Systematic Editing: Generating Program Transformations from an Example

Na Meng   Miryung Kim   Kathryn S. McKinley
The University of Texas at Austin
mengna152173@gmail.com, miryung@ece.utexas.edu, mckinley@cs.utexas.edu

Abstract

Software modifications are often *systematic*—they consist of similar, but not identical, program changes to multiple contexts. Existing tools for systematic program transformation are limited because they require programmers to manually prescribe edits or only suggest a location to edit with a related example. This paper presents the design and implementation of a program transformation tool called SYDIT. Given an example edit, SYDIT generates a *context-aware, abstract edit script*, and then applies the edit script to new program locations. To correctly encode a relative position of the edits in a new location, the derived edit script includes unchanged statements on which the edits are control and data dependent. Furthermore, to make the edit script applicable to a new context using different identifier names, the derived edit script abstracts variable, method, and type names. The evaluation uses 56 systematic edit pairs from five large software projects as an oracle. SYDIT has high coverage and accuracy. For 82% of the edits (46/56), SYDIT matches the context and applies an edit, producing code that is 96% similar to the oracle. Overall, SYDIT mimics human programmers correctly on 70% (39/56) of the edits. Generation of edit scripts seeks to improve programmer productivity by relieving developers from tedious, error-prone, manual code updates. It also has the potential to guide automated program repair by creating program transformations applicable to similar contexts.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—restructuring

General Terms Algorithm, Measurement, Experimentation

Keywords Software evolution, program transformation, program differencing, empirical study

1. Introduction

A typical software life-cycle begins with design, prototyping, and new code development. After deployment, software enters a phase where developers spend time fixing bugs, refactoring, and adding functionality to existing code. Recent work observes that many changes are *systematic*—programmers add, delete, and modify code in numerous classes in similar, but not identical ways [15, 16, 25]. For example, Kim et al. find that on average, 75% of structural changes to mature software are systematic. They find that these changes are not identical, but that their contexts have similar characteristics, such as calling the same method or accessing the same field. Nguyen et al. find that 17% to 45% of bug fixes are recurring fixes that involve similar edits to numerous methods [25]. Another class of systematic changes occur when API changes require all the API clients to update their code [12]. Performing systematic edits is currently tedious and error prone.

Existing tools offer limited support for systematic edits. The search and replace feature in a text editor is the most popular approach, but it supports only simple text replacements and cannot handle non-contiguous edits, nor edits that require customization for different contexts. Integrated Development Environments (IDEs), such as Eclipse, help with refactorings, but are confined to a predefined set of semantics-preserving transformations. Recent work proposes approaches for systematic editing, but none derive and apply context-aware edit scripts that use different variable, method, and type names in a new context. Nguyen et al. suggest new locations to edit based on changes to similar code fragments with an example, but require programmers to edit the code manually [23, 25]. 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 [7, 22, 29]. Programmers can also encode systematic edits in a formal syntax using a source transformation language, but this approach forces programmers to plan edit operations in advance [4, 5]. Andersen and Lawall's patch inference derives a more general edit from *diff* output, but its expressiveness is confined to term-replacements [1, 2]. Furthermore, it does not model the control and data dependence context of edits, and it does not encode the edit positions with respect to relevant context nodes.

This paper describes the design and implementation of an automated program transformation tool called SYDIT. SYDIT generates edit scripts from program differences and their context, and then applies scripts to similar code fragments. SYDIT characterizes edits as Abstract Syntax Tree (AST) node additions, deletions, updates, and moves. It uses control and data dependence analysis to capture the AST *change context*, i.e., relevant unchanged program fragments that depend on the edits or on which edits depend. It abstracts edit positions and the names of variables, methods, and types to create a generalized program transformation that does not depend on exact locations nor concrete identifiers. We call these transformations, *abstract, context-aware edit scripts*. Given a new target location, SYDIT generates concrete AST transformations customized to the new context. SYDIT then transforms the code accordingly.

To evaluate SYDIT, we create an oracle test suite of 56 systematic edit pairs—where two method locations are at least 40% similar in terms of their syntactic contents, and experience at least one common edit between two program versions. We draw this test suite directly from systematic updates performed by pro-

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...$10.00

grammers in `jEdit`, Eclipse `JDT core`, Eclipse `debug`, Eclipse `core.runtime` and Eclipse `compare` plug-ins. SYDIT takes as input a source exemplar edit, which consists of an old and a new program fragment, and generates an edit script. In this study, the programmer selects the target, and SYDIT generates concrete edits and applies them to the target. SYDIT produces syntactically valid transformations for 82% of the target methods (46/56). It perfectly mimics developer edits on 70% (39/56) of the targets. Syntactic program differencing considers the human generated version and the SYDIT generated version 96% similar. Therefore, it would likely require only modest manual effort to correct SYDIT's version. SYDIT achieves similar results on a suite of six systematic edits applied to five or more method locations from open-source projects. The key contributions of this paper are: (1) how to generalize a program transformation from an example to make it applicable to similar but not identical contexts, and (2) a rigorous empirical validation of SYDIT.

Our systematic editing approach seeks to improve programmer productivity when developers fix similar bugs, refactor multiple methods similarly, migrate code when APIs change [12, 23, 27], and add similar features to multiple related code locations. This approach is very flexible since developers can first develop and test a modification in a single context and then apply it to multiple contexts. Although in this paper, the programmer is required to select the target edit location and then examine the results, we envision more automated use cases as well. Given an example transformation, additional analysis could automate finding potential edit locations based on code similarity. By integrating SYDIT with automated compilation and testing, developers can have more confidence about the correctness of generated edits before reviewing them. Furthermore, this functionality could help guide automatic program repair by creating transformations applicable to similar contexts, applying them, and running regression tests on the SYDIT generated program version.

The rest of the paper is organized as follows. Section 2 illustrates SYDIT's edit script generation and application on a motivating example from the Eclipse `debug` plug-in. Section 3 presents our algorithms for edit script generalization and application. Section 4 shows the effectiveness of SYDIT using a test suite of independently developed systematic changes gathered from open-source projects. Section 5 compares our approach to related work and Section 6 discusses the limitations of our approach and ways to improve SYDIT's precision.

2. Motivating Example

This section overviews our approach with a running example drawn from revisions to org.eclipse.debug.core on 2006-10-05 and 2006-11-06. Figure 1 shows the original code in black, additions in **bold blue** with a **'+'**, and deletions in red with a '-'. Consider methods `mA` and `mB`: `getLaunchConfigurations(ILaunchConfigurationType type)` and `getLaunchConfigurations(IProject project)`. These methods iterate over elements received by calling `getAllLaunchConexfigurations()`, process the elements one by one, and when an element meets a certain condition, add it to a predefined list.

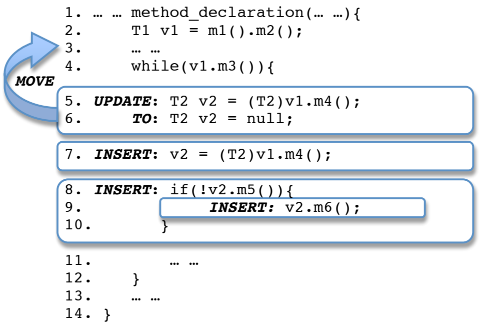
Suppose that Pat intends to apply similar changes to `mA` and `mB`. In `mA`, Pat wants to move the declaration of variable `config` out of the `while` loop and add code to process `config` as shown in lines 4, and 6-10 in `mA`. Pat wants to perform a similar edit to `mB`, but on the `cfg` variable instead of `config`. This example typifies *systematic edits*. Such similar yet not identical edits to multiple methods cannot be applied using the search and replace feature or existing refactoring engines in IDE, because they change the semantics of a program. Even though these two program changes are similar, without assistance, Pat must manually edit both methods, which is tedious and error-prone.

A$_{old}$ to A$_{new}$

```
1. public ILaunchConfiguration[] getLaunchConfigurations
   (ILaunchConfigurationType type) throws CoreException {
2.    Iterator iter = getAllLaunchConfigurations().iterator();
3.    List configs = new ArrayList();
4.  + ILaunchConfiguration config = null;
5.    while (iter.hasNext()) {
6.      - ILaunchConfiguration config =
                 (ILaunchConfiguration)iter.next();
7.      + config = (ILaunchConfiguration)iter.next();
8.      + if (!config.inValid()) {
9.      +   config.reset();
10.     + }
11.      if (config.getType().equals(type)) {
12.         configs.add(config);
13.      }
14.   }
15.   return (ILaunchConfiguration[])configs.toArray
                 (new ILaunchConfiguration[configs.size()]);
17.}
```

SYDIT's replication of relevant edits on B$_{old}$, resulting in B$_{new}$

```
1. protected List getLaunchConfigurations(IProject project) {
2.    Iterator iter = getAllLaunchConfigurations().iterator();
3.  + ILaunchConfiguration cfg = null;
4.    List cfgs = new ArrayList();
5.     while (iter.hasNext()) {
6.       - ILaunchConfiguration cfg =
                 (ILaunchConfiguration)iter.next();
7.       + cfg = (ILaunchConfiguration)iter.next();
8.       + if (!cfg.inValid()) {
9.       +   cfg.reset();
10.      + }
11.      IFile file = cfg.getFile();
12.      if (file != null && file.getProject().equals(project)) {
13.         cfgs.add(cfg);
14.      }
15.   }
16.   return cfgs;
17.}
```



```
1.  … … method_declaration(… …){
2.      T1 v1 = m1().m2();
3.      … …
4.      while(v1.m3()){
5.  UPDATE: T2 v2 = (T2)v1.m4();
6.      TO: T2 v2 = null;
7.  INSERT: v2 = (T2)v1.m4();
8.  INSERT: if(!v2.m5()){
9.              INSERT: v2.m6();
10.          }
11.          … …
12.     }
13.     … …
14. }
```

Abstract edit script

Figure 1. Systematic edit from revisions of org.eclipse.debug.core

Using SYDIT, Pat applies the change only to `mA` and then SYDIT creates an edit script and applies it to `mB`. SYDIT applies a syntactic program differencing algorithm to `mA`$_{old}$ and `mA`$_{new}$ and then characterizes the exemplar edit (lines 4, and 6-10) in terms of a sequence of inserts, deletes, moves, and updates. For each edit, it performs data and control dependence analysis to determine the edit's corresponding context. It then abstracts variable, method, and type names and encodes edit positions relative to the extracted context. This identifier and edit position abstraction makes the edits applicable to similar yet not identical contexts.

