LASE: An Example-Based Program Transformation Tool for Locating and Applying Systematic Edits

John Jacobellis Na Meng Miryung Kim
The University of Texas at Austin
Austin, US
jwjacobellis@gmail.com, mengna09@cs.utexas.edu, miryung@ece.utexas.edu

***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 edit locations and editing them correctly is tedious and error-prone. In this paper, we demonstrate an Eclipse plug-in 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 (3) transform the code. In LASE, users can view syntactic edit operations and corresponding context for each input example. They can also choose a different subset of the examples to adjust the abstraction level of inferred edits. When LASE locates target methods matching the inferred edit context and suggests customized edits, users can review and correct LASE's edit suggestion. These features can reduce developers' burden in repetitively applying similar edits to different methods. The tool's video demonstration is available at https://www.youtube.com/watch?v=npDqMVP2e9Q.**

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. 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 [1]. When an API evolves, client applications must systematically adapt by constructing new objects, passing new arguments, or replacing API calls. 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 [2].

Our prior work introduced LASE (Locating and Applying Systematic Edits) to help developers perform systematic editing in multiple places [3]. When using LASE, developers specify two or more example methods that they edited by hand. LASE learns a *partially abstract, context-aware* edit script out of them, uses the same edit script to find other edit locations, and applies a customized edit to each location. Then developers judge for themselves whether to accept the suggested edits.

Intuitively, LASE infers edit *operations* (insert, delete, update, and move) common to selected examples and abstracts edit operations that differ between the examples. For instance, if two examples delete the same `if` statement, but disagree on specific names (variable, type, or method names), LASE creates a partially abstract delete statement in the edit script that abstracts the different names. It uses concrete identifiers if they are common among all examples. LASE next computes the *context* of inferred edit operations, which determines the relative position of edits in the method. It determines the largest common context with a novel algorithm that combines clone detection [4], maximum common embedded subtree extraction [5], and control/data dependence analysis. The result is an edit script that consists of partially-abstract edit operations and context. LASE searches for code locations that match the context of the script and then customizes the script for each found location. It replaces the abstract identifiers in the script with concrete identifiers used by the target method. It suggests the customized edit for developers' review.

This paper introduces the LASE Eclipse plug-in. Using the input selection view shown in Figure 1, a user supplies two or more training examples, which consist of the old and new version of methods. LASE visualizes a group of derived edit scripts using an *edit script hierarchy view*. In this view, the top node represents an edit script derived from *all* examples, and the bottom nodes represent edit scripts derived from each single example. The intermediate nodes represent edit scripts derived from different subsets of examples. By clicking on individual nodes, a user can see edit operations common to the selected subset. With this hierarchy view, the user is not limited to an edit script common to all given examples but have a choice of deriving a script from a selected subset. Furthermore, using the *edit operation view*, she can examine edit operations with respect to underlying abstract syntax trees and corresponding control and data flow contexts relevant to those edit operations. Once a user selects an edit script, LASE automatically searches for all target edit locations that match the context of an inferred edit script. It visualizes edit suggestions using a *code comparison view*—concrete, customized edits are shown using an Eclipse *compare* style, side-by-side view. Before approving the edit suggestion, the user can correct the suggested edit.

According to our prior evaluation 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 [3]. In future, we hope to build features of directly modifying and storing edit scripts.

II. Motivating Example and Tool Features

Suppose Bob is working on `org.eclipse.jdt` project's revision 9800. To modify the code

