client applications by shipping an edit script learned from examples that already migrated to the new API and tested by the developer.

LASE matches the context of a learned edit script against all methods in the entire program reasonably fast—in our study, matching a learned context against *all* methods in each subject program took only 28 seconds on average. Our subject programs (jEdit, Eclipse jdt.core, Eclipse compare, Eclipse core.runtime, and Eclipse debug) are representative of medium and large size Java projects.

Currently, LASE focuses on single method updates, but it may be possible to generalize this approach for higher-level changes, such as class hierarchy changes. These activities make coordinated edits to multiple classes and methods. For example, an *extract super class* refactoring moves a set of related fields and methods from subclasses to a super class. This type of functionality will require more sophisticated context representations and matching algorithms.

LASE customizes edits by establishing a mapping between symbolic identifiers and concrete identifiers from the target context. However, it cannot always find such mapping for code that only exists in the new version. To overcome this limitation, we plan to investigate algorithms to synthesize identifier names based on existing concrete identifiers in the target context.

We believe that users may want to correct inferred edit scripts or write their own from scratch, although we did not evaluate it here. Users would then choose which names to abstract or which parts of the edit are more widely applicable. Given this modified script, LASE can verify it against the examples. It can also compare the suggested edits from the original script with those from the modified version to show users effects of a particular modification.

LASE aims to relieve some of programmers' burden in manual and tedious application of similar changes, to eliminate errors due to copy-and-paste practices, to find and transform locations that developers may have otherwise missed.

In summary, LASE provides needed functionality that helps developers locate code requiring similar edits and apply these edits automatically. By learning the common edit operations

and context from multiple examples and matching context against all method locations in the entire program, LASE achieves high precision, recall, and edit application accuracy. In our extensive study of repetitive bug fixes, it achieves 95% precision, 88% recall, and 91% accuracy. Furthermore, LASE found and transformed several locations that developers missed. This approach for program evolution opens a new approach that gives developers tools they need when adding features and fixing bugs.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, CCF-1043810, SHF-0910818, and CCF-0811524 and by a Microsoft SEIF award. We thank Jihun Park for sharing supplementary patch data.

REFERENCES

[1] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.

[2] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 315–324.

[3] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 481–490.

[4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *ACM Symposium on Operating Systems Principles*, 2001, pp. 57–72.

[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code." in *OSDI*, 2004, pp. 289–302.

[6] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081755

[7] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *USENIX Security Symposium*, 2008, pp. 379–394.

[8] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[9] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321.

[10] 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. New York, NY, USA: ACM, 2011, pp. 329–342.

[11] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI*, 2011, pp. 389–400.

[12] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 87–102.

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

[14] J. Andersen and J. L. Lawall, "Generic patch inference," in *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 337–346.

[15] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, *Learning repetitive text-editing procedures with SMARTedit*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 209–226.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[17] A. Lozano and G. Valiente, "On the maximum common embedded subtree problem for ordered trees," in *In C. Iliopoulos and T Lecroq, editors, String Algorithmics, chapter 7. Kings College London Publications*, 2004.

[18] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *MSR '12: The 9th IEEE Working Conference on Mining Software Repositories*, 2012, pp. 40–49.

[19] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *ESEC/FSE-20: ACM SIGSOFT the 20th International Symposium on the Foundations of Software Engineering*, 2012, to appear, p. 11 pages.

[20] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling—tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, p. 18, November 2007.

[21] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[22] G. W. Adamson and J. Boreham, "The use of an association measure based on character structure to identify semantically related pairs of words and document titles," *Information Storage and Retrieval*, vol. 10, no. 7-8, pp. 253–260, 1974.

[23] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch infere," in *ASE 2012: The 27th International Conference on Automated Software Engineering*, 2012 (to appear), p. 4 pages (Tool Demo Paper).

[24] R. C. Miller and B. A. Myers, "Interactive simultaneous editing of multiple text regions," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 161–174.

[25] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 457–466.

[26] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 365–383.

[27] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, ser. PPDP '10. New York, NY, USA: ACM, 2010, pp. 13–24.

[28] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 61–72.