Pat next identifies mB as a target for this edit script. Based on the extracted context and the abstract names in the edit script, SYDIT matches the extracted context against the mB target and identifies relevant code fragments that the edits are applicable to (i.e., lines 2, 5, and 6 in mB). It then creates *concrete* edits customized to mB, and applies them to mB.

The bottom of Figure 1 illustrates the content of the abstract edit script derived from the two versions of mA. Currently, SYDIT does not guarantee to generate an edit script containing the fewest edits.

Context extraction. SYDIT extracts the context of edits through data and control dependence analysis, which makes it possible to apply the edit script to code fragments that share similar data and control flows but are not identical to mA. For example, the location of the updated ILaunchConfiguration declaration is the child position 0 inside the while loop. Position 0 means the first AST child node of the while loop. The updated statement is control dependent on the while (line 5), and data dependent on the iterator declaration (line 2). We therefore include both lines 2 and 5 in the abstract context of this update.

Identifier abstraction. SYDIT abstracts all variable, method, and type names to make the edits applicable to contexts that use different identifier names. For instance, in the example, it abstracts the config variable in mA to v2.

Edit position abstraction. SYDIT encodes the syntactic position of each edit with respect to the extracted context nodes. For example, the source of the moved ILaunchConfiguration declaration is child position 0 of the while (i.e., its first AST child node), and the target is child position 1 of the method declaration node (i.e., its second AST child node).

Edit script application. To apply an abstract edit script to a target context, SYDIT matches the abstracted context against the target method and concretizes the abstract identifier names and edit positions with respect to the target method. For example, it concretizes v2 to cfg, and encodes the target move position as the child position 1 of mB's method declaration node.

This automatic program transformation approach offers a flexible mechanism for improving programmer productivity. Systematic editing simplifies the tedious task of applying similar changes to multiple locations. Context abstraction and concretization increases the chance to apply systematic edits correctly and consistently. In this paper, the programmer selects both the source and target method and audits the result. With integration with automated compilation and testing, examining the correctness of SYDIT generated edits could be further automated. In all cases, the programmer should audit SYDIT generated versions through testing, inspection, or both. The coverage and accuracy of the edit scripts depend on the choice of the source exemplar edit, how context and abstract names are represented, and the algorithm that matches the extracted edit context to a target.

3. Approach

This section describes the two phases of SYDIT. Phase I takes as input an old and new version of method mA as its exemplar edit, and creates an edit script from $\texttt{mA}_o$ and $\texttt{mA}_n$. Phase II applies the edit script to a new context, mB, producing a modified method $\texttt{mB}_s$. We first summarize the steps in each phase and then describe each step in detail.

Phase I: Creating Edit Scripts

- SYDIT compares an exemplar edit, $\texttt{mA}_o$ and $\texttt{mA}_n$, and describes the differences as a sequence of insertions, deletions, updates, and moves: $\Delta_A = \{e_o, e_1, \ldots, e_n\}$.
- SYDIT identifies the context of the edit Δ_A based on data, control, and containment dependences between each e_i and other statements in $\texttt{mA}_o$ and $\texttt{mA}_n$.
- SYDIT abstracts the edit, Δ, by encoding each e_i position with respect to its extracted context and by replacing all concrete variable, method, and type names with abstract identifier names.

Phase II: Applying Edit Scripts

- SYDIT matches the abstract context for Δ to mB's syntax tree.
- If they match, SYDIT generates a concrete edit Δ_B by translating abstract edit positions in Δ into concrete positions in mB and abstract identifiers in Δ into concrete identifiers in mB.
- SYDIT then applies Δ_B to mB, producing $\texttt{mB}_s$.

3.1 Phase I: Creating Abstract Edit Scripts

This section explains how SYDIT create an abstract edit script.

3.1.1 Syntactic Program Differencing

SYDIT compares the syntax trees of an exemplar edit, $\texttt{mA}_o$ and $\texttt{mA}_n$, using a modified version of ChangeDistiller [9]. ChangeDistiller generates deletes, inserts, moves, and updates. We chose ChangeDistiller in part because it produces concise AST edit operations by (1) representing related node insertions and deletions as moves and updates and (2) aggregating multiple fine-grained expression edits into a single statement edit.

ChangeDistiller computes one-to-one node mappings from the original and new AST trees for all updated, moved, and unchanged nodes. If a node is not in the mappings, ChangeDistiller generates deletes or inserts as appropriate. It creates the mappings bottom-up using: *bigram string similarity* for leaf nodes (e.g., statements and method invocations), and *subtree similarity* for inner nodes (e.g., while and if statements). It first converts each leaf node to a string and computes its bigram—the set of all adjacent character pairs. The bigram similarity of two strings is the size of their bigram set intersection divided by the average of their sizes. If the similarity is above an input threshold, σ, ChangeDistiller includes the two leafs in its pair-wise mappings. It then computes subtree similarity based on the number of leaf node matches in each subtree, and establishes inner node mappings bottom up.

We modify ChangeDistiller's matching algorithms in two ways. First, we require inner nodes to perform equivalent control-flow functions. For instance, the original algorithm sometimes mapped a while to an if node. We instead enforce a structural match, i.e., while nodes only map to while or for nodes. Second, we match leaf nodes to inner nodes using bigram string similarity. This change overcomes inconsistent treatment of blocks. For example, ChangeDistiller treats a catch clause with an empty body as a leaf node, but a catch clause with a non-empty body is an inner node.

Given a resulting set of AST node mappings, SYDIT describes edit operations with respect to the original method $\texttt{mA}_o$ as follows:

delete (Node u): delete node u from $\texttt{mA}_o$.

insert (Node u, Node v, int k): insert node u and position it as the $(k+1)^{th}$ child of node v.

move (Node u, Node v, int k): delete u from its current position in $\texttt{mA}_o$ and insert u as the $(k+1)^{th}$ child of v.

update (Node u, Node v): replace u in $\texttt{mA}_o$ with v. This step includes changing the AST type in the resulting tree to v's type and maintaining any of u's AST parent and children relationships in v.

The resulting sequence of syntactic edits is $\Delta_A = \{e_i | e_i \in \{$delete (u), insert (u,v,k), move (u,v,k), update (u,v)$\}\}$. We use a total order for e_i to ease relative positioning of edits.

Figure 2 presents the mapping for our example, where 'O' is a node in the old version `mA`$_o$ and 'N' is a node in the new version `mA`$_n$. Below we show the concrete edit script Δ_A that transforms `mA`$_o$ into `mA`$_n$ for our example:

1. update (O6, N4)
   O6 = ‘`ILaunchConfiguration config =`
   `(ILaunchConfiguration) iter.next();`’
   N4 = ‘`ILaunchConfiguration config = null;`’
2. move (O6, N1, 2)
3. insert (N7, N5, 0)
   N7 = ‘`config = (ILaunchConfiguration) iter.next();`’
4. insert (N8, N5, 1) N8 = ‘`if (!config.inValid())`’
5. insert (N9, N8, 0) N9 = ‘then’
6. insert (N10, N9, 0) N10 = ‘`config.reset();`’

3.1.2 Extracting Edit Contexts

Sydit extracts relevant context from both the old and new versions. For each edit $e_i \in \Delta_A$, Sydit analyzes `mA`$_o$ and `mA`$_n$ to find the unchanged nodes on which changed nodes in e_i depend. These dependences include containment dependences and the source and sink of control and data dependences. We call these nodes *context*.

Context information increases the chance of generating syntactically valid edits and also serves as anchors to position edits correctly in a new target location. First, in order to respect the syntax rules of the underlying programming language, we include AST nodes that the edits require. For example, insertion of a `return` statement must occur inside a method declaration subtree. This context increases the probability of producing a syntactically valid, compilable program. Second, we use control dependences to describe the position to apply an edit, such as inserting a statement at the first child position of `while` loop. While the edit may be valid outside the `while`, positioning the edit within the `while` increases the probability that the edit will be correctly replicated. Third, context helps preserve data dependences. For example, consider an edit that inserts statement `S2: foo++;` after `S1: int foo = bar;`. If we require `S1` to precede `S2` by including `S1` in the context of `S2`, the resulting edit will guarantee that `foo` is defined before it is incremented. However, including and enforcing more dependence requirements in the edit context may decrease the number of target methods that will match and thus may sacrifice coverage.