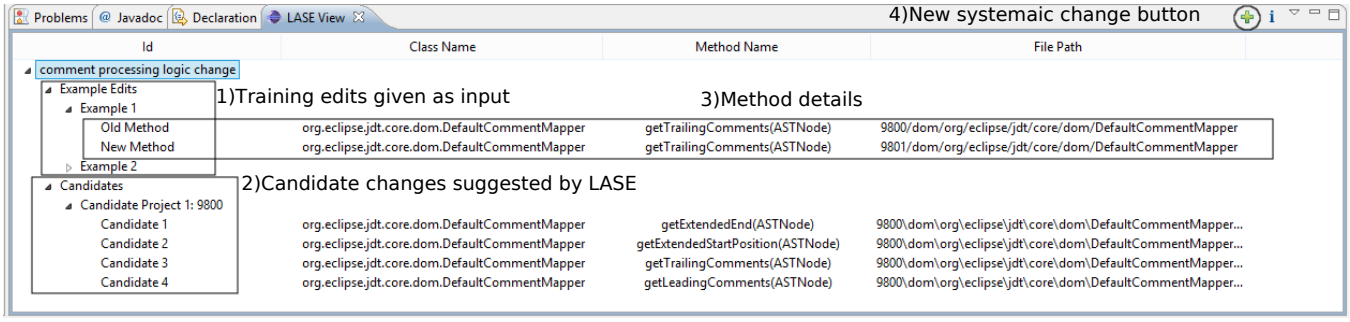

Fig. 1. LASE allows developers to provide edit examples. LASE then searches for edit locations requiring similar edits.

comment processing logic, he updates two methods `getTrailingComments` and `getLeadingComments` in `core.dom.DefaultCommentMapper`, shown in Figure 2. This change requires similar updates to both methods. In the `getTrailingComments` method, he modifies the `if` condition, modifies an assignment to `range`, and inserts a `for` loop to scan for a given AST node. In the `getLeadingComments` method, he makes a similar edit by modifying its `if` condition, an assignment to `range`, and inserting a `for` loop. After making these repetitive edits to the two methods, Bob suspects a similar edit may be needed to all methods with a comment processing logic. He uses LASE to automatically search for candidate edit locations and view edit suggestions.

Input Selection. Bob specifies the old and new versions of `getTrailingComments` and `getLeadingComments` respectively. He names this group of similar changes as a *comment processing logic change*. He then selects *an edit script generation* option to derive generalized program transformation among the specified examples.

Edit Operation View. For each example, using an *edit operation view*, Bob examines the details of constituent edit operations (*insert, delete, move* and *update*) with respect to underlying abstract syntax trees. In this view, Bob can also examine corresponding edit context—surrounding unchanged code that has control or data flow dependences on edited code. Figure 4 shows edit operations and corresponding context within method `getTrailingComments`'s abstract syntax tree. The AST nodes include both unchanged nodes and changed nodes which are the source and/or target of individual insert, delete, move, or update operations. These nodes can be expanded to show more details. The dark blue node in the figure represents the updated `if` statement. The light blue nodes represent the inserted new assignment to `range` and the new `for` loop. The beige node represents the deleted assignment to `range`, and the rest gray nodes represent unchanged context nodes relevant to edited code. The algorithm for identifying edit context is described in detail elsewhere [3].

Edit Script Hierarchy View. To create an edit script from multiple examples, LASE generalizes example edits, pair-by-pair. Figure 3 shows how we explore the space given four exemplar changed methods, LASE creates a base cluster for each method. It then compares them pair-by-pair. By merging the results of two cluster nodes, LASE generalizes common edit sequences in the edit hierarchy through a bottom up construction. For example, it merges the results of common edit sequences at Level 1 of methods (1,2) and (1,3). At level 3, it generates the generalization of methods (1,2,3) by computing the generalization of (1,2) and (1,3).

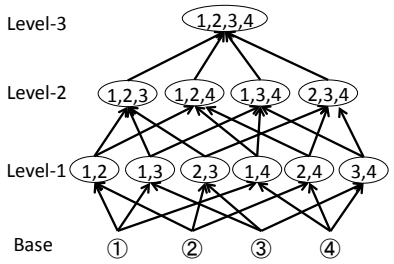

Fig. 3. Generation of an Edit Script Hierarchy

For example, by opening the *edit script hierarchy view* shown in Figure 5, Bob can examine a group of inferred edit scripts at different abstraction levels. By default, LASE uses the top node, i.e., an edit script inferred from *all* examples. By clicking a node in the edit script hierarchy, Bob may select a different subset of provided examples to adjust the abstraction level of an edit script. The script selected by a user is colored in gray, while other nodes are colored in blue. The selected script is used for searching edit locations and generating edits.

