Sydit: Creating and Applying a Program Transformation from an Example

Na Meng[*] Miryung Kim[*] Kathryn S. McKinley[*†]
[*] The University of Texas at Austin
[*] Austin, TX
[†]Microsoft Research
[†] Seattle, WA
mengna09@cs.utexas.edu, miryung@ece.utexas.edu, mckinley@cs.utexas.edu

ABSTRACT

Bug fixes and feature additions to large code bases often require *systematic edits*—similar, but not identical, coordinated changes to multiple places. This process is tedious and error-prone. Our prior work introduces a systematic editing approach that creates generalized edit scripts from exemplar edits and applies them to user-selected targets. This paper describes how the SYDIT plug-in integrates our technology into the Eclipse integrated development environment. A programmer provides an example edit to SYDIT that consists of an old and new version of a changed method. Based on this one example, SYDIT generates a *context-aware, abstract edit script*. To make transformations applicable to similar but not identical methods, SYDIT encodes control, data, and containment dependences and abstracts position, type, method, and variable names. Then the programmer selects target methods and SYDIT customizes the edit script to each target and displays the results for the programmer to review and approve. SYDIT thus automates much of the systematic editing process. To fully automate systematic editing, future tool enhancements should include automated selection of targets and testing of SYDIT generated edits.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*

General Terms

Design, Experimentation, and Measurement

Keywords

Software evolution, program transformation and differencing

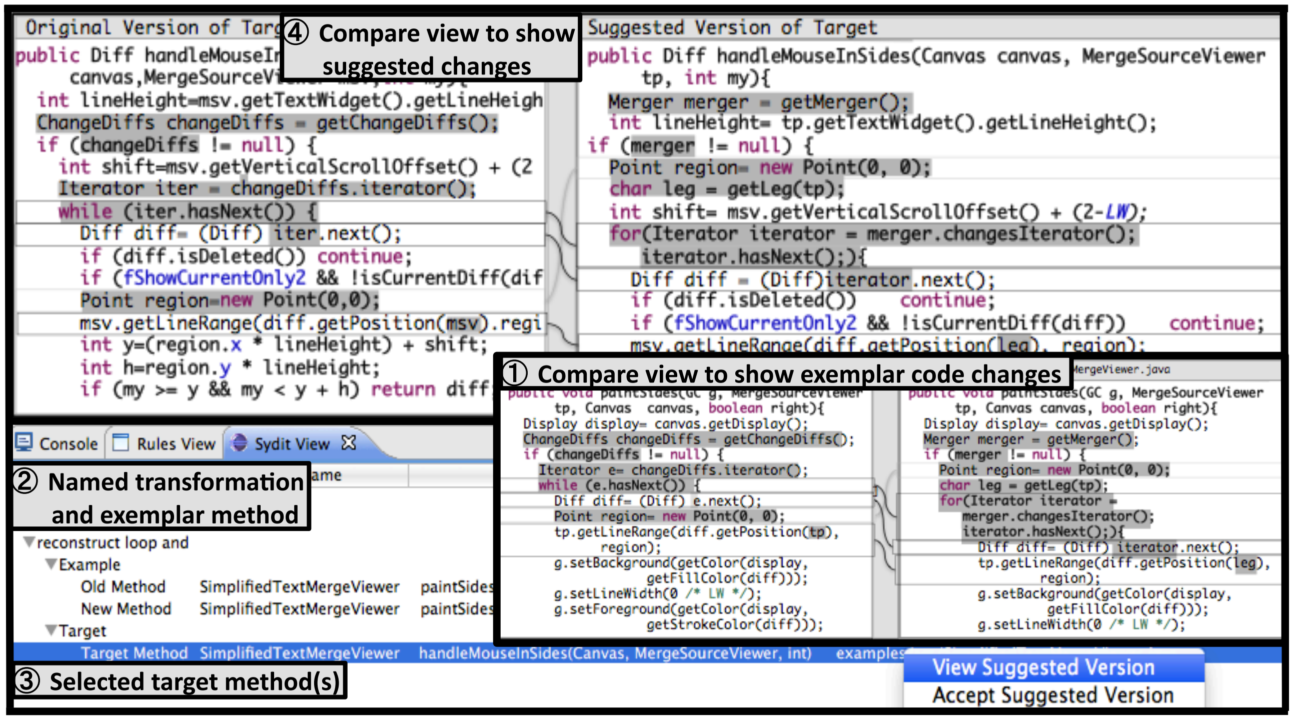
1. INTRODUCTION

A software life cycle begins with design, prototyping, and writing code. After the initial burst of development, programmers continue to fix bugs, refactor code, and add new features. Recent work observes that many changes during a maintenance phase are *systematic*—programmers add, delete, and modify code in numerous classes in similar, but not identical ways [9]. Manually making these edits is tedious and error prone.