Formally, node y is *context dependent* on x if one of the following relationships holds:

- *Data dependence*: node x uses or defines a variable whose value is defined in node y. For example, N10 uses variable `config`, whose value is defined in N7. Therefore, N10 is data dependent on N7.
- *Control dependence*: node y is control dependent on x if y may or may not execute depending on a decision made by x. Formally, given a control-flow graph, node y is control dependent on x, if: (1) y post-dominates every vertex p in $x \rightsquigarrow y$, $p \neq x$, and (2) y does not strictly post-dominate x [6].
- *Containment dependence*: node y is containment dependent on x if y is a child of x in the AST. For instance, N4 is containment dependent on N1.

To extract the context for an edit script, we compute control, data, and containment dependences on the old and new versions. The context of an edit script Δ_A is the union of these dependences. The containment dependence is usually redundant with immediate control dependence of x, except when loops contain early returns. To combine dependences, Sydit *projects* nodes found in the new version `mA`$_n$ onto corresponding nodes in the old version `mA`$_o$ based on the mappings generated by the modified version of ChangeDistiller. For each $e_i \in \Delta_A$, we determine relevant context nodes as follows.

delete (u)**:** The algorithm computes nodes in `mA`$_o$ that depend on the deleted node, u.

insert (u, p, k)**:** Since u does not exist in `mA`$_o$, the algorithm first computes nodes in `mA`$_n$ on which u depends and then projects them into corresponding nodes in `mA`$_o$.

move (u, v, k)**:** The algorithm finds the dependent nodes in both `mA`$_o$ and `mA`$_n$ related to u. The nodes in the new version help guarantee dependence relationships after the update. It projects the nodes from `mA`$_n$ into corresponding nodes in `mA`$_o$ and then unions the two sets.

update (u, v)**:** The algorithm finds the dependent nodes in `mA`$_o$ related to the updated node, u. It also finds dependent nodes in `mA`$_n$ related to the node v. It projects the nodes from `mA`$_n$ into corresponding nodes in `mA`$_o$ and then unions the two sets.

Consider insert (N7, N5, 0) from Figure 2. The inserted node N7 is control dependent on N5, and data dependent on N2 and N4. Mapping these nodes to the old version yields the context node set $\{$O2, O4, O6$\}$. The move (O6, N1, 2) operation is more complicated because O6 depends on the node set C1 = $\{$O4, O2$\}$ in the old version, while N4, N1's child at position 2, depends on the node set C2 = $\{$N1$\}$ in the new version. After deriving the two sets, we project C2 onto nodes in `mA`$_o$, which yields C3 = $\{$O1$\}$. Finally, we union C1 and C3 to get the context node set $\{$O1, O2, O4$\}$ for the move operation. Figure 2 illustrates the result, marking irrelevant nodes with dotted lines and context nodes in gray.

Sydit allows the user to configure the amount of context. For example, the number of dependence hops, k, controls how many surrounding, unchanged nodes to include in the context. Setting $k = 1$ selects just the immediate control and data dependent nodes. Setting $k = \infty$ selects all control and data dependent nodes. We can restrict dependences to reaching definitions or include the nodes in a chain of definitions and uses depending on k. Sydit differentiates *upstream* and *downstream* dependences. Upstream dependences precede the edit in the text, whereas downstream dependences follow the edit. The default setting of Sydit is $k = 1$ with control, data, and containment upstream dependences, which was best in practice. Section 4 shows how varying context affects Sydit's coverage and accuracy.

3.1.3 Abstracting Identifiers and Edit Positions

At this point, the edit script and its context use concrete identifier names and edit positions from the exemplar edit. To make the edit script more applicable, we abstract identifier names and edit positions in the edit script.

To abstract identifiers, we replace all concrete variable, method, and type names with equivalent abstract representations: `T$x`, `m$x`, and `v$x` respectively. Each unique concrete identifier corresponds to a unique abstract one. For example, we convert the concrete expression `!config.inValid()` in Figure 2 to `!v2.m5()` in Figure 3.

We abstract the position of edits to make them applicable to code that differs structurally from the original source example. We encode an edit position as a relative position with respect to all the context nodes, instead of all nodes in the original syntax tree. For example, we convert the concrete edit move (O6, N1, 2) to an abstract edit move (AO4, AN1, 1). In this case, the abstract edit position is child position 1 of the `while` because the context of the edit includes the definition of `ILaunchConfiguration` at abstract child position 0 and no other dependences. This relative position ensures that `ILaunchConfiguration` is defined by some

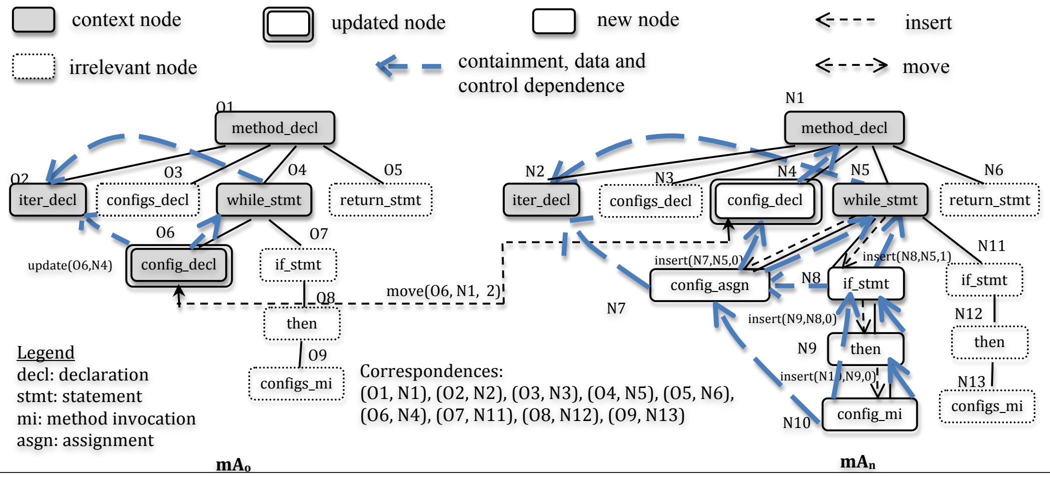


Figure 2. Syntactic edit extraction for `mA`

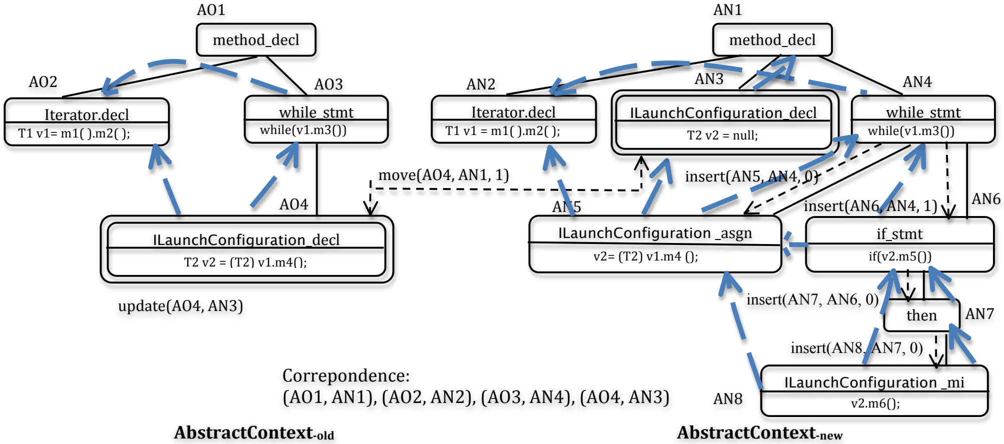


Figure 3. Abstract edit script

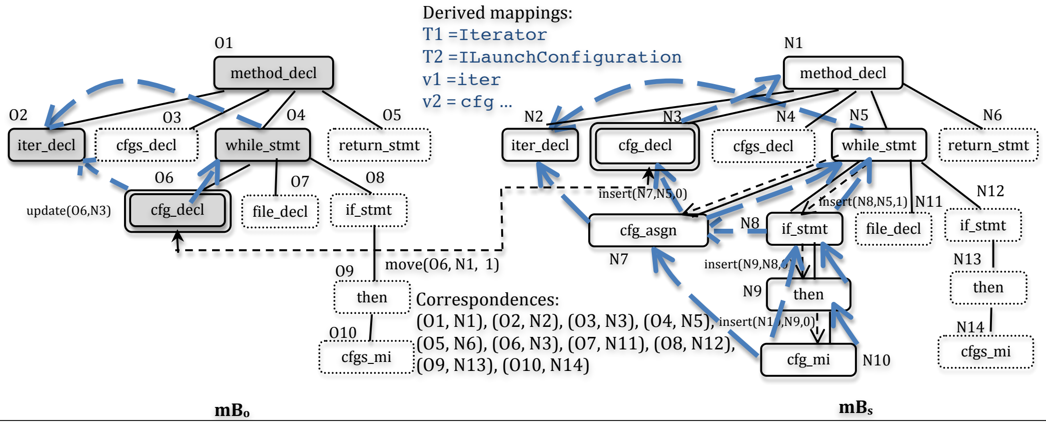


Figure 4. Syntactic edit suggestion for `mB`

statement before it is used, but requires no other statements in the `while`. When we apply the edit, we require the context to match and apply edits relative to the context position in the target, not the concrete positions in the original method. Abstracting edit positions is essential for applying an edit when the target method satisfies the context dependences, regardless of the exact positions of the statements in the code.