Searching for Edit Locations and Applying Customized Edits. By using a right-click menu in the input selection view, Bob begins his search for edit locations with similar context. In this case, when LASE finishes searching for the target locations, Bob sees four candidate change locations in the menu. Two of them are `getTrailingComments` and `getLeadingComments`, which are used as input examples and thus match the context of the inferred edit script—this provides an additional confirmation that the edit script can correctly describe the common edits for the two examples.

Bob then examines the edit suggestions for the first candidate method `getExtendedEnd` using the *comparison view*. See Figure 6. He sees that `getExtendedEnd` contains the same structure as his example methods. For example, the

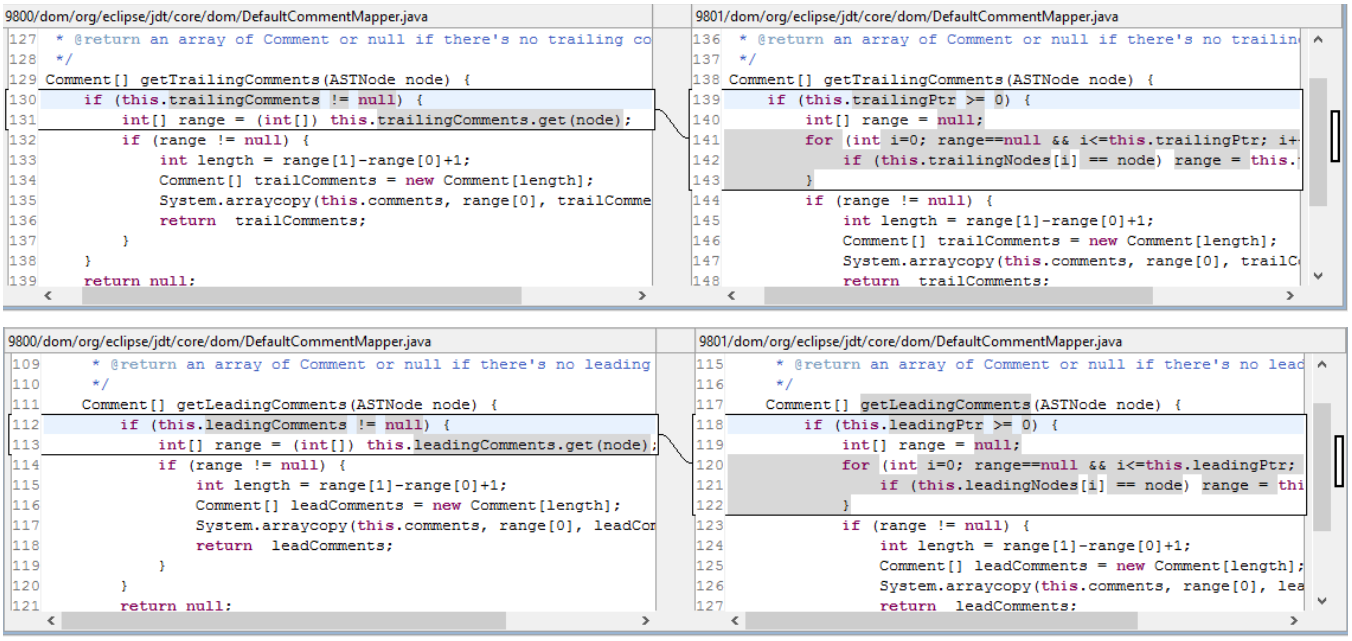

Fig. 2. A programmer makes similar but not identical edits to `getTrailingComments` and `getLeadingComments`. While `getTrailingComments` involves edits to `trailingComments` and `trailingPtr`, `getLeadingComments` involves edits to `leadingComments` and `leadingPtr`. The two examples are provided as input to LASE to generate a partially abstract, context-aware edit script.

