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†
Austin, US* Seattle, US†
mengna09@cs.utexas.edu, miryung@ece.utexas.edu, mckinley@microsoft.com

***Abstract*—Adding features and fixing bugs often require *systematic edits* that make similar, but not identical, changes to many code locations. Finding all the relevant locations and making the correct edits is a tedious and error-prone process for developers. This paper addresses both problems using edit scripts learned from multiple examples. We design and implement a tool called LASE that (1) creates a context-aware edit script from two or more examples, and uses the script 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 99% precision and 89% 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. Furthermore, we confirmed with developers that LASE found edit locations which they missed. Our novel algorithm that learns from multiple examples is critical to achieving high precision and recall; edit scripts created from only one example produce too many false positives, false negatives, or both. Our results indicate 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 most structural changes involve systematic change patterns [12]. Nguyen et al. find that many bug fixes are systematic, and most occur in methods with similar functions and object interactions [21]. When an API evolves, client applications must systematically adapt by constructing new objects, passing new arguments, or replacing API calls [4]. When developers fork software, they often copy patches between products of the same family as the software evolves. For example, in a recent case study of BSD products, developers copy 11%-16% patches between OpenBSD, FreeBSD, and NetBSD [24]. In all these examples, programmers manually find many code locations and then apply similar, but not identical, edits to them one by one. This process is tedious and error-prone.

Existing tools either suggest code locations or transform 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 [5], [14], [15], [26], [30], or they identify code clones that may require similar edits [20], [21]. However, these tools do not fix programs by applying code transformations. Other tools apply code transformations, but the user must specify target locations [18]. The closest work locates and applies very simple or limited stylized changes [9], [23], [25], [29]. We refer to limited changes such as API, concurrency, or security policy corrections [2], [9], [25], and identical, lexical edits to similar text [13] as *stylized*. Andersen and Lawall [2] perform API updates, but they cannot position edits well because they do not consider data and control dependences. Given concurrency bug reports, Jin et al. insert synchronization patterns and test them [9]. The most sophisticated approach adds missing security logic [25]. None of these tools however perform *general* program transformations that both fix bugs and add functionality.

This paper describes the design and implementation of LASE (Locating and Applying Systematic Edits) to help developers evolve programs by performing general systematic edits. Developers specify two or more example methods that they edited by hand. LASE uses the examples to learn an edit script, uses the script to find other edit locations, customizes the script to each location, and applies the customized script.

From the examples, LASE infers the *most specific generalization* of the edit operations and their context, 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 examples. For instance, if two examples delete an `if` statement, but disagree on the names of types, methods, or variables in the condition, LASE abstracts the discrepant names, and if they agree on names, LASE uses the concrete names, creating a partially abstract delete operation. LASE next computes the common *context* of edit operations, i.e., other statements that constrains edits. LASE uses context to search for edit locations and position edits. For instance, if two examples insert S as the first statement in a `while` loop, the context is the `while` loop with the position of S as the loop's first child. LASE determines the largest common context of edits with a novel algorithm that combines clone detection [10], maximum common embedded subtree extraction [16], 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. We assume that methods with similar contexts require similar edits and search for methods with code that matches the script's context. For methods that match, LASE customizes the

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

edit script by making abstract names and edit positions in the script concrete based on the concrete names and code positions in the target method. LASE then applies the customized edit script and suggests the changed method to the developer for review since LASE cannot guarantee the edit is correct.

We perform a thorough evaluation of LASE and its features on 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 [22]. We evaluate LASE by learning edit scripts from the initial patches and determining if LASE correctly derives the subsequent, supplementary patches. On average, LASE identifies edit locations with 99% precision and 89% recall. The accuracy of applied edits is 91%, i.e., the tool-generated version is 91% similar to the developer's version. 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 single examples with fully abstract [18] or fully concrete names. The partially abstract context and edit scripts that LASE infers from multiple examples are critical to achieving high precision, recall, and accuracy. Furthermore, LASE identifies and correctly edits 9 locations in the oracle test suite that developers confirmed they missed.

LASE is the first tool to learn nontrivial program edits from multiple changed methods, to use scripts to find edit locations, and to perform customized program transformations at each found location. Our results show the power of automation for adding features and fixing bugs in a large code base. Tools are simply more systemic than humans.

II. MOTIVATING EXAMPLE

This section uses a motivating example drawn from revisions of org.eclipse.compare to show LASE's work flow and compare it to our prior work that learns from one example. Figure 1 shows three methods with similar changes: `mA`, `mB`, and `mC`. 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. Figure 2 shows how SYDIT [18] generates an edit script from one exemplar change and then the developer tells SYDIT *where* to apply the inferred edit script.