3.2 Phase II: Applying Abstract Edits

This section describes how SYDIT applies an edit script Δ to a method `mB`, producing a modified method `mB`$_s$.

3.2.1 Matching Abstract Contexts

The goal of our matching algorithm is to find nodes in the target method that match the context nodes in Δ and that induce one-to-one mappings between abstract and concrete identifier names. To simplify this process, we first abstract the identifiers in `mB`. We use the procedure as described in Section 3.1.3 to create mB$_{Abstract}$ from mB. For concision in this section, we simply use mB instead of mB$_{Abstract}$.

This problem can be posed as the labeled subgraph isomorphism problem. Although we experimented with an off-the-shelf implementation [20], adapting it to match nodes while simultaneously requiring one-to-one symbolic identifier mappings is difficult. Yet these two features are essential requirements for applying edits to new contexts. See Section 3.2.2 for more details. The algorithm we propose below tolerates inexact label matches for unchanged context nodes while enforcing one-to-one symbolic identifier mappings.

The intuition behind our algorithm is to first find candidate leaf matches and then use them to match inner nodes. We find as many candidate matches as possible between leaf nodes in the abstract context and leaf nodes in the target tree, $x \in$ AC, $y \in$ mB, where x and y form *an exact match*, i.e., the equivalent AST node types and node labels (see below). Based on these exact matches (x, y), we add node matches (u, v), where u and v are on paths from the respective root nodes, $u \in (root_{AC} \rightsquigarrow x)$ and $v \in (root_{mB} \rightsquigarrow y)$. We add (u, v) type matches bottom-up, requiring only their node types to be equivalent. Finally for each unmatched leaf in the abstract context, we find additional *type matches* based on the established set of matches. We repeat these steps until the set of candidate leaf matches, CL, does not increase any more. We define two types of node matches and one type of path matches:

Type match: Given two nodes u and v, (u,v) is a type match if their AST node types match. For example, both are ifs or one is a while and the other is a for. The conditions need not match.

Exact match: Given two nodes u and v, (u,v) is an exact match if it is a type match *and* their AST *labels* are equivalent. We define the label as the abstract strings in the statements and ignore numerics in abstract identifiers. For example, 'T1 v1 = null;' and 'T2 v2 = null;' are equivalent since we ignore the numeric and convert them both to 'T v = null;'.

Path match: Given two paths $p1$ and $p2$, $(p1, p2)$ is a path match if for every node u on $p1$, there exists node v on $p2$ where (u, v) is a type match. For example, given two leaf nodes x and y, the paths match if parent(x) and parent(y) type match, parent(parent(x)) and parent(parent(y)) type match, and so on.

We use these definitions to map the nodes in the abstract context AC in Δ to mB in the following four steps, which Algorithm 1 describes procedurally.

1. SYDIT finds all exact leaf matches between AC and mB and adds each (x, y) pair to a set of candidate leaf matches, CL.
2. Based on CL, SYDIT then tries to find the best path match for each leaf node x where $(x, y) \in CL$ and finds node matches based on the best path match. Let $p1 = root_{AC} \rightsquigarrow x$ and $p2 = root_{mB} \rightsquigarrow y$. This step is broken into three cases, for each node match (x, y) in CL,
   (a) If there exists one path match $(p1, p2)$ between AC and mB, we add all its constituent node matches (u, v) on these paths to M.
   (b) If there exists multiple path matches, e.g., ($p1$, $(root_{mB} \rightsquigarrow y_1)$) and ($p1$, $(root_{mB} \rightsquigarrow y_2)$), and one of these path matches contains more constituent nodes already in the set established matches M, we select the best of these path matches and add constituent (u, v) matches to M.
   (c) If there exists multiple path matches with the same number of constituent node matches in M, SYDIT leverages sibling ordering relationships among the leaf nodes to disambiguate the best path match. Given a leaf node $x \in$ AC, suppose that path $p1$ matches with multiple paths, e.g., ($p2 = root_{mB} \rightsquigarrow y_2$), ($p3 = root_{mB} \rightsquigarrow y_3$), with the same score and assume that y_2 precedes y_3 in sibling order.

   If a node match (u, v) exists in M in which u precedes x in terms of sibling order, and v is a sibling between y_2 and y_3, SYDIT prefers a path match ($(root_{AC} \rightsquigarrow x)$, $(root_{mB} \rightsquigarrow y_3)$), since this choice is consistent with an already estab-

Algorithm 1: Matching Abstract Context to Target Tree

```
Input: AC, mB /* abstract context and abstract target
   tree                                                  */
Output: M /* a set of node matches from AC to mB         */
/* 1. create candidate leaf exact matches                */
CL := ∅;
M := ∅;
foreach leaf node $x \in$ AC do
    foreach leaf node $y \in$ mB do
        if $exactMatch(x, y)$ then
            CL:= CL ∪ {$(x, y)$};
        end
    end
end
repeat
    /* 2(a). create matches based on path matches         */
    foreach $(x, y) \in$ CL such that $\nexists$ $(x, z) \in$ CL $\wedge$ $y \neq z$ do
        $p1 = (root_{AC} \rightsquigarrow x)$;
        $p2 = (root_{mB} \rightsquigarrow y)$;
        if $pathMatch(p1, p2)$ then
            M:= M ∪ {$(u, v) \mid u \in p1$ , where $(u,v)$ is a type
            match and $v \in p2$ and $u$ and $v$ appear in the same
            position on paths $p1$ and $p2$};
        end
    end
    /* 2(b). select the best path match and add new
       node matches it induces                            */
    foreach (leaf node $x \in AC$ such that $(x, y) \in CL \wedge (x,y) \notin$ M)
    do
        $p1 = (root_{AC} \rightsquigarrow x)$;
        $p2 = (root_{mB} \rightsquigarrow y)$;
        select $y$ with the maximum $pathMatchScore(p1, p2,$
        M);
        M:= M ∪{$(u, v) \mid u \in p1$ and $v \in p2$, where $(u,v)$ is a
        type match and $u$ and $v$ appear in the same position on
        paths $p1$ and $p2$};
    end
    /* 2(c). disambiguate path matches based on the
       sibling order of matched leaf nodes in M           */
    foreach (leaf node $x \in AC$ such that $(x, y) \in CL \wedge (x,y) \notin$ M)
    do
        select $y$ with the maximum $LCSMatchScore(x,y,$M);
        M:= M ∪{$(u, v) \mid u \in p1$ and $v \in p2$, where $(u,v)$ is a
        type match and $u$ and $v$ appear in the same position on
        paths $(root_{AC} \rightsquigarrow x)$ and $(root_{mB} \rightsquigarrow y$};
    end
    /* 3. establish symbolic identifier mappings          */
    S:= ∅;
    foreach $(u, v) \in$ M do
        S:= S ∪ {(T$n, T$m), (v$i, v$j), and/or (m$k, m$l)
           that are supported by $(u, v)$};
    end
    $removeConflicts$(S, M);
    /* 4. relax constraints to add leaf candidates
       */
    CL:= CL ∪ $relaxConstraints$(AC, M);
until CL reaches its fix point ;
```

lished match (u, v). Similarly, based on this path match, we add constituent node matches on the matched paths to `M`. While this approach is similar to how the longest common subsequence (LCS) algorithm align nodes [14], our approach matches leaf nodes based on established matches in `M`.

3. SYDIT establishes mappings between symbolic identifiers in `AC` and `mB` by enumerating all node matches in `M`. For example, if the label of matched nodes are '`T1 v1 = null;`' and '`T2 v2 = null;`', we add the symbolic identifier mappings `(T1, T2)` and `(v1, v2)` to `S`. While collecting identifier mappings, the algorithm may encounter inconsistencies, such as `(T1, T3)`, which violates an already established mapping from `T1` to `T2`. To remove the conflict between `(T1, T2)` and `(T1, T3)`, SYDIT counts the number of node matches that support each mapping. It keeps the mapping with the most support, and removes other mappings from `S` and all their supporting node matches from `M`.
4. SYDIT leverages the parent-child relationship of matched nodes in `M` to introduce type matches for unmatched leaf(s) in `AC`. For each unmatched leaf z in `AC`, SYDIT traverses bottom-up along its path to root in order to find the first ancestor u which has a match $(u, v) \in M$. Next, if it finds an unmatched node w in the subtree rooted at v and if (z, w) is a type match, SYDIT adds it into `CL`. We repeat steps 2 to 4 until step 4 does not add any to `CL`.

At any point in this process, if every node in the abstract context `AC` has a match in `M`, then we proceed to derive concrete edits customized to `mB`, described in Section 3.2.3. If we fail to find a match for each node, SYDIT reports to the user that the edit context does not match and it cannot replicate the edit on the target context.

3.2.2 Alternative matching algorithms

Standard labeled subgraph isomorphism is a promising alternative approach for matching abstract context in Δ to a new target method `mB` that we also explored. We formulated both the abstract content and target method as graphs in which nodes are labeled with their AST node types, and edges are labeled with constraint relationships between nodes, such as containment, data, and control dependences. To preserve a one-to-one mapping between abstract and concrete identifiers, we included additional labeled nodes to represent the sequence of symbols appearing in the statement. We included variable names, method names, type names, as well as constants like `null` and operators like `=` as node labels. Next, we connected all the identifiers with the same name with edges labeled "same name." We thus converted our problem to finding an isomorphic labeled subgraph in the target method's graph for the abstract context's graph.

