Generating, Locating, and Applying Systematic Edits by Learning from Example(s)

Ph.D. Proposal

Na Meng
Department of Computer Science
University of Texas at Austin
Oct. 3, 2012

Abstract

Programmers make systematic edits—similar, but not identical changes to multiple places during software development and maintenance. Finding all the correct locations and making correct edits is a tedious and error-prone process. Existing tools for automating systematic edits are limited because they do not support edit generation, edit location suggestion, or edit application at the same time, except for specialized or trivial edits. However, since many similar changes are needed in locations that contain similar context (the surrounding dependent code and syntactic structures), there is an opportunity to automate the systematic editing process by inferring edit scripts and characterizing their context from code that developers changed already. The challenge is we need to abstract and generalize from example(s) in order to create an edit script that is correct in many different contexts. This thesis seeks to substantially improve the efficiency and correctness of automatic systematic program transformation. (1) We design and implement Sydit to generate an abstract, context-aware edit script from a single changed method, and apply it to user-selected target method(s). This approach correctly performs many edits, but we show that the edit scripts from one example are not well suited to finding new locations. (2) We thus design and implement Lase to generate a partially abstract, context-aware edit script from multiple changed methods and show how to use the same script to automatically find edit locations and apply edits. Our experiments compare the systematic edits generated by Lase and those created by developers and find that Lase is effective at finding correct locations and making correct edits. (3) We propose to explore a new approach to generate high-level, collaborative edit scripts that span more than one method to handle sophisticated program transformations, such as refactoring. (4) Finally, we propose an approach for checking the correctness of the transformations. Our work aims at relieving developers from tedious and error-prone code changes by performing systematic transformations in a consistent way, and by guiding automated program repair, feature addition, and refactoring.

1 Introduction

As applications become larger and more complex, it becomes more difficult to maintain them. Researchers have observed that a large proportion of changes made during software maintenance are systematic, i.e., similar but not identical changes to multiple places in the program [17, 16, 27]. Finding all the correct locations and making correct edits to them is a tedious and error-prone process. Existing tools for automating systematic transformations are limited because they do not perform edit generation, edit location suggestion, or edit application all in one, except for specialized or trivial edits. For instance, code pattern detection tools [8, 20, 21, 33] and code change pattern detection tools [27, 26] suggest locations to edit, but they do not generate or apply edits. Most source transformation languages and tools [5, 6, 11, 18, 35] suggest
edit locations and apply the edits prescribed by developers, but they cannot infer or generalize edits automatically. All the other existing tools are limited to specific edits, such as predefined semantic-preserving edits, or trivial edits which are independent of the program’s context. The hypothesis this thesis explores is whether nontrivial systematic program transformations can be automated. The reality is developers make a lot of systematic transformation consisting of similar edits to places containing similar context. Therefore, the opportunity we have is by inferring edit scripts and characterizing their context from code that developers changed already, we can find locations containing similar context but not changed by developers and propagate similar changes to them.

The Problem

The challenge to inferring an edit script and charactering its relevant context is that we need to abstract and generalize from example(s) in order to create an edit script, which is correct in many different contexts. The abstraction should be general enough so that the inferred edit script is applicable to code locations which use different variable, method, or type identifiers, and have different contexts, such as a for or a while loop. Meanwhile, the abstraction should also be specific enough so that the characterized edit-relevant context can be used to effectively find locations in need of similar changes. To meet the two conflicting requirements, we develop two approaches and plan to develop a third one for automatic program transformation based on example(s). Besides, we will explore an approach to automatically check whether the abstraction works in a meaningful way such that an abstract systematic edit script is always correctly applied to new contexts.

Our Solution

We designed and implemented an approach to derive an abstract, context-aware edit script from one changed method and apply the script to user-selected target methods. The approach is implemented in a tool called Sydit. It makes the abstraction general by replacing all concrete identifiers used in the exemplar changed method with abstract ones. Meanwhile, Sydit makes the abstraction specific by finding the edit context—surrounding dependent code for each edited statement—and putting it in the edit script. Experiments show that these edit scripts perform many systematic edits correctly, but the scripts are not always well suited to finding edit locations due to learning from a single example and fully abstracting identifiers.