We recently introduced a flexible systematic editing approach that consists of two algorithms: (1) *transformation creation*, which derives a generalized *context-aware, abstract* edit script from an exemplar edit, and (2) *transformation application*, which applies and customizes the derived transformation to target methods specified by the user [7]. Our prior work describes and evaluates this approach in detail. Our empirical evaluation used version histories from five Java open source projects and showed that (1) SYDIT produces syntactically valid transformations for 82% of target methods, (2) it perfectly mimics developer edits on 70% of the targets, and (3) syntactic program differencing judges the human generated version and the SYDIT generated version as 96% similar. We argue that working from examples is intuitive and this level of accuracy is sufficient to make it appealing for programmers as well.

This paper describes an Eclipse IDE plug-in implementation of the SYDIT technology in an interactive programming tool. To use SYDIT, users provide an *exemplar* edit. SYDIT then generates a program transformation that is more general than the concrete edit, which makes the edit applicable to similar but not identical target methods. Users next specify one or more target methods to apply the transformation and SYDIT produces a new version of each target method. Users examine each new version, edit them further, and/or accept the new versions as correct. To ease this process, SYDIT presents a *diff-style* comparison of each original target and its suggested versions. Users may examine, name, and store transformations for later use. Our initial experiences suggest that interactive refinement of the derived transformations and automated selection of targets are important features for future work.

Compared to prior work for helping programmers systematically fix and enhance software, SYDIT is more effective, convenient, and automated [7]. For instance, existing refactoring engines in Integrated Development Environments (IDEs), such as Eclipse, automate repetitive edits, but are confined to pre-defined semantic-preserving transformations. Source transformation languages and tools such as iXJ [1] and TXL [2] force programmers to prescribe systematic edits in a formal syntax in advance. Search-and-replace in text editors is the most popular approach, but it typically only supports simple text replacements and does not han-


④ Compare view to show suggested changes
① Compare view to show exemplar code changes
② Named transformation and exemplar method
③ Selected target method(s)
Console
Rules View
Sydit View
View Suggested Version
Accept Suggested Version

```
Original Version of Targ
public Diff handleMouseIn
    canvas,MergeSourceVi
  int lineHeight=msv.getTextWidget().getLineHeigh
  ChangeDiffs changeDiffs = getChangeDiffs();
  if (changeDiffs != null) {
    int shift=msv.getVerticalScrollOffset() + (2
    Iterator iter = changeDiffs.iterator();
    while (iter.hasNext()) {
      Diff diff= (Diff) iter.next();
      if (diff.isDeleted()) continue;
      if (fShowCurrentOnly2 && !isCurrentDiff(dif
      Point region=new Point(0,0);
      msv.getLineRange(diff.getPosition(msv).regi
      int y=(region.x * lineHeight) + shift;
      int h=region.y * lineHeight;
      if (my >= y && my < y + h) return diff
```

```
Suggested Version of Target
public Diff handleMouseInSides(Canvas canvas, MergeSourceViewer
      tp, int my){
  Merger merger = getMerger();
  int lineHeight= tp.getTextWidget().getLineHeight();
if (merger != null) {
  Point region= new Point(0, 0);
  char leg = getLeg(tp);
  int shift= msv.getVerticalScrollOffset() + (2-LW);
  for(Iterator iterator = merger.changesIterator();
      iterator.hasNext();){
    Diff diff = (Diff)iterator.next();
    if (diff.isDeleted())    continue;
    if (fShowCurrentOnly2 && !isCurrentDiff(diff))    continue;
    msv.getLineRange(diff.getPosition(leg), region);
```

```
      tp, Canvas  canvas, boolean right){
  Display display= canvas.getDisplay();
  ChangeDiffs changeDiffs = getChangeDiffs();
  if (changeDiffs != null) {
    Iterator e= changeDiffs.iterator();
    while (e.hasNext()) {
      Diff diff= (Diff) e.next();
      Point region= new Point(0, 0);
      tp.getLineRange(diff.getPosition(tp),
          region);
      g.setBackground(getColor(display,
                getFillColor(diff)));
      g.setLineWidth(0 /* LW */);
      g.setForeground(getColor(display,
                getStrokeColor(diff)));
```

```
MergeViewer.java
public void paintSides(GC g, MergeSourceViewer
      tp, Canvas canvas, boolean right){
  Display display= canvas.getDisplay();
  Merger merger = getMerger();
  if (merger != null) {
    Point region= new Point(0, 0);
    char leg = getLeg(tp);
    for(Iterator iterator =
        merger.changesIterator();
        iterator.hasNext();){
      Diff diff= (Diff) iterator.next();
      tp.getLineRange(diff.getPosition(leg),
          region);
      g.setBackground(getColor(display,
                getFillColor(diff)));
      g.setLineWidth(0 /* LW */);
```

```
ame
▼reconstruct loop and
  ▼Example
      Old Method     SimplifiedTextMergeViewer    paintSides
      New Method     SimplifiedTextMergeViewer    paintSides
  ▼Target
      Target Method  SimplifiedTextMergeViewer    handleMouseInSides(Canvas, MergeSourceViewer, int)    examples
```