An edit script consists of edit operations (insert, delete, move, and update) and *context*, unmodified statements in the method on which the edit statements are control or data dependent. The context expresses constraints that other statements place on the edit and positions the edit. In Figures 2 and Figure 3, 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 from `mA`. Unfortunately with only one example, the resulting edit script

A_{old} to A_{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.}
```

B_{old} to B_{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.}
```

C_{old} to C_{new}

```
1. public void selectionChanged (SEvent event) {
2.   Iterator e = fActions.values().iterator();
3.   while(e.hasNext()){
4. -   MVAction action=(MVAction)e.next();
5. -   if(action.isSelectionDependent())
6. -     action.update();
7. +   Object next = e.next();
8. +   if (next instanceof MVAction){
9. +     MVAction action =(MVAction)next;
10.+    if(action.isSelectionDependent())
11.+      action.update();
12.+  }
13.  }
14.}
```

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

cannot find edit locations because it suffers both from *over specification* and *over generalization*.

The script learnt from `mA` includes deleting two statements (lines 3 and 4), but these operations are overly specific to `mA` and prevent the script from being applied to `mB` and `mC`. Meanwhile, the script learnt from `mA` abstracts *all* names of variables, methods, and types, used in `mA` to increase flexibility when looking for matches in a target method. For instance, abstracting variable `e` of type `Iterator` to `v$0` of type `t$0` enables `v$0` to match variables `e` in `mA` and `mC`, and `iter` in `mB`, both of which have type `Iterator`. However, abstractiing every name is too flexible and matches variables in many

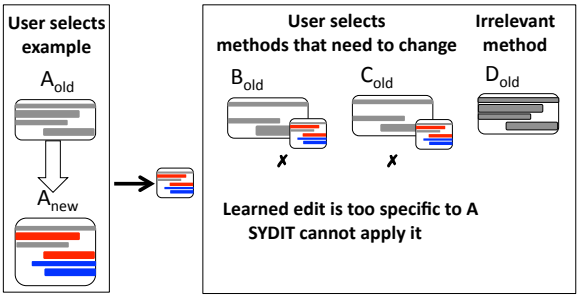

Fig. 2. SYDIT learns an edit from one example. A developer must locate and specify the other methods to change.

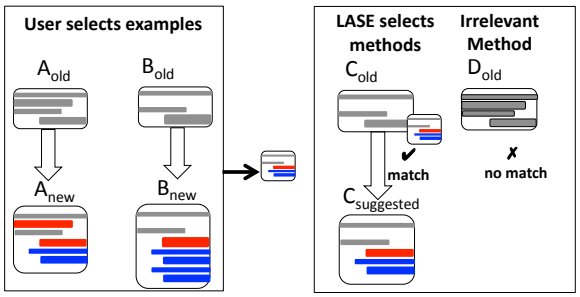

Fig. 3. LASE learns an edit from two or more examples. LASE locates other methods to change.

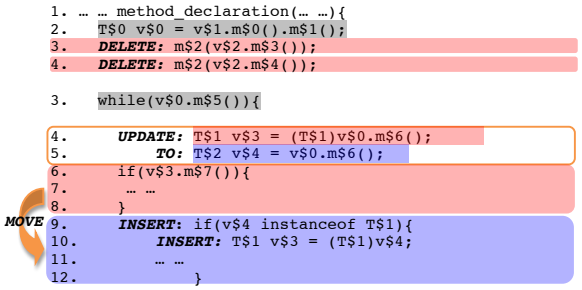

Fig. 4. Edit script from SYDIT abstracts all concrete names. Gray marks edit context, red marks deletions, and blue marks additions.

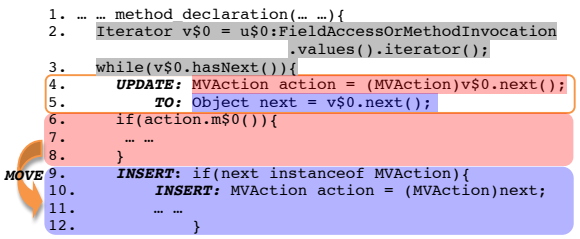

Fig. 5. Edit script from LASE abstracts code names that differ in the examples and uses concrete names for common ones. Gray marks edit context, red marks deletions, and blue marks additions.

unrelated methods, and consequently would incorrectly apply edit scripts to too many methods.

This paper seeks an edit script that serves double duty, both finding edit locations and accurately transforming the code. LASE learns from two or more example edits 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 write and edit code, we think providing multiple examples is a natural interface.