To address the above approach’s limitation in automatically finding edit locations, we design and implement a second approach to derive a partially abstract, context-aware edit script from multiple changed methods, and an approach for using the same script to automatically find edit locations where to apply the edit. The approaches are implemented in a tool called Lase. It makes the abstraction general by extracting the common edit operations from all exemplar methods’ edit scripts, and abstracting the involved identifiers used in different scripts when they diverge. Meanwhile, Lase makes the abstraction specific by characterizing the edit context for the common edited statements and keeping the original identifiers when they are used identically in different scripts. Our experiments compare the systematic edits generated by Lase and those created by developers, showing that the approach is effective in both finding correct locations and making correct edits.

Sydit and Lase focus on systematic edits composed of fine-grained changes that insert, delete, update, and move statements inside methods. However, developers also perform many systematic program transformations that involve coarser grained changes consisting of collaborative systematic changes to different entities, such as refactoring. To handle these high-level structured changes, we propose to develop a new tool by extending our existing approaches to detect declaration changes, such as addition, deletion, update of fields, methods and types, and to explore a new approach to reason about relations between changed entities in order to correlate changes appearing in different context together. By constructing test suites of these sophisticated systematic program transformations, we will evaluate how effectively the tool can help developers maintain software in a consistent and efficient way.

Although we have always tried to infer the most appropriate edit script from example(s), we cannot guarantee the correctness of generated edit scripts. In order to provide developers some confidence in the suggested edit scripts’ correctness, for each target method, after customizing the systematic edit and applying
the suggested edit script, we will explore automatically regenerating a systematic edit from the method’s old and suggested versions and check whether it is equivalent to the original systematic edit applied. If it is, we are more confident about the suggested edit script’s correctness; otherwise, we are less confident.

Software maintenance costs programmers a lot of time and effort, a significant portion of which has been poured into making similar or collaborative systematic changes to multiple places. If all approaches in our solution provide important assistance to developers in making such changes, we can relieve developers from tedious and error-prone code changes and help improve software quality with higher efficiency and lower cost. However, if our solution has limitations, we will learn more about the obstacles stopping us from hitting our goal. We will know more about the easy cases which can be automatically handled well by tools, so that we only bring to developers’ attention the difficult cases which cannot be automatically handled well without human involvement.

2 Related Work

This section discusses existing approaches related to ours for assisting or automating systematic program transformations. They vary in the flexibility and capability of edit generation, edit location suggestion, as well as edit application, and the complexity of edits to handle.

The search-and-replace feature in a text editor and more sophisticated command-line tools, such as `sed` and `awk` [7] assist users to perform program transformation via text substitution based on keywords or regular-expression patterns. These approaches treat source code as flat and structureless text. Once programmers want to make structure-based transformation, such as replacing a conditional assignment with an `if`-statement; regular-expression patterns can be difficult to write and read, and restrict users to trivial edits.

Example-based text-editing tools automate repetitive editing tasks [25, 19, 34]. They learn edit operations, such as `copy-and-paste` and `insert a piece of text`, by observing a user performing his or her task(s), and propagate the exact operations to specified similar text. Compared with the tools mentioned above, these tools provide more automated support for text manipulation, but they put unnecessary format constraints on the text to change because they align and change text based on the format. DMP (Diff-Match-Patch) [1] is more sophisticated because it performs text differencing algorithm between two versions of a program in order to identify which words/characters are replaced by alternatives, creates a patch to record the detected text substitutions, and changes a given program by applying the patch. However, it still does not handle well structure-based transformation.

Refactoring is the process of changing a software system without altering the external behavior of the code, yet improves the internal structure [24]. 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]. However, refactoring engines are confined to a predefined set of semantics-preserving transformations.

Code pattern detection tools [8, 20, 21, 33] infer code patterns to find buggy code violating the patterns, pinpoint edit locations, and suggest correct patterns to help developers fix bugs. However, they do not change programs automatically.