Figure 1: A screen snapshot of Sydit Eclipse plug-in and its features

dle non-contiguous edits nor edits that require customization for different contexts. Clone detection and approaches do suggest edit locations, but leave the task of editing the code manually to programmers [9, 6]. With *simultaneous editing*, programmers edit pre-specified clones in parallel, naïvely propagating exactly the same edit to all clones without regard to context, which leads to errors [11, 8, 3]. In the work closest to Sydit, Lanza et al. monitor the evolution of a code fragment, deriving a sequence of change operations to represent this transformation [10]. Developers then specify any generalization of position and names by hand. In contrast, Sydit automatically derives a general edit script from the original and final version of an exemplar changed method, which makes it convenient for programmers to use, and our prior evaluation shows Sydit is effective.

Whereas our prior publication focused on the algorithms and evaluation, this tool paper makes new contributions by presenting Sydit's instantiation as an Eclipse plug-in, workflow architecture, user interfaces, and implementation details. The next section contains some background [7]. Section 2 illustrates the Sydit Eclipse plug-in features and user interface with an example scenario. Section 3 describes the implementation.

Background

This section summarizes Sydit's two key algorithms: (1) generating program transformations from an example and (2) applying the resulting context-aware abstract transformations to new code.

Sydit transformation generation algorithm takes an *exemplar edit* as input, which consists of the original and modified versions of a changed method. It first performs program differencing on the original and modified versions to generate a sequence of Abstract Syntax Tree (AST) node *additions, deletions, updates,* and *moves.* For each edit in the sequence, Sydit computes control, data, and containment dependences and then combines them to describe the edit's *context*, i.e., other program statements on which the edits depend, and statements that depend on the edits. Sydit then abstracts edit positions and the names of variables, methods, and types. This abstraction process generalizes edit positions and names. The result of this algorithm is an abstract context-aware edit script.

Sydit's transformation application algorithm takes as input this edit script and a target method. Sydit tries to establish a map of the edit context to the target method for exact and approximate matches of statements, dependences, positions, and names. If Sydit establishes a map, it next customizes the abstract edit script by replacing the matched abstract identifiers with the corresponding names used by the concrete AST nodes in the target method. Similarly, it calculates concrete edit positions based on the target. Sydit then generates a new version of the target by applying the customized concrete edit script to the target method.

2. SYDIT FEATURES

Consider a simplified example from the Eclipse `compare` project. Suppose Alice wants to replace the use of class `ChangeDiffs` with `Merger` and refactor iteration over `diff` objects in two methods: `paintSides` and `handleMouseInSides`. Part ① on the right-hand side of Figure 1 shows the exemplar edit, and ④ on the left-hand side shows Sydit's suggestion modification on a different target. Though both methods use the same API for `ChangeDiffs` and iterate `diff` objects, they use different variables names (`Iterator e` versus `Iterator iter`), and contain different code. For example, `paintSides` lays out the `diff` information while `handleMouseInSides` captures mouse activities over the `diff` layout. The required edits for these changes are not exactly the same with respect to edit content and position, but they are very similar.

In this example, both edits involve complex coordinated changes. To replace the use of `ChangeDiffs` with `Merger`, Alice needs to remove the declaration of `ChangeDiffs` and insert one for `Merger`. She also needs to replace all uses of `changeDiffs` with `merger` and its method invocations. To refactor the `diff` object iterator, Alice needs to replace the `while` loop with a `for` loop, move the declaration of variable `region` from inside to outside of the loop, and adjust the API usage. We use this relatively complicated example to show the power of Sydit. Since each method needs multiple non-contiguous edits and the edit content is different for each method, `search-and-replace` is not very helpful to Alice.