Figure 3 shows the work flow of LASE. The developer specifies two exemplar changed methods, `mA` and `mB`. LASE infers the edit script shown in Figure 5 from the examples. It uses the edit script to find matching locations, specializes the script to each location, applies the result, and suggests the transformed code to the developer. Using multiple examples requires new algorithms to identify common changes and context, and to abstract or omit differences. None of these algorithms are necessary when learning from a single example.

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 extracts the context for each common edit and then determines the largest common edit-relevant context. This algorithm combines clone detection, maximum common embedded subtree extraction on the Abstract Syntax Tree (AST), and dependence analysis. Finally, if type, method, and variable names agree, LASE uses these concrete names, otherwise LASE abstracts the discrepant names in both edit operations and context. For example in Figure 5, LASE uses `Iterator` because it is common to `mA` and `mB`. Since field access `fActions` in `mA` and method invocation `getActions()` in `mB` match but differ, LASE generalizes them to an abstract name `u$0:FieldAccessOrMethodInvocation`.

III. APPROACH

This section summarizes LASE's three phases and formalizes our terminology. The following sections then describe each phase in detail. We represent edit operations and context with Abstract Syntax Trees (AST).

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: $E_i = [e_1, e_2, \ldots, e_k]$ where e_i is an insert, delete, move, or update operation 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) *Generalizing Identifiers in Edit Operations.* When a common edit operation $e \in E_c$ uses distinct type, method, and variable names in different methods, LASE replaces the concrete names with abstract names, resulting in E. Otherwise, it uses the original concrete names.
4) *Extracting Common Edit Context.* LASE finds the largest common context C relevant to E using code clone detection, maximum common embedded subtree extraction,

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

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

and dependence analysis. LASE abstracts names in the context C as well as the edits E.

The result of this process is 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 $\texttt{m}_f \in M_f$, LASE specializes Δ_P to $\texttt{m}_f$ by mapping abstract names and abstract edit positions in Δ_P to concrete ones in m_f, producing Δ_f. LASE applies this concrete edit script to $\texttt{m}_f$ and suggests the resulting method $\texttt{m}_f\texttt{'}$ to the developer.

IV. Phase I: Learning from Multiple Examples

A. Generating Syntactic Edits

For each exemplar changed method $\texttt{m}_i \in M$, LASE uses Fluri et al.'s AST differencing algorithm [6] to compare the AST of $\texttt{m}_i$'s old and new versions and create 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.

Although insert and delete are sufficient to describe all edits, update and move operations link dependent edit operations and make the edit easier to apply.

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 [8], as shown in Equation (1).

We do not require exact equivalence between edit operations because systematic edits are not necessarily identical. We define the comparison function $equivalent(e_p, e_q)$ in two ways: *concreteMatch*(e_i, e_j, t_s) and *abstractMatch*(e_i, e_j).

LASE first applies *concreteMatch*(e_i, e_j, t_s) to compare e_i and e_j using their edit types and *labels*. Labels are string representations of AST nodes with identifiers and operators. If two operations have the same node type and their labels' bigram string similarity [1] is above the threshold t_s, the function returns true. We use $t_s = 0.6$ to include more matches. LASE also matches AST node types inexactly, tolerating mapping `return` statement to `expression` statement, and `while` to `for` to `do`.

If LASE fails to find any common edit operation between two edits with *concreteMatch*, it applies *abstractMatch*(e_i, e_j), which converts all concrete names of types, methods, and variable to abstract identifiers `t$`, `m$`, and `v$`. If two operations have the same edit type and their labels' abstract representations match, *abstractMatch* returns true. Other matching heuristics may also perform well, such as abstracting names one at a time, but we did not explore them.

The result of matching is a list of concrete edit operations that are *equivalent* and common to all exemplar methods, but their identifier names and AST types may not match.

C. Generalizing Identifiers in Edit Operations

LASE next generalizes names as needed. When all edit operations agree on a concrete name, LASE uses the concrete name. If one or more edit operations use a different 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 discrepant variable names `e` vs. `iter`, it generalizes them by creating a fresh abstract identifier `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. It 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. If some subsequent common edit operations are inconsistent with the current mappings, LASE omits them, resulting in 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 name usage, AST subtree extraction, and dependences.

1) Finding Common Text with Clone Detection: For all matched edits $\{E_1, E_2, \ldots, E_n\}$, LASE extracts relevant unchanged *context* code. LASE first finds common text by dividing the methods with edits into three parts. Given matching edits E_1 on AST nodes $n_1, n_2 \in m_1$ and E_2 on $n_1', n_2' \in m_2$, let n_1 precede n_2 in m_1. LASE divides the methods into code preceding n_1 and n_1', code between the two matching nodes, and code after them. LASE next compares each of these three code segments in m_1 and m_2 with a clone detector [10]. This step reveals all possible common *text* shared between each pair of methods. This common text over approximates context. The steps below refine the context based on name mapping, common embedded subtree extraction, and dependence analysis.