Function `pathMatch(`*path* $p1$, *path* $p2$`)`

```
t1 := p1's bottom-up iterator;
t2 := p2's bottom-up iterator;
while $t1.hasPrev() \wedge t2.hasPrev()$ do
    $u := t1.prev()$;
    $v := t2.prev()$;
    if $!EquivalentNodeType(u, v)$ then
        return $false$;
    end
end
if $t1.hasPrev()$ then
    return $false$;
end
return $true$;
```

Function `pathMatchScore(`*path* $p1$, *path* $p2$, *matches* M`)`

```
$counter := 0$;
$t1 := p1$'s bottom-up iterator;
$t2 := p2$'s bottom-up iterator;
while $t1.hasPrev() \wedge t2.hasPrev()$ do
    $u := t1.prev()$;
    $v := t2.prev()$;
    if $(u, v) \in M$ then
        $counter++$;
    end
end
return $counter$;
```

Function `LCSMatchScore(`*node x, node y, matches* M`)`

```
$score := 0$;
/* identify left siblings of $x$ and $y$ */
$l1 := left\_children(parent(x), x)$;
$l2 := left\_children(parent(y), y)$;
/* identify right siblings of $x$ and $y$ */
$r1 := right\_children(parent(x), x)$;
$r2 := right\_children(parent(y), y)$;
/* compute the size of longest common sequences of
   $l1$ and $l2$ and $r1$ and $r2$ respectively with respect
   to $M$. */
$score := LCS(l1, l2, M) + LCS(r1, r2, M)$;
return $score$;
```

Function `removeConflicts(`*mappings S, matches M*`)`

```
foreach $(s1, s2) \in S$ do
    $T = \{t \mid (s1, t) \in S\}$;
    if $|T| > 1$ then
        select $t$ with the most supporting matches;
        $T = T$ - (s1, t);
        foreach $s2 \in T$ do
            $S := S - \{(s1, s2)\}$;
            $M := M - \{(u, v) | (u, v)$ supports $(s1, s2)\}$;
        end
    end
end
```

Function `relaxConstraints(`*context AC, matches M*`)`

```
$CL := \emptyset$
foreach $leaf$ node $z \in$ AC such that $\nexists(z, w) \in M$ do
    $u := z$;
    repeat
        $u := parent(u)$;
    until $u = null \vee \exists(u, v) \in M$ ;
    if $u \neq null$ then
        $CL := CL \cup \{(z, w) | w$ is a node in the subtree rooted at $v$,
        where $(z, w)$ is a type match and $w$ is not matched$\}$;
    end
end
return CL;
```

A problem with this direct conversion is that it requires each symbol in the abstract context must match a symbol in the target method. This requirement is needlessly strict for the unchanged context nodes. For instance, consider inserting a child of an `if` in the target. When the guard condition of the target `if` is a little different from the known `if`, i.e., `field != null` vs. `this.getField() != null`, exact graph isomorphism fails in this case. Although our algorithm is a little messy compared with an off-the-shelf labeled subgraph isomorphism algorithm [20], the heuristics for identifier replacements and siblings alignment work well in practice. Specifying which node matches to relax, and when and how to relax them is the key contribution of the algorithm we present above.

3.2.3 Generating Concrete Edits

To generate the concrete edit script Δ_B for mB, Sydit substitutes symbolic names used in Δ and recalculates edit positions with respect to the concrete nodes in mB. This process reverses the abstraction performed in Section 3.1.3.

The substitution is based on the symbolic identifier mappings established in Section 3.2.1, e.g., (T1, T2), and the abstract-concrete identifier mappings established in Section 3.1.3, e.g., (T1, int), (T2, int). For this specific case, each time T1 occurs in Δ, Sydit uses int in Δ_B.

Some edits in Δ use symbolic identifiers that only exist in the new version, thus the name does not exist in the original code of mA$_o$ or mB$_o$ and this name thus has no match. In this case, we borrow the identifier name from mA$_n$. For example, the identifier inValid used in Figure 1 only exists in mA$_n$, and is not in mB$_o$, nor should we ever expect it to appear in mA$_o$. We thus just use the name from mA$_n$, stored in Δ for Δ_B.

We make edit positions concrete with respect to the concrete nodes in mB. For instance, with the node match (u, v), an abstract edit which inserts a node after u is translated to a concrete edit which inserts a node after v. Using the above algorithms, Sydit produces the following concrete edits for mB (see Figure 4).

1. update (O6, N3), N3 = ‘ILaunchConfiguration cfg = null;’
2. move (O6, N1, 1)
3. insert (N7, N5, 0),
   N7 = ‘cfg = (ILaunchConfiguration) iter.next();’
4. insert (N8, N5, 1), N8 = ‘if (!cfg.inValid())’
5. insert (N9, N8, 0), N9 = then
6. insert (N10, N9, 0), N10 = ‘cfg.reset();’

This edit script shows that mB is changed similarly to mA. It differs because of the move (O6, N1, 1), which puts the designated node in a different location compared to mA. This difference does not compromise the edit's correctness since it respects the relevant data dependence constraints encoded in Δ. Sydit then converts Δ_B to Eclipse AST manipulations to produce mB$_s$.

4. Evaluation

To assess the coverage and accuracy of Sydit, we create an oracle data set of 56 pairs of example edits from open source projects, which we refer to simply as the *examples*. To examine the capabilities of Sydit, we select a range of simple to complex examples, and show that Sydit produces accurate edits across the examples. We compare Sydit to common *search and replace* text editor functionality and demonstrate that Sydit is much more effective. We evaluate the sensitivity of Sydit to the source and target method. Most correct edits are insensitive to this choice, but when there is a difference, choosing a simpler edit as the source method typically leads to higher coverage. We also study the best way to characterize edit context. We find that more context does not always yield more accurate edits. In fact, minimal, but non-zero context seems to be the sweet spot that leads to higher coverage and accuracy. Configuring Sydit to use an *upstream* context with $k = 1$ yields the highest coverage and accuracy on our examples.

For the evaluation data set, we collected 56 method pairs that experienced similar edits. We included 8 examples from a prior study of systematic changes to code clones from the Eclipse jdt.core plug-in and from jEdit [16]. We collected the remaining 48 examples from 42 releases of the Eclipse compare plug-in, 37 releases of the Eclipse core.runtime plug-in, and 50 releases of the Eclipse debug plug-in. For each pair, we computed the syntactic differences with Change Distiller. We identified method pairs mA and mB that share at least one common syntactic edit between the old and new version and their content is at least 40% similar. We use the following similarity metric:

	Single node	Multiple nodes	
		Contiguous	Non-contiguous
Identical	SI	CI	NI
examples	7	7	11
matched	5	7	8
compilable	5	7	8
correct	5	7	8
coverage	71%	100%	73%
accuracy	71%	100%	73%
similarity	100%	100%	100%
Abstract	SA	CA	NA
examples	7	12	12
matched	7	9	10
compilable	6	8	9
correct	6	6	7
coverage	100%	75%	83%
accuracy	86%	50%	58%
similarity	86%	95%	95%
Total coverage	82%	(46/56)	
Total accuracy	70%	(39/56)	
Total similarity	96%	(46)	

Table 1. Sydit's capabilities, coverage, and accuracy for k=1, upstream control and data dependences

$$\texttt{similarity(mA, mB)} = \frac{|matchingNodes(mA, mB)|}{size(mA) + size(mB)} \tag{1}$$

where matchingNodes(mA, mB) is the number of matching AST node pairs computed by ChangeDistiller, and size(mA) is the number of AST nodes in method mA.

We manually inspected and categorized these examples based on (1) whether the edits involve changing a *single* AST node vs. *multiple* nodes, (2) whether the edits are *contiguous* vs. *non-contiguous*, and (3) whether the edits' content is *identical* vs. *abstract*. An abstract context requires type, method, or variable name abstraction. To test this range of functionality in Sydit, we chose at least 7 examples in each category. Table 1 shows the number of examples in each of these six categories. The systematic change examples in the data set are non-trivial syntactic edits that include on average 1.66 inserts, 1.54 deletes, 1.46 moves, and 0.70 updates.

Coverage and accuracy. For each method pair (mA$_o$, mB$_o$) in the old version that changed similarly to become (mA$_n$, mB$_n$) in the new version, Sydit generates an abstract, context-aware edit script from mA$_o$ and mA$_n$ and tries to apply the learned edits to the target method mB$_o$, producing mB$_s$. In Table 1, *matched* is the number of examples for which Sydit matches the learned context to the target method mB$_o$. The *compilable* row is the number of examples for which Sydit produces a syntactically-valid program, and *correct* is the number of examples for which Sydit replicates edits that are semantically identical to what the programmer actually did. *Coverage* is $\frac{matched}{examples}$, and *accuracy* is $\frac{correct}{examples}$. We also measure syntactic *similarity* between Sydit's output and the expected output according to the above similarity formula (1).