Alice reviews and tests her changes to `paintSides` as shown in ① in Figure 1. She next provides this example edit to Sydit. Using syntactic program differencing, Sydit computes a set of edits that transform the old version to the new version. In this case, Sydit computes: delete the `changeDiffs` declaration, insert the `merger` declaration, update the `if` condition `changeDiffs != null` with `merger != null`, move the `region` declaration out of the `while` loop, insert the `leg` declaration on the true branch of the `if` statement, delete the `e` declaration, update the `while` loop with a `for` loop, update the `diff` declaration, and update the argument of the `tp.getLineRange()` method invocation.

Sydit next generalizes these edits, abstracting position and names. The result is an abstract, context-aware edit script. Alice names this script, *reconstruct loop and change API*, and stores the **Old Method** and **New Method** of `paintSides` as an example edit (see Figure 1 ②).

Alice then selects one or more target methods to which she wants to apply the derived script. Alice selects `handleMouseInSides` (see Figure 1 ③). Sydit suggests similar but not identical edits, e.g., delete `iter`'s declaration instead of `e`'s, and update the argument of method invocation `msv.getLineRange()` instead of `tp.getLineRange()`.

Alice inspects the suggested version of `handleMouseInSides` using Eclipse's `compare` view (see Figure 1 ④). After Alice examines the suggested edits, she approves Sydit's suggestion by choosing **Accept Suggested Version**.

Although the target method `handleMouseInSides` and the source method `paintSides` have different structures and use different identifier names, Sydit correctly identifies a reusable, abstract transformation and applies it. Sydit ignores concrete names in the edit content and computes edit positions relative to the edit context. These abstraction features make the edit application problem harder, but are key to replicating similar but not identical program transformations.

3. SYSTEM ARCHITECTURE

This section describes the system architecture and implementation details of Sydit. Sydit currently targets Java programs and consists of five components: (1) program differencing, (2) edit context extraction through dependence analysis, (3) identifier and edit position abstraction, (4) context matching, and (5) program transformation through AST rewrites. As shown in Figure 2, the first three components correspond to program transformation creation while the last two correspond to program transformation application.

Program Differencing. Sydit compares the syntax trees

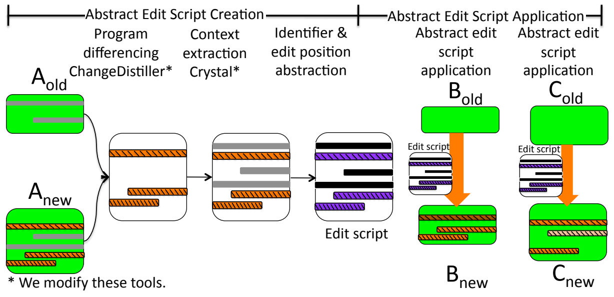

Abstract Edit Script Creation
Abstract Edit Script Application
Program differencing ChangeDistiller*
Context extraction Crystal*
Identifier & edit position abstraction
Abstract edit script application
Abstract edit script application
A_old
A_new
B_old
C_old
Edit script
Edit script
Edit script
B_new
C_new
* We modify these tools.

Figure 2: Sydit User and Tool Workflow

of two versions of the same method to derive a sequence of AST node edits. It performs this task with a modified version of ChangeDistiller [4]. For example, Figure 3 shows the derived edits for `paintSides` in Figure 1. We modify ChangeDistiller's algorithms in two ways. First, instead of allowing any inner nodes to match if their children nodes match, we also require the inner nodes should at least be similar control-flow constructs, such as `while` and `for` loops. Second, we match leaf nodes and inner nodes to handle cases such as matching a `catch clause` with an empty body (a leaf node) to a `catch clause` with a non-empty body (an inner node).

Edit Context Extraction. Sydit extracts a change context from both the old and new versions using control, data, and containment dependence analysis. By *context*, we mean other program statements on which the edits depend or those that depend on the edits. The context nodes serve as anchors to position edits correctly in a new target location. We implement the context analysis in the Crystal static analysis framework [5]. Sydit takes an input a parameter k, which indicates the dependence chain length, i.e., the *hop* distance from edits to context nodes. When k is set to 1, Sydit selects only unchanged nodes on which the edits are directly dependent upon or statements that direct depend on the edit. In Figure 3, the gray boxes represent the found contextual nodes when $k = 1$. Setting k to 2 selects unchanged directly dependent nodes and unchanged nodes that transitively depend on these nodes, i.e., dependences with a 2 hop distance.

Edit Position and Identifier Abstraction. Sydit then replaces concrete identifiers of variables, methods, and types with corresponding symbolic names, `v$n, m$n`, and `T$n` in the edits and extracted context, creating conversion mappings between them. It then recalculates the position of each edit with respect to the extracted context.