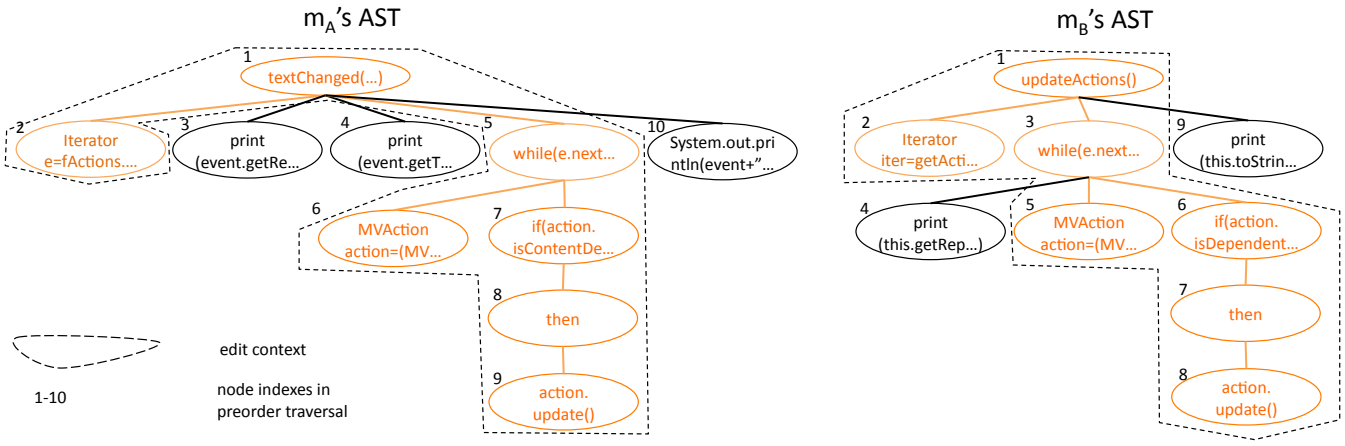

Fig. 6. $\mathtt{m}_A$'s and $\mathtt{m}_B$'s AST

2) Generalizing identifiers: Because clone detection uses text similarity, it does not guarantee that type, method, and variable names in one method are mapped consistently in other methods. LASE collects all name mappings between the two methods' clone pairs. If there are conflicting mappings, LASE retains the context statements for the most frequent mappings and excludes any inconsistent statements from the context. If different concrete names map consistently with each other, LASE generalizes them to a fresh abstract name and substitutes it for the names in all context statements and edit operations to create an abstract common context C_{abs}.

3) Extracting Common Subtree(s) with MCESE: Because clone detection uses text matching, the AST structure of two nodes may not match, e.g., two matching nodes may have different parent nodes. To solve this problem, LASE uses an off-the-shelf Maximum Common Embedded Subtree Extraction (MCESE) algorithm [16] to find the largest common forest structure, as shown in Equation 2. This algorithm traverses each AST in pre-order, indexes nodes, and encodes the tree structure into a node sequence. By computing the longest common subsequence between the two sequences and reconstructing trees from the subsequence, LASE finds the largest common embedded subtree(s), C_{sub}. It then excludes all the other statements from the context.

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

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 "−" 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, 2, -2, 5, 6, -6, 7, 8, 9, -9, -8, -7, -5, -1] of s and [1, 2, -2, 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 $s[0]$ and -$s[0]$), 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 puts unnecessary constraints on potential edit locations, causing false negatives during edit location search. LASE thus further refines the extracted context based on control and data dependences.

LASE performs control and data dependence analysis and then determines direct and transitive dependences between edit operations and context statements in each changed method. For each edit operation, LASE unions all the unchanged AST statements that are the source of dependences with the edit as relevant context. Finally, it intersects the identified edit-relevant context in each method to produce C_{dep}. If $C = C_{sub} \bigcap C_{dep}$ is not empty, LASE sets the context of E to C, omitting unrelated statements. If C_{dep} is empty, LASE sets $C = C_{sub}$, since matching an empty context is not useful for detecting edit locations. E and C define a *partially abstract, context-aware edit script*, Δ_P, where each edit operation in E is positioned 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 type, method, and variable names. When LASE matches C with a method `m`, it matches concrete names exactly and abstract names by type or AST node. 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 from Section IV-D to find the maximum common context between C and `m`, but redefines the $equivalent(i, j)$ function to compare concrete names based on string equality and abstract names based on identifier type and AST node type. If each node of the common context C matches a node in method `m`, LASE records name mappings between them and then suggests `m` as an edit location $\mathtt{m}_f$.