This table uses our best configuration of k=1, upstream context only, i.e., one source node for each control and data dependence edge in the context, in addition to including a parent node of each edit. For this configuration, Sydit matches the derived abstract

A_{old} to A_{new}

```
private void paintSides(GC g, MergeSourceViewer tp,
  Canvas canvas, boolean right) {
  ...
- g.setLineWidth(LW);
 + g.setLineWidth(0 /* LW */);
  ...
}
```

B_{old} to B_{new}

```
private void paintCenter(Canvas canvas, GC g) {
  ...
  if (fUseSingleLine) {
    ...
  - g.setLineWidth(LW);
   + g.setLineWidth(0 /* LW */);
  ...
  } else {
    if(fUseSplines){
      ...
    - g.setLineWidth(LW);
    +  g.setLineWidth(0 /* LW */);
      ...
      } else {
        ...
      - g.setLineWidth(LW);
      + g.setLineWidth(0 /* LW */);
      }
    }  ...
}
```

Figure 5. A non-contiguous identical edit script (NI) for which SYDIT cannot match the change context (org.eclipse.compare: v20060714 vs. v20060917)

A_{old} to A_{new}

```
public IActionBars getActionBars() {
 + IActionBars actionBars = fContainer.getActionBars();
 - if (fContainer == null) {
 + if (actionBars == null && !fContainerProvided) {
     return Utilities.findActionBars(fComposite);
   }
   - return fContainer.getActionBars();
    + return actionBars;
}
```

B_{old} to B_{new}

```
public IServiceLocator getServiceLocator() {
 + IServiceLocator serviceLocator
     = fContainer.getServiceLocator();
  - if (fContainer == null) {
 + if (serviceLocator == null && !fContainerProvided) {
     return Utilities.findSite(fComposite);
   }
    - return fContainer.getServiceLocator();
    + return serviceLocator;
}
```

B_{old} to $B_{suggested}$

```
public IServiceLocator getServiceLocator() {
 + IServiceLocator actionBars
     = fContainer.getServiceLocator();
  - if (fContainer == null) {
 + if (actionBars == null && !fContainerProvided) {
     return Utilities.findSite(fComposite);
   }
    - return fContainer.getServiceLocator();
    + return actionBars;
}
```

Figure 6. A non-contiguous, abstract edit script for which SYDIT produces edits equivalent to the developer's (org.eclipse.compare: v20061120 vs. v20061218)

SI: single, identical edit		
8 targets	8 matched	8 correct
100% coverage (8/8)	100% accuracy (8/8)	100% similarity
CI: contiguous, identical edits		
5 targets	4 matched	4 correct
80% coverage (4/5)	80% accuracy (4/5)	100% similarity
NI: non-contiguous, identical edits		
6 targets	4 matched	0 correct
67% coverage (4/6)	0% accuracy (0/6)	67% similarity
SA: single, abstract edit		
5 targets	5 matched	5 correct
100% coverage (5/5)	100% accuracy (5/5)	100% similarity
CA: contiguous, abstract edits		
4 targets	4 matched	4 correct
100% coverage (4/4)	100% accuracy (4/4)	100% similarity
NA: non-contiguous, abstract edits		
4 targets	4 matched	4 correct
100% coverage (4/4)	100% accuracy (4/4)	100% similarity

Table 2. Replicating similar edits to multiple contexts

context for 46 of 56 examples, achieving 82% coverage. In 39 of 46 cases, the edits are semantically equivalent to the programmer's hand editing. Even for those cases in which SYDIT produces a different edit, the output and the expected output are often similar. For the examples SYDIT produces edits, on average, its output is 96% similar to the version created by a human developer.

In the examples where SYDIT cannot match the abstract context, the target method was usually very different from the source method, or the edit script needs to be applied multiple times in the target method. In Figure 5 (from org.eclipse.compare: v20060714 vs. v20060917), g.setLineWidth(LW) was replaced with g.setLineWidth(0) once in the source method. The same edit needs to be replicated in three different control-flow contexts in the target. Additional user assistance would solve this problem.

Figure 6 shows a complex example (from org.eclipse.compare v20061120 vs. v20061218) that SYDIT handles well. Although the methods mA_o and mB_o use different identifiers, SYDIT successfully matches mB_o to the abstract context AC derived from mA, creating a version mB_s, which is semantically equivalent to the manually crafted version mB_n.

In addition to these 56 pairs, we collected six examples that perform similar edits on multiple contexts—on at least 5 different methods. Table 2 shows the results. In four out of six categories, SYDIT correctly replicates similar edits to all target contexts. In the CI category, SYDIT misses one of five target methods because the target does not fully contain the inferred abstract context. In the NI category, SYDIT produces incorrect edits in two out of six targets because it inserts statements before the statements that define variables used by the inserts, causing a compilation error. To prevent undefined uses, SYDIT should, and in the future will, adjust its insertion point based on data dependences.

Comparison with search and replace. The *search and replace* (S&R) feature is the most widely used approach to systematic editing. Though SYDIT's goal is not to replace S&R but to complement it, we nevertheless compare them to assess how much additional capability SYDIT provides for automating repetitive edits. 32 of the 56 examples in our test suite require non-contiguous and abstract edit scripts. S&R cannot perform them in a straightforward manner because even after a developer applies one or more S&R actions, he or she would have to customize either the type, method, and/or variable names. For those 32 examples, SYDIT produces correct edits in 20 cases. For the remaining 24 examples, we categorize typical user-specified S&R sophistication into three levels:

- Level 1: Search for a single line and replace it.
- Level 2: Search for several contiguous lines and replace them.
- Level 3: Perform multiple S&R operations to modify several non-contiguous lines.

On the remaining 24 examples, Sydit handles 7 of 11 Level 1 examples, 5 of 5 in Level 2, 7 of 8 in Level 3. Even though Level 1 examples are straightforward with S&R, Sydit misses cases like the one in Figure 5. Overall, Sydit is much more effective and accurate than S&R.

Self application of a derived edit script. To assess whether Sydit generates correct program transformations from an example, we derive an edit script from $\mathtt{mA}_o$ and $\mathtt{mA}_n$ and then apply it back to $\mathtt{mA}_o$. We then compare the Sydit generated version with $\mathtt{mA}_n$. Similarly, we derive an edit script from $\mathtt{mB}_o$ and $\mathtt{mB}_n$ and compare the application of the script to $\mathtt{mB}_o$ with $\mathtt{mB}_n$. In our experiments, Sydit replicated edits correctly in *all* 112 cases.

Selection of source and target method. Sydit currently requires the user to select a source and target method. To explore how robust Sydit is to which method the user selects, we switched the source and target methods for each example. In 35 of 56 examples (63%), Sydit replicates edit scripts in both directions correctly. In 9 of 56 examples (16%), Sydit could not match the context in either direction. In 7 out of 56 examples (13%), Sydit replicates edit scripts in only one direction. In the failed cases, the source method experiences a super set of the edits needed in the target. Additional user guidance to select only a subset of edits in the source would solve this problem.

Context characterization. Table 3 characterizes the number of AST nodes and dependence edges in each edit script with the best configuration of $k = 1$ upstream only dependences. On average, an edit script involves 7.66 nodes, 2.77 data dependence edges, 5.63 control dependence edges, 5.04 distinct type names, 4.07 distinct method names, and 7.16 distinct variable names. These results show that Sydit creates and uses complex abstract contexts.

Table 4 explores how different context characterization strategies affect Sydit's coverage, accuracy, and similarity for the 56 examples. These experiments vary the amount of control and data dependence context, but always include the containment context (see Section 3.1.2).

The first part of the table shows that Sydit's results degrade slightly as the number of hops of control and data dependence chains in the context increases. $k = 1$ selects context nodes with one direct upstream or downstream control or data dependence on any edited node. We hypothesized that the inclusion of more contextual nodes would help Sydit produce more accurate edits without sacrificing coverage. Instead, we found the opposite.

The second part of Table 4 reports on the effectiveness of identifier abstraction for variable (V), method (M), and type (T) names. As expected, abstracting all three leads to the highest coverage, while no abstraction leads to the lowest coverage.

The third part of the same table shows results when varying the setting of upstream and downstream dependence relations for $k = 1$. All uses both upstream and downstream dependence relations to characterize the context, containment only neither uses upstream nor downstream data or control dependences, and upstream only uses only upstream dependence relations. Surprisingly, upstream only—which has neither the most nor fewest contextual nodes—gains the best coverage and accuracy.

ChangeDistiller similarity threshold. Sydit uses ChangeDistiller to compute AST-level edits between two program versions. When comparing the labels of AST nodes, ChangeDistiller uses a bigram similarity threshold and if the similarity between two node labels is greater than σ, it matches the nodes. Our experiments use a default setting of 0.5 for σ. Since our edit script generation capability depends heavily on ChangeDistiller's ability to compute syntactic edits accurately in the source example, we experimented with different settings of σ. Table 5 shows that when σ is in the range of 0.3 to 0.6, Sydit's accuracy does not change. When σ is 0.2, the relaxed similarity criterion leads AST node mismatches, which produce incorrect updates or moves, and consequently Sydit's coverage, accuracy and similarity decrease.