`if` statement checking whether `trailingComments` is set to `null` and the assignment to `range`. When viewing the LASE's automatically provided edit suggestions, Bob notices that the suggested change involves inserting new variables. LASE cannot infer the names of the new variables because there are no matching variable names in the target context. Bob thus chooses the names of those variables by replacing `$v_1_`, `$v_2_` and `$v_3_` to concrete names. Choosing variables and any other changes Bob wishes to make could be easily done by making direct modifications on the edit suggestion in this comparison view. He applies the modified edits and repeats the process with the other methods.

III. Approach and Evaluation

This section summarizes LASE's three phase approach. We analyze edit operations and context using an Abstract Syntax Tree (AST) representation. Phase I takes as input multiple changed methods. LASE compares the old and new version of every input method. LASE identifies the longest common edit operation subsequence among the examples. When common edit operations in the examples use distinct names (variable, type, and method), LASE replaces the concrete identifier names with abstract names. Otherwise, it uses the original concrete identifiers. It finds the largest common context relevant to the edit operations using code clone detection, maximum common embedded subtree extraction, and dependence analysis. It then abstracts identifiers in the common context. Phase II uses the edit script's context to search for methods that match the context. Phase III customizes the edit to each new location and applies it. For each found method, LASE concretizes identifiers and code positions in the script to the target method, produces customized edits, and suggests a modified version.

To measure LASEs precision, recall, and accuracy, a test suite of supplementary bug fixes [2], [6] was used. Supplementary bug fixes are fixes that span multiple commits, where initial commits tend to be incomplete or incorrect, and thus developers apply supplementary changes to resolve the bug. If a bug is fixed more than once and there are clones of at least two lines in bug patches checked in at different times, they are manually examined for systematic changes. Using this method, 2 systematic edits in Eclipse JDT and 22 systematic edits in Eclipse SWT are found.

Meng et al. then 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. If LASE however produces the same results as developers do in later patches, it indicates that LASE can help programmers detect edit locations earlier, reduce errors of omissions, and make systematic edits. LASE locates edit positions with respect to the oracle data set with 95% precision, 88% recall, and performs edits with 91% accuracy. More details on LASE evaluation is described elsewhere [3].

IV. Summary

Our prior work, SYDIT [7], [8], produces code transformation from a single example only. It does not search for edit locations and requires developers to supply target edit locations. LASE learns non-trivial data and control context-aware edits from *multiple* edit examples and automatically searches for edit locations, and applies customized edits to the locations. LASE Eclipse plug-in allows users to select input

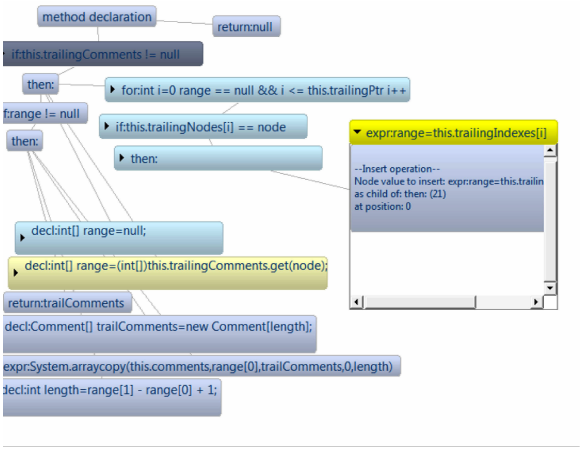

Fig. 4. LASE visualizes edit operations and corresponding context with respect to AST. Edited nodes are marked in light blue for insertions, dark blue for updates, yellow for deletes, and green for moves. Blue nodes are context nodes.