This algorithm for identifying edit locations is simple because the context in the edit scripts precisely encodes the exact and flexible matching criteria.

VI. PHASE III: APPLYING THE EDIT

To apply the edit to a suggested location $\mathtt{m}_f$, LASE must customize the edit Δ_P for $\mathtt{m}_f$. For this process, we slightly modify the edit customization and edit application algorithms that we introduced previously [18]. The customization algorithm replaces all abstract names in Δ_P with the corresponding concrete names from $\mathtt{m}_f$ based on the name mappings established in Phase II. LASE retains all the concrete names in Δ_P. In addition, LASE positions each edit operation concretely in the target method in terms of AST node positions. LASE uses the the data and control dependences contained in the edit script and computes dependences in the target method. It uses the dependences to maintain correct and consistent relationships between identifiers and statements as described in our prior work [18]. The result is Δ_f, which fully specifies each edit operation as an AST modification with concrete labels and node positions. LASE applies this customized, concrete edit script and suggests the resulting version to developers.

VII. EVALUATION

This section evaluates LASE's precision and recall when finding correct edit locations and its accuracy when applying edits. We use two oracle test suites. One test suite consists of multiple systematic edits that fix the same bug in multiple commits, drawn from two open-source programs, Eclipse JDT and Eclipse SWT. The other one contains 37 systematic edits from five Java open-source programs (jEdit, Eclipse jdt.core, Eclipse compare, Eclipse core.runtime, and Eclipse debug). We explore sensitivity to (1) multiple examples versus one example, (2) example choice, and (3) strategies for identifier abstraction and context. LASE matches context against *all* methods in the entire program reasonably quickly, taking 28 seconds on average on the five open-source subject programs, which are representative of medium and large Java projects.

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

Index	Bug (patches)		$\mathtt{m}_i$	Edit Location Σ	✓	P %	R %	A %	Operations E	C	A_E %
1	73784	(1)	4	4	4	100	100	53	7	2	29
2	82429	(2)	16	13	12	92	75	81	9	9	100
3	114007	(3)	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	(5)	7	7	7	100	100	100	34	34	100
*8	129314	(3)	3	4	4	100	100	100	2	2	100
9	134091	(4)	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)	9	9	9	100	100	83	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	(3)	7	9	9	100	100	78	4	4	100
17	97981	(2)	4	3	3	100	75	100	3	3	100
Average			**6**	**5**	**5**	**98**	**93**	**89**	**10**	**9**	**91**
18	74139	(3)	5	5	5	100	100	100	1	1	100
19	76391	(3)	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	15	100	100	95	3	3	100
Average			**5**	**5**	**4**	**100**	**78**	**97**	**4**	**4**	**94**
Total Average			**6**	**5**	**5**	**99**	**89**	**91**	**8**	**7**	**92**

* LASE suggests edits *missed* by developers.

A. Precision, Recall, and Accuracy with an Oracle Data Set

To measure precision, recall, and accuracy, we use an oracle test suite based on edits to Eclipse JDT and Eclipse SWT identified by work on supplementary bug fixes [22], [24]. They find bug fixes spanning multiple commits to understand characteristics of incomplete or incorrect bug fixes, using the bug ID and clone detection. The work illustrates that developers miss locations that they need to change when initially fixing a bug, further motivating our work.

We select systematic edits from these programs. If a bug is fixed more than once and there exist clones of at least two lines in 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, as shown in Table VII, where the first two rows are from JDT, while the rest are from SWT. The table groups the examples into two sets based on whether LASE refines 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 $C = C_{sub}$.

We use these patches as an oracle test suite for correct systematic edits and test if LASE can produce the same results as the developers given the first two fixes in each set of systematic fixes. Since the developers may not be perfect, there may be incorrect edits or missing edits for which we cannot control. Indeed, we confirmed with developers that LASE found 9 methods in 3 fixes (starred in Table VII) and applied correct edits that they missed! When LASE produces the same results as developers do in latter patches, it indicates

that LASE will help programmers detect edit locations earlier, reduce errors of omission, and make systematic edits.

We give LASE as input two random changed methods in the first patch. If there is only one changed method 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 the customized edit to suggest a new version.