Size	**Min**	**Max**	**Median**	**Average**
nodes	1	56	3.5	7.66
data dependences	0	34	0.5	2.77
control dependences	1	38	3	5.63
Abstraction				
types	0	17	4	5.04
methods	0	17	2	4.07
variable	0	26	4.5	7.16

Table 3. Sydit's context characterization

	matched	correct	% coverage	% accuracy	% similarity
Varying the number of dependence hops					
k=1	44	37	79%	66%	95%
k=2	42	35	75%	63%	95%
k=3	42	35	75%	63%	95%
Varying the abstraction settings					
abstract V T M	46	39	82%	70%	96%
abstract V	37	31	66%	55%	55%
abstract T	37	31	66%	55%	55%
abstract M	45	38	80%	68%	96%
no abstraction	37	31	66%	55%	55%
Control, data, and containment vs. containment only vs. upstream only					
all (k=1)	44	37	79%	66%	95%
containment only	47	38	84%	68%	90%
upstream only (k=1)	46	39	82%	70%	96%

Table 4. Sydit's sensitivity to context characterization

σ	matched	correct	% coverage	% accuracy	% similarity
0.6	46	39	82%	70%	96%
0.5	46	39	82%	70%	96%
0.4	46	39	82%	70%	96%
0.3	46	39	82%	70%	96%
0.2	45	33	80%	59%	86%

Table 5. Sydit's sensitivity to input threshold σ used in ChangeDistiller

In summary, Sydit has high coverage and accuracy, and is relatively insensitive to the thresholds in ChangeDistiller and the number of dependences in the context. The best configuration is upstream with $k = 1$ for Sydit and σ = 0.5 for ChangeDistiller, which together achieve 82% coverage, 70% accuracy, and 96% similarity.

5. Related Work

The related work includes program differencing, source transformation languages, simultaneous text editing, and example-based program correction.

Program differencing. Program differencing takes two program versions and matches names and structure at various granularities, e.g., lines [14], abstract syntax tree nodes [9, 32], control-flow graph nodes [3], and program dependence graph nodes [13]. For example, the ubiquitous tool *diff* computes line-level differences per file using the longest common subsequence algorithm [14]. JDiff computes CFG-node level matches between two program versions based on similarity in node labels and nested hammock structures [3]. ChangeDistiller computes syntactic differences using a hierarchical comparison algorithm [9]. It matches statements, such as method invocations, using *bigram string similarity*, and control structures using *subtree similarity*. It outputs tree edit operations—*insert, delete, move,* and *update*. More advanced tools group sets of related differences with similar structural characteristics and find exceptions to identify potentially inconsistent updates [15, 17]. SYDIT extends ChangeDistiller and goes beyond these approaches by deriving an edit script from program differences, abstracting the script, and then applying it elsewhere.

Refactoring. Refactoring is the process of changing a software system that does not alter the external behavior of the code, yet improves the internal structure [21]. Refactorings often require applying one or more elementary transformations to multiple code locations, and refactoring engines in IDEs automate many common types of refactorings such as *replace a magic number with a constant* [10, 26]. While refactoring engines are confined to predefined, semantic-preserving transformations, SYDIT can automate semantic-modifying transformations.

Source transformation languages. Source transformation tools reduce programmer burden by exhaustively applying repetitive, error-prone updates. Programmers use special syntax to specify the code location and transformation [8, 11, 18, 27, 30]. The most ubiquitous approach is simple text substitution, e.g., find-and-replace in Emacs. More sophisticated systems use program structure information. For example, A* and TAWK expose syntax trees and primitive data structures, and Stratego/XT uses algebraic data types and term pattern matching [11, 18, 30]. TXL borrows syntax from the underlying programming language to express systematic tree edits [5]. These tools require programmers to understand low-level program representations. To make this approach easier for programmers, Boshernitsan et al. provide a visual language and an interactive source transformation tool [4]. All these tools require programmers to plan and create edit scripts, whereas SYDIT generates an abstract program transformation from an example edit.

Simultaneous editing. Simultaneous text editing automates repetitive editing [7, 22, 29]. Users interactively demonstrate their edit in one context and the tool replicates *identical lexical* edits on the pre-selected code fragments. In contrast, SYDIT learns an edit script from program differences and performs *similar yet different* edits by instantiating a *syntactic*, context-aware, abstract transformation. The Clever version control system detects inconsistent changes in clones and propagates *identical* edits to inconsistent clones [24]. While Clever and SYDIT both replicate similar edits, SYDIT exploits program structure and generates abstracts edits applicable to contexts using different identifiers.

Suggesting edit locations. LibSync helps client applications migrate library API usages by learning migration patterns [23] with respect to a partial AST with containment and data dependences. Though it suggests what code locations to examine and shows example API updates, it is *unable* to transform code. Furthermore, its flexibility is limited by its inability to abstract variable, method, and type names.

FixWizard identifies code clones based on object usage and interactions, recognizes recurring bug-fixes to the clones, and suggests a location and example edit [25]. FixWizard identifies edit locations automatically only in pre-identified clones. It does *not generate syntactic edits*, nor does it support abstraction of variables, methods, and types. These limitations leave programmers with the burden of manually editing the suggested fix-location, which is error-prone and tedious.

Example based program migration and correction. Programming-by-example [19] (PBD) is a software agent-based approach that infers a generalized action script that corresponds to user's recorded actions. SMARTedit [19] instantiates this PBD approach to automate repetitive text edits by learning a series of functions such as *'move a cursor to the end of a line.'* However, this approach is not suitable for editing a program as it does not consider a program's syntax, control, or data dependences.

The most closely related work focuses on API migration [2, 23]. Andersen and Lawall find differences in the API usage of client code, create an edit script, and transform programs to use updated APIs [1, 2, 27]. Compared to SYDIT, their approach is limited in two respects: the edit scripts are confined to term-replacements and they only apply to API usage changes. Similar to our approach, Andersen and Lawall use control and data dependence analysis to model the context of edits [1]. However, the context includes only inserted and deleted API method invocations and control and data dependences among them. Their context does not include unchanged code on which the edits depend. Thus, when there is no deleted API method invocation, the extracted context cannot be used to position edits in a target method. SYDIT is more flexible because it computes edit context that is not limited to API method invocations and it can include unchanged statements related to edits. Therefore, even if the edits include only insertions, SYDIT can correctly position edits by finding corresponding context nodes in a target method. Furthermore, Andersen and Lawall only evaluate their approach on a few examples, whereas we perform a comprehensive evaluation on open-source applications.

Automatic program repair generates candidate patches and checks correctness using compilation and testing [28, 31]. For example, it generates patches that enforce invariants observed in correct executions but are violated in erroneous executions [28]. It tests patched executions and selects the most successful patch. Weimer et al. [31] generate their candidate patches by replicating, mutating, or deleting code *randomly* from the existing program and thus far have focused on single line edits. SYDIT automates sophisticated multi-line edits that can add functionality or fix bugs. Integrating SYDIT into a testing framework to automate validation is a promising future direction.

6. Discussions and Conclusions

SYDIT is the first tool to perform non-contiguous, abstract edits to different contexts, significantly improving the capabilities of the state-of-the-practice developer tools such as line-based GNU patch or the search and replace feature in text editors. This approach is however amenable to additional user guidance and automation.

To learn and apply a systematic edit, users must provide a source and target method. It would be relatively straight-forward to extend SYDIT to help programmers select a subset of edits to be replicated. Users may also want to configure SYDIT to update multiple locations within a target method, or to select a specific location to perform learned edits.

SYDIT does not recognize naming patterns between related types and variables such as `IServiceLocator` and `serviceLocator`, as shown in Figure 6. Thus, developers may not easily understand the output, even when SYDIT produces semantically equivalent code. By leveraging systematic naming patterns in program differences [17], it should be possible to produce edits that better mimic human developers.

SYDIT relies on ChangeDistiller to detect syntactic differences between two versions. In some cases, ChangeDistiller fails to report a minimal concrete edit and thus SYDIT's derived edit script may include superfluous contextual nodes. For example, instead of selecting one contextual node relevant to an update, it may select multiple contextual nodes relevant to an insert and a corresponding delete. We leave to future work how the choice of program differencing algorithm affects the flexibility of the learned edit scripts.

This paper focuses on single method updates, but it may be possible to generalize the approach for higher-level changes. For example, Kim et al. show that a large percentage of API-level refactorings and class hierarchy changes consist of similar edits to different class and method contexts [15, 17]. For instance, the *extract super class* refactoring moves a set of related fields and methods from subclasses to a super class. This type of functionality will require more sophisticated context representations and matching algorithms.

Another area for future work is automated target selection. For example, exhaustively examining every method in the program may prove useful. When a programmer fixes a bug, SYDIT could generate an edit, then apply it to all applicable code regions, and test the SYDIT generated version. While prior work suggests edit locations for recurring bug-fixes [25], SYDIT could actually apply the edit, automating some program repair and modification tasks. Furthermore, library component developers could use SYDIT to automate API usage updates in client applications by shipping an edit script together with changed library components.

In summary, SYDIT provides needed functionality that helps developers make simple and complex changes that span large programs. By using context extraction, identifier abstraction, and edit position abstraction, SYDIT learns and applies edits with high coverage and accuracy. This approach for program evolution opens a new way of providing higher confidence to developers when they add features and fix bugs.

A. Appendix

We include example edits to show the limitations and power of the current implementation of SYDIT.

Figure 7 shows one of the reasons that prevent SYDIT from mapping an abstract context with a target method. Both source and target share all changes except deletion of line 14 and insertion of line 19. In `mA`, line 14 is a `return` statement, while in `mB`, line 14 is an expression statement. The two statements have different AST node types. As a result, they cannot be matched by SYDIT.