Context Matching. Given a target method, Sydit uses context matching to establish a mapping between the AST nodes in the edit script and the target method. The mapping problem is similar to a labeled graph isomorphism problem and we invented a specialized tree matching algorithm customized to the needs of this problem. The intuition behind our algorithm is that it first finds candidate leaf matches and then uses them to match inner nodes. We find as many candidate matches as possible between leaf nodes in the abstract context and the target tree. Starting from these candidate leaf matches, we determine a best match in the target context for each path. This algorithm is described in detail

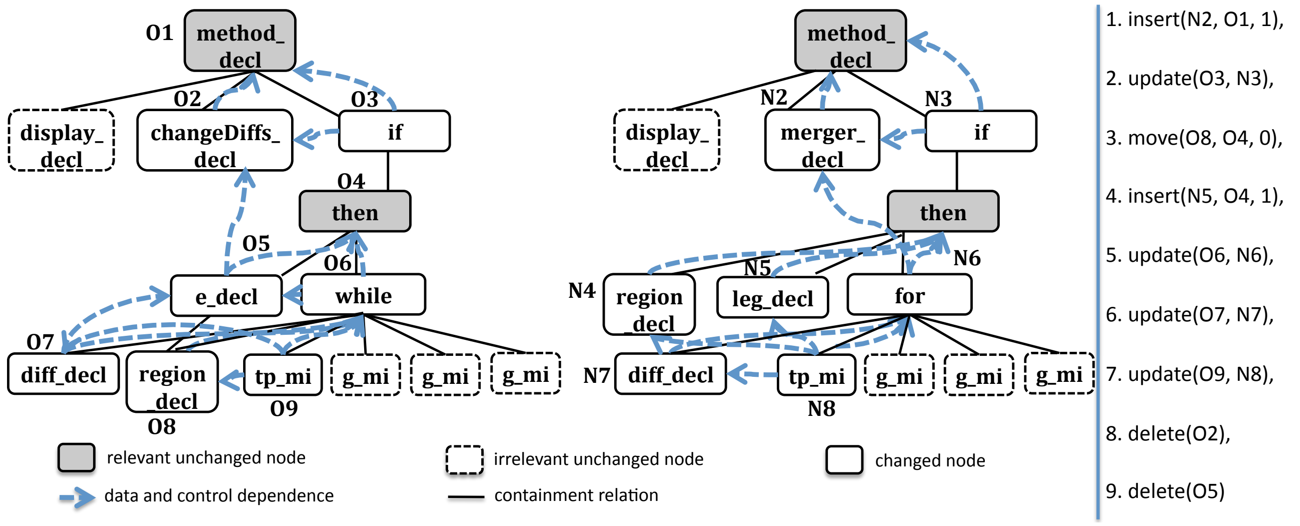


Figure 3: Concrete AST edits of `paintSides`

in reference [7]. If every node in the abstract script finds a unique correspondence in the target tree, SYDIT collects the identifier mappings based on the node matches and proceeds to the next step.

Program Transformation. To generate concrete edits for a target method, SYDIT replaces symbolic names used in the abstract edit script with corresponding concrete identifiers found in the target context. It also recalculates each edit position with respect to the concrete target method. SYDIT translates the resulting edits to a sequence of AST rewrite operations step by step using the Eclipse ASTRewrite API, and provides suggested edits to a user.

4. SUMMARY

SYDIT helps developers perform systematic editing tasks that involve similar but not necessarily identical changes to multiple places. The SYDIT plug-in makes it easier for developers to specify an exemplar changed method and multiple target methods to change similarly. It assists programmers by showing them the suggested changes for each target method and only using them if the programmer approves. In future, we plan to support developers to edit the generated abstract transformations for better flexibility. We also plan to simplify target method selection by searching for candidates which are similar to the exemplar changed method and providing potential candidates to programmers.

5. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CCF-1043810, SHF-0910818, and CCF-0811524. We thank anonymous reviewers for their thorough comments on our earlier version of the paper.

6. REFERENCES

[1] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07*, pages 567–576, New York, NY, USA, 2007. ACM.

[2] J. R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[3] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[4] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE TSE*, 33(11):18, November 2007.

[5] C. Jaspan, K. Bierhoff, and J. Aldrich. Crystal tutorial notes. 2009.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 28(7):654–670, 2002.

[7] N. Meng, M. Kim, and K. S. McKinley. Systematic Editing: Generating Program Transformations from an Example. In *PLDI '11*, pages 329–342, San Jose, CA, 2011. ACM.

[8] 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*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[9] 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*, pages 315–324, New York, NY, USA, 2010. ACM.

[10] R. Robbes and M. Lanza. Example-based program transformation. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 174–188, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.