Table VII shows the results. The table lists the **Bug** identifier, the number of **patches**, and number of methods $\mathbf{m}_i$ 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 correctly identified edit locations compared to all found locations; recall **R**: the percentage of correct locations out of all expected locations; and accuracy **A**: the syntactic similarity between the tool-suggested version and the expected version, only for edited methods. The **Operation** columns 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 operations correctly inferred by LASE; and $\mathbf{A}_E$: the percentage of operations correctly inferred over expected operations.

LASE locates edit positions with respect to the oracle data set with 99% precision, 89% recall, and performs edits with 91% accuracy. We check accuracy by visual inspection and compilation. Most of the inferred edits are nontrivial and LASE handles these cases well. For instance, edit case 7 requires 34 operations. LASE correctly infers all 34 of them, correctly suggests 7 edit locations, and correctly applies customized edits with 100% accuracy.

In three edit cases (8, 16, and 24), LASE suggests 9 edits that developers *missed*. Note that the number of methods correctly identified for each is larger than the number of methods developers changed. We confirmed all these omission errors with the Eclipse developers and mark the test cases with an asterisk in Table VII. These results indicate that LASE will help developers make systematic edits consistently and help 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 edits and LASE cannot suggest edits specific to a single location. For instance in edit case 2, LASE infers all 9 edit operations shared among repetitive fixes for the same bug, but it misses some specific edits and does not achieve 100% accuracy. Second, abstract names may not have corresponding concrete names in the edit location. For example, if an abstract name is only used by inserted statements, LASE cannot decide how to concretize it. Third, based on string similarity, LASE's AST differencing algorithm cannot always infer edits operations correctly. For instance, if `trailingComments != null` is updated to `trailingPtr >= 0` in one method, and `rComments != null` is updated to `rPtr >=0` in another method, the inferred operations for the later includes an insert and delete operation pair since the two strings are not similar enough for LASE to infer an update operation. When LASE compares an update operation to the insert and delete operations, the edit types do not match and it does not extract a common edit operation. 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 to choose the best one. Although each of these problems occurred, none occurs frequently.

TABLE II
LASE'S EFFECTIVENESS WHEN LEARNING FROM MULTIPLE EXAMPLES

	# of exemplars	P %	R %	A %
Index 4	2	100	51	72
	3	100	82	67
	4	100	96	67
	5	100	100	67
Index 5	2	100	80	100
	3	100	84	100
	4	100	91	100
Index 7	2	100	83	100
	3	100	84	100
	4	100	88	100
	5	100	92	100
	6	100	96	100
Index 12	2	78	90	85
	3	49	98	83
	4	31	100	82
Index 13	2	100	100	95
	3	100	100	94
	4	100	100	93
	5	100	100	91
Index 19	2	100	66	100
	3	100	94	100
	4	100	100	100
	5	100	100	100
Index 23	2	100	72	100
	3	100	88	100
	4	100	96	100

The number of exemplar edits influences effectiveness. To determine how sensitive LASE is to the number and choice of exemplar edits, we randomly pick 7 cases in the oracle data set and enumerate subsets of exemplar edits, e.g., all pairs of two exemplar methods. We evaluate the precision, recall, and accuracy for each choice of exemplars and calculate the average for each cardinality to determine how sensitive LASE is to the choice and number of exemplar edits.

Table VII-A shows that precision **P** does not change as a function of the number of exemplar edits for these examples, except for case 12, where two exemplars are the most accurate. Recall **R** is more sensitive to the choice and number of exemplar edits, increasing as a function of exemplars. The more exemplar edits provided, the less common context is likely to be shared among them, and the easier it is to match. However, the context will still be specific, resulting in high precision. Precision can go down when more diverse examples are given, but this case (case 12) only occurred once in these tests.

In theory, Accuracy **A** can vary inconsistently with the number of exemplar edits, because it strictly depends on the similarity between edits. For instance, when exemplar edits are diverse, LASE extracts fewer common edit operations, which

lowers accuracy. When exemplar edits are similar, adding exemplar methods may not decrease the number of common edit operations, but may induce more identifier abstraction and result in a more flexible edit script, which increases accuracy.

B. Sensitivity of Edit Scripts

This section explores how sensitive the results are to edit script features. We first compare learning from multiple examples to a single example. We use LASE to generate edit scripts from example pairs and SYDIT [18] to generate edit scripts from single examples. SYDIT only learns from one example and abstracts all names. It does not find edit locations but relies on developers to choose locations. We use LASE to find locations for SYDIT's scripts and apply edits. The experiments show that using multiple examples finds locations with higher precision and recall than using one example, and motivates using two or more examples.