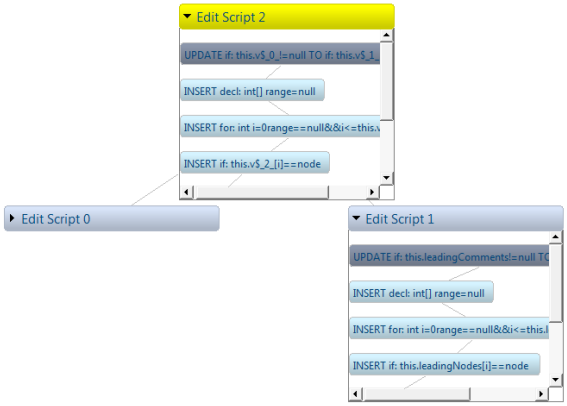

Fig. 5. LASE learns an edit from two or more examples. Each node in the edit script hierarchy corresponds to an edit script from a different subset of the input examples.

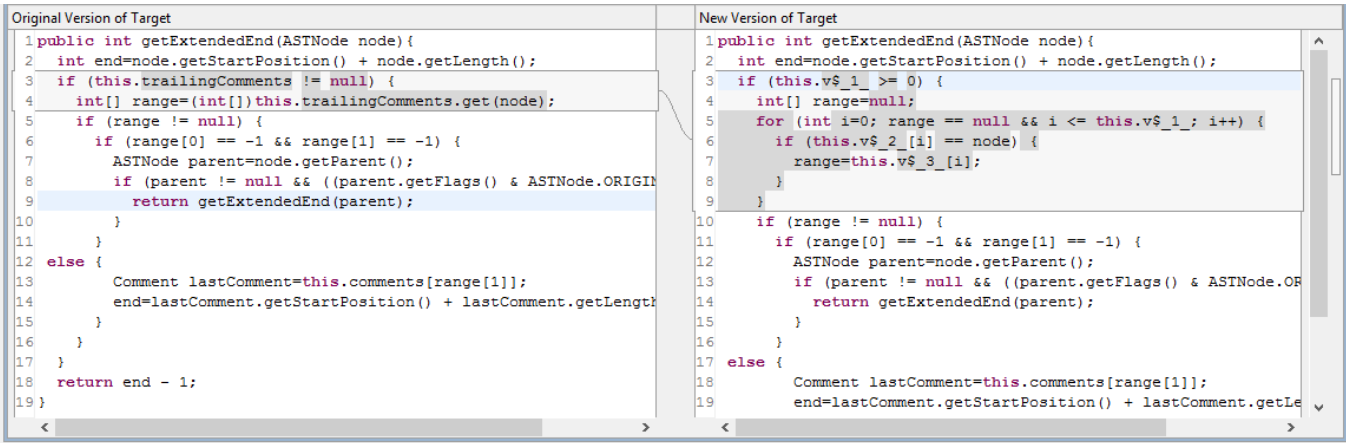

Fig. 6. A user can review and correct edit suggestions generated by LASE before approving the tool-suggested edit.

examples, to browse edit scripts inferred from the examples, and to inspect and correct tool-suggested edits.

ACKNOWLEDGMENT

We thank Kathryn McKinley for her valuable contribution in designing and evaluating the key algorithms in LASE and for discussions that helped us refine our idea. This work was supported by National Science Foundation under grants CCF-1149391, CCF-1117902, CCF-1043810, SHF-0910818, and CCF-0811524 and Microsoft SEIF award.

REFERENCES

[1] 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.

[2] 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, p. 11 pages.

[3] N. Meng, M. Kim, and K. McKinley, "Lase: Locating and applying systematic edits," in *ICSE' 13: Proceedings of the 35th ACM/IEEE International Conference on Software Engineering, Research Track (To appear)*, 2012, p. 10 pages.

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

[5] 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.

[6] 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.

[7] 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.

[8] N. Meng, M. Kim, and K. McKinley, "Sydit: Creating and applying a program transformation from an example," in *ESEC/FSE'11*, 2011, pp. 440–443.