Figure 10 shows an example in which SYDIT establishes matches successfully but produces incorrect edits. In this case, both `mA` and `mB` have a method invocation replaced with the other method invocation. However, when calling the new method, `mA` and `mB` pass different input arguments: `monitor` object in the source vs. `null` in the target. The target method does not provide enough clues on why `null` must be passed as an argument.

Figure 8 shows an example that SYDIT produces edits that are not entirely identical to the programmer's actual edits, because the context and the edit content are slightly different between the source and the target. In both methods, the programmer updates some statements, moves some statements out of the `for` loop, and inserts and deletes some statements within the loop to make programs more concise. However, the task involves different number of edits in the two methods. When SYDIT replicates learned edits to the target, three lines in $\texttt{mB}_{suggested}$ diverge from $\texttt{mB}_{new}$ (see Figure 9): (1) line 19 is incorrectly inserted with `LIST_ENTRY`; (2) line 22 is not deleted since it does not have a counterpart in the abstract context and there is no edit dealing with it; (3) line 25 is incorrectly inserted since `map.put(...)` is not mapped correctly to `list.add(...)`. Despite the three incorrect edits, SYDIT still makes the rest of the edits correctly in `mB`, alleviating part of this programming task.

A_{old} to A_{new}

```
1. public boolean isMigrationCandidate
     (ILaunchConfiguration candidate) throws CoreException {
2. - if(getAttribute(MIGRATION_DELEGATE) != null) {
3.   - if(fDelegates == null) {
4.     - fDelegates = new Hashtable();
5.   - }
6.   - Object delegate = fDelegates.get(MIGRATION_DELEGATE);
7.   - if(delegate == null) {
8.     - delegate = getConfigurationElement()
9.       .createExecutableExtension(MIGRATION_DELEGATE);
10.    - fDelegates.put(MIGRATION_DELEGATE, delegate);
11.  - }
12.  - if(delegate instanceof
13.      ILaunchConfigurationMigrationDelegate) {
14.    - return ((ILaunchConfigurationMigrationDelegate)
15.      delegate).isCandidate(candidate);
16.  - }
17.  + initializeMigrationDelegate();
18.  + if(fMigrationDelegate != null) {
19.    + return fMigrationDelegate.isCandidate(candidate);
20.  }
21.  return false;
22.}
```

B_{old} to B_{new}

```
1. public void migrate(ILaunchConfiguration candidate)
     throws CoreException {
2. - if(getAttribute(MIGRATION_DELEGATE) != null) {
3.    - if(fDelegates == null) {
4.      - fDelegates = new Hashtable();
5.    - }
6.    - Object delegate = fDelegates.get(MIGRATION_DELEGATE);
7.    - if(delegate == null) {
8.      - delegate = getConfigurationElement()
9.        .createExecutableExtension(MIGRATION_DELEGATE);
10.     - fDelegates.put(MIGRATION_DELEGATE, delegate);
11.   - }
12.   - if(delegate instanceof
13.     ILaunchConfigurationMigrationDelegate) {
14.   - ((ILaunchConfigurationMigrationDelegate)
15.     delegate).migrate(candidate);
16.  - }
17.  + initializeMigrationDelegate();
18.  + if(fMigrationDelegate != null) {
19.    + fMigrationDelegate.migrate(candidate);
20.  }
21.  return false;
22.}
```

Figure 7. A contigous, abstract edit script (CA) for which SYDIT cannot match the change context

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.

References

[1] J. Andersen. *Semantic Patch Inference*. Ph.D. Dissertation, University of Copenhagen, Copenhagen, Nov. 2009. Adviser-Julia L. Lawall.

[2] J. Andersen and J. L. Lawall. Generic patch inference. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2187-9.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2131-2.

A_{old} to A_{new}

```
protected void setMapAttribute(Element element)
    throws CoreException {
- String listKey = element.getAttribute("key");
+ String listKey = element.getAttribute(KEY);
    NodeList nodeList = element.getChildNodes();
    int entryCount = nodeList.getLength();
    List list = new ArrayList(entryCount);
+ Node node = null;
+ Element selement = null;
    for (int i = 0; i < entryCount; i++) {
    - Node node = nodeList.item(i);
    + node = nodeList.item(i);
    - short type = node.getNodeType();
    - if (type == Node.ELEMENT_NODE) {
      - Element subElement = (Element) node;
      - String nodeName = subElement.getNodeName();
      - if (!nodeName.equalsIgnoreCase("listEntry")) {
    + if (node.getNodeType() == Node.ELEMENT˙NODE) {
      + selement = (Element) node;
      + if (!selement.getNodeName().
          equalsIgnoreCase(LIST_ENTRY)) {
          throw getInvalidFormatDebugException();
        }
          - String value = getValueAttribute(subElement);
          - list.add(value);
          + list.add(getValueAttribute(selement));
      }
    }
    setAttribute(listKey, list);
}
```

B_{old} to B_{new}

```
protected void setMapAttribute(Element element)
    throws CoreException {
- String mapKey = element.getAttribute("key");
 + String mapKey = element.getAttribute(KEY);
      NodeList nodeList = element.getChildNodes();
      int entryCount = nodeList.getLength();
      Map map = new HashMap(entryCount);
    + Node node = null;
    + Element selement = null;
      for (int i = 0; i < entryCount; i++) {
      - Node node = nodeList.item(i);
       + node = nodeList.item(i);
      - short type = node.getNodeType();
      - if (type == Node.ELEMENT_NODE) {
        - Element subElement = (Element) node;
        - String nodeName = subElement.getNodeName();
        - if (!nodeName.equalsIgnoreCase("mapEntry")) {
       + if (node.getNodeType() == Node.ELEMENT˙NODE) {
         + selement = (Element) node;
         + if (!selement.getNodeName().
             equalsIgnoreCase(MAP_ENTRY)) {
                 throw getInvalidFormatDebugException();
          }
            - String key = getKeyAttribute(subElement);
            - String value = getValueAttribute(subElement);
            - map.put(key, value);
             + map.put(getKeyAttribute(selement),
                 getValueAttribute(selement));
        }
    }
    setAttribute(mapKey, map);
}
```

Figure 8. A non-contiguous, abstract edit script example

[4] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 567–576, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9.

[5] J. R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. ISSN 0167-6423.

B_{old} to $\mathrm{B}_{suggested}$

```
protected void setMapAttribute(Element element)
    throws CoreException {
- String mapKey = element.getAttribute("key");
+ String mapKey = element.getAttribute(KEY);
    NodeList nodeList = element.getChildNodes();
    int entryCount = nodeList.getLength();
    Map map = new HashMap(entryCount);
+ Node node = null;
+ Element selement = null;
    for (int i = 0; i < entryCount; i++) {
    - Node node = nodeList.item(i);
     + node = nodeList.item(i);
    - short type = node.getNodeType();
    - if (type == Node.ELEMENT_NODE) {
      - Element subElement = (Element) node;
      - String nodeName = subElement.getNodeName();
      - if (!nodeName.equalsIgnoreCase("mapEntry")) {
     + if (node.getNodeType() == Node.ELEMENT˙NODE) {
       + selement = (Element) node;
       + if (!selement.getNodeName()
          .equalsIgnoreCase(LIST˙ENTRY)) {
            throw getInvalidFormatDebugException();
        }
        String key = getKeyAttribute(subElement);
        - String value = getValueAttribute(subElement);
        - map.put(key, value);
         + map.add(getValueAttribute(selement));
      }
    }
    setAttribute(mapKey, map);
}
```

Figure 9. A non-contiguous, abstract edit script for which SYDIT produces output different from the developer's version

A_{old} to A_{new}

```
public void flush(IProgressMonitor monitor){
  - saveContent(getInput());
  + flushContent(getInput(), monitor);
}
```

B_{old} to B_{new}

```
public void run() {
  - saveContent(getInput());
  + flushContent(getInput(), null);
}
```

Figure 10. A single, abstract edit script (SA) for which SYDIT cannot produce correct edits

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[7] 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. ISBN 0-7695-2828-7.

[8] M. Erwig and D. Ren. A rule-based language for programming software updates. In *RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 67–78, New York, NY, USA, 2002. ACM. ISBN 1-58113-606-4.

[9] 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.

[10] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[11] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension*, page 144, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7283-8.

[12] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2.

[13] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7.

[14] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5): 350–353, 1977. ISSN 0001-0782.

[15] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4.

[16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0.

[17] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7.

[18] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21 (11):894–901, 1995. ISSN 0098-5589.

[19] H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers, 2001.

[20] A. Matzner, M. Minas, and A. Schulte. Efficient graph matching with application to cognitive automation. In A. Schrr, M. Nagl, and A. Zndorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 297–312. Springer Berlin / Heidelberg, 2008.

[21] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. ISSN 0098-5589.

[22] 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. ISBN 1-880446-09-X.

[23] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. Nguyen. A graph-based approach to api usage adaptation. In *OOPSLA '10: Proceedings of the 2010 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications*, page 10 pages, New York, NY, USA, 2010. ACM.

[24] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. doi: http://dx.doi.org/10.1109/ASE.2009.90.

[25] 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*, pages 315–324, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.

[26] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[27] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 247–260, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5.

[28] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3.

[29] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8696-5.

[30] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Domain-Specific Program Generation*, 3016:216–238, 2004.

[31] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4.

[32] W. Yang. Identifying syntactic differences between two programs. *Software – Practice & Experience*, 21(7):739–755, 1991.