We measure precision and recall of edit location suggestion and accuracy of edit application on the SYDIT test suite [18] of 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 method pair demonstrates at least one similar edit operation and the two methods are at least 40% similar according to the syntactic program differencing (see Section IV-A). We remove the simple cases, e.g., edits on initially empty methods or only one statement, resulting in 37 pairs. 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 all found locations. If a location is indeed edited similarly in the next version but not in the known pairs, we include it in the oracle set.

Table VII-B 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 context from two examples results in a mix of concrete and abstract identifiers. Searching with a partially abstract context is more precise than a fully abstract context, which matches more methods. The partially abstract context recalls more than a concrete context does, which matches fewer methods.
- Using two examples reduces the edit to a common subset, so the derived edit is likely to be less accurate for any one target location, since it may lack some edit operations.
- LASE includes all nodes transitively depended on by any edited node in the inferred context, deriving a more precise context as compared to SYDIT's context, which is based on direct dependence relations.
- LASE matches context differently than SYDIT.

Table IV shows the average sensitivity of LASE to the abstraction and context algorithms by comparing their average precision, recall, and accuracy on all 37 systematic edits.

TABLE III
LEARNING FROM ONE EXAMPLE VERSUS MULTIPLE EXAMPLES

ID	m_i	Two Examples Σ	✓	P %	R %	A %	One Example 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 IV
COMPARISON BETWEEN LASE AND ITS VARIANTS

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

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 context inferred by LASE and it matches more methods, causing more false positives and lower precision.

LASE $_{SigCon}$ learns from a single example and uses all concrete identifiers which makes LASE $_{SigCon}$'s inferred edit very specific to the example. Consequently, LASE $_{SigCon}$'s derived context is too specific to find all 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 and cannot detect any other edit location. In contrast, LASE has 100% recall on these examples.

LASE $_{SigAbs}$ learns from a single example and abstracts all names. The resulting context is too general and it suggests edit locations with lower precision and higher recall, but applies edits with lower accuracy than context from LASE$_{SigCon}$.

LASE $_{Sydit}$ differs from LASE by using SYDIT's context matching algorithm to search for locations instead of MCESE. SYDIT's algorithm assumes that developers specified the target method and it matches. The comparison shows that LASE $_{Sydit}$ results in lower precision and recall, but higher accuracy. MCESE is much better at identifying the correct locations.

LASE $_{DirDep}$ uses direct dependence relations to include unchanged nodes for edit context, instead of using the transitive closure of dependence relations. In many cases, this algorithm produces the same context as LASE, but even when LASE $_{DirDep}$ is smaller, excluding the extra dependences in edit context does not affect precision, recall, or accuracy. This result suggests that the direct control and data dependences are often sufficient to position the edit relative to all the dependences.

To sum up, compared with learning from one example, LASE's new algorithms that learn from multiple examples are critical to finding correct edit locations. LASE's context discriminates well and thus finds the correct edit locations with high precision and recall. Since programmers struggle a lot with errors of omission, if need be, LASE should sacrifice edit application accuracy to increase precision and recall. Once LASE identifies correct locations, if the applied edit is not fully correct, developers still have the opportunity to review and correct the code. Finding and suggesting missed edit locations is thus the most important feature of LASE.

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 [2]. This work detects client differences in API usage from multiple instances, creates an edit script (referred to as a semantic patch) for the correct usage, and transforms programs to use updated APIs. They focus on 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 on the unchanged surrounding context [3]. In comparison, LASE supports more expressive, customizable transformations and uses context to correctly position edits. Their evaluation is limited to understanding a handful of API usage changes, whereas our evaluation *uses* the edit scripts 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 [18]. It requires developers to supply target edit locations, whereas LASE finds the target locations. Furthermore, we find one example is not sufficient to create an edit script that can find other edit locations. The partial abstraction LASE obtains from multiple examples significantly reduces false positives and negatives.

Simultaneous text editing automates repetitive editing [19]. 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, such as 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 [27]. In PQL, developers write declarative rule-based queries to look for matching code fragments and correct an erroneous execution on the fly [17]. LibSync helps client applications migrate API calls by learning migration patterns with respect to a partial AST with control and data dependences [20]. Although LibSync suggests example API updates, it *does not* transform code. These tools suggest edit locations, but developers must manually apply edits.

Program Synthesis. Recent work learns from examples or specifications and then automatically synthesizes a program in a domain-specific language [7]. Researchers have applied this approach to string manipulation macros, table transformation in Excel spreadsheets, geometry construction, and programs. However, none edit general purpose programs.

Automated Code Repair. Automatic program repair generates candidate patches and checks correctness using compilation and testing [29]. 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. FixMeUp inserts missing security checks interprocedurally using a specification, but these additions are very specific and stylized [25]. Kim et al. [11] use ten common bug fix patterns inferred from Eclipse JDT's version history to improve the patch suggestions of Weimer et al. [29]. However, the patterns are created manually. LASE automates bug fix pattern inference and reduces manual effort significantly. Given concurrency error reports, Jin et al. select from and test a handful of synchronization patterns to fix them [9]. They insert appropriate synchronization into a compiler intermediate representation, whereas LASE directly modifies the program. Although LASE does not coordinate cross method or interprocedural changes, it handles a much larger class of edits than all these tools.

IX. Discussion and Conclusions

Our results reveal some limitations and directions for future work. For example, LASE enforces a total order on edit operations, but the edits themselves define a partial order. As a result, LASE may miss locations with different statement orders but the same semantics. LASE applies an edit script to a method exactly once. If the method location requires multiple applications of the script, LASE will only suggest

a partially correct transformed program. Another limitation is that LASE only learns from examples changed in syntactically similar ways. When developers make semantically similar but syntactically different changes to examples, LASE cannot derive an edit script. In addition, LASE performs intraprocedural analysis within a single method, but some changes require moving code from one method to another or coordinating changes to multiple methods, such as inserting synchronization.

To overcome some of these limitations, a future tool could help users directly modify and apply inferred edit scripts as necessary. Another direction would be to add interprocedural analysis to coordinate cross-method updates, although our experience indicates that combining interprocedural analysis and the expressiveness of general purpose edits is a very hard problem. A first approach may be to leverage refactoring and bug fix patterns from prior work.

In summary, LASE is the first tool to learn edits from *multiple* examples, automatically find appropriate edit locations, and apply customized edits to these locations. The edits are general-purpose and partially abstract. They include context, which consists of unchanged nodes that position edits based on data and control dependences. Other tools are either limited to much simpler changes, or only suggest locations, or only perform stylized edits. Our results show that LASE has very high recall, precision, and accuracy across a range of real-world nontrivial bug fixes and feature additions. Furthermore, LASE found edits that developers confirmed they missed, suggesting this type of tool will help programmers.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, CCF-1043810, SHF-0910818, CCF-1018271, CCF-0811524, and a Microsoft SEIF award. We thank Jihun Park for sharing supplementary patch data.

REFERENCES

[1] 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*, 10(7-8):253–260, 1974.

[2] J. Andersen and J. L. Lawall. Generic patch inference. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, 2008.

[3] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo. Semantic patch inference. In *International Conference on Automated Software Engineering*, page (Tool Demo Paper), 2012.

[4] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ACM International Conference on Software Engineering*, pages 481–490, 2008.

[5] 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*, pages 57–72, 2001.

[6] 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*, 33(11):18, November 2007.

[7] S. Gulwani. Dimensions in program synthesis. In *ACM Symposium on Principles and Practice of Declarative Programming*, pages 13–24, 2010.

[8] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *CACM*, 20(5):350–353, 1977.

[9] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Symposium on Operating System Design and Implementation*, 2012.

[10] 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*, 28(7):654–670, 2002.

[11] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *IEEE/ACM International Conference on Software Engineering (to appear)*, 2013.

[12] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ACM/IEEE International Conference on Software Engineering*, pages 309–319, 2009.

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

[14] 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 *ACM Symposium on Operating System Design and Implementation*, pages 289–302, 2004.

[15] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ACM Foundations of Software Engineering*, pages 306–315, 2005.

[16] A. Lozano and G. Valiente. On the maximum common embedded subtree problem for ordered trees. In *String Algorithmics, Chapter 7. Kings College London Publications*, 2004.

[17] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2005.

[18] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *ACM Conference on Programming Language Design and Implementation*, pages 329–342, 2011.

[19] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *2002 USENIX Annual Technical Conference*, pages 161–174, 2001.

[20] 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 *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 302–321, 2010.

[21] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ACM/IEEE International Conference on Software Engineering*, pages 315–324, 2010.

[22] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *IEEE Working Conference on Mining Software Repositories*, pages 40–49, 2012.

[23] 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 *ACM Symposium on Operating Systems Principles*, pages 87–102, 2009.

[24] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *ACM International Symposium on the Foundations of Software Engineering*, page 11 pages, 2012.

[25] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *Network and Distributed System Security Symposium*, 2013.

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

[27] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 457–466, 2010.

[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 *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[29] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *IEEE International Conference on Software Engineering*, pages 364–374, 2009.

[30] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *European Conference Object-Oriented Programming*, pages 318–343, 2009.