Code change pattern detection tools [27, 26] infer program transformations or code change patterns from code change examples, pinpoint edit locations, and suggest code change examples to help developers transform programs systematically. However, they do not change programs automatically, either.

Source transformation languages and tools allow users to define source transformation by describing patterns to match and actions to take. Their transformation engines then automatically search for places matching the patterns in order to take the specified transformation actions. However, most of them [5, 6, 11, 18, 35] are difficult to use because they define manipulation primitives on a syntax tree or the grammar of an underlying programming language. In order to use the tools, programmers have to learn the languages and prescribe transformation from scratch. iXJ [4] is simpler to use because it allows programmers to define code transformations in a visualized way by selecting a piece of code to change, generalizing part of it with wildcards, and describing changes based on the generalized representation. However, it only handles trivial
update edits without concerning much about edit context. SmPL [28] is a semantic patch language which is very close to the C language and builds on the existing patch notation. Developers can use spdiff [3] to infer source transformation described with SmPL from examples and use Coccinelle [28] to apply it. This series of work is the one most related to ours, but it is mainly focused on API migration changes—collateral evolution of client applications to adapt to API changes. Besides, spdiff’s transformation inference algorithm cannot always correctly position edits, because it computes positions without considering data and control dependence constraints that edits have on their surrounding context.

Program synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints [12]. It has been used to synthesize and apply program transformation by learning from examples written in domain-specific languages, but has not been used to change programs written in any general-purpose language, such as Java. The reason is that the solution search algorithm underlying the synthesis algorithm does not scale well when a language’s grammar contains more rules.

Automatic program repair generates candidate patches and checks correctness using compilation and testing [30, 37, 36]. These tools are limited to trivial bug fixes requiring single line edits.

3 Completed Work

This section describes the completed portion of our work. Section 3.1 details SYDIT—a tool to generate an abstract, context-aware edit script from a single changed method and apply it to user-selected target method(s). We published this work at PLDI 2011 [23]. Section 3.2 introduces LASE—a tool to generate a partially abstract, context-aware edit script from multiple changed methods, find edit locations, and apply edits. We submitted this work to ICSE 2013.

3.1 Sydit

As described in Section 1, during software maintenance, programmers make a lot of changes to fix bugs, refactor, or add new features to existing code. Recent work observes that many of these changes are systematic—programmers add, delete, and modify code in numerous classes in similar but not identical ways [17, 16, 27]. These changes are tedious and error-prone, because programmers need to manually find all correct edit locations and make correct edit to each of them.

To help programmers with their systematic editing task, we design and implement an approach that generates an edit script from a single exemplar changed method, and applies the script to user-selected method(s). We implement the approach in a tool called SYDIT (Systematic Editing). When given an example method and one or more target methods to change similarly, SYDIT characterizes edits as Abstract Syntax Tree (AST) node additions, deletions, updates, and moves. It uses containment, 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 or concrete identifiers. We call these transformations abstract, context-aware edit scripts. For each target method, SYDIT generates concrete AST transformations customized to the new context, and finally transforms the code accordingly.

We evaluate SYDIT on a test suite of 56 systematic edit pairs drawn from five Java open source projects. For each systematic edit pair, SYDIT generates an edit script from one exemplar edit, which consists of an old and a new version of a changed method. It then generates concrete edits and applies them to the other changed method’s old version. By checking whether the generated version is equivalent to the method’s new version, we measure how well SYDIT can perform automatic program transformation tasks. 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. A syntactic program differencing algorithm 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.
A<sub>old</sub> to A<sub>new</sub> & SYDIT’s application of similar edits on B<sub>old</sub>, resulting in B<sub>new</sub>

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

1. protected List getLaunchConfigurations(IProject project) {
2. 
3. + if (config.getType().equals(type)) {
4. + config.reset();
5. +
6. + if (cfg.inValid()) {
7. + cfg.reset();
8. +
9. + 
10. + 11. 
12. + if (file != null && file.getProject().equals(project)) {
13. +
14. +
15. +
16. +
17. +

Figure 1: Systematic edit pair to show SYDIT’s functionality

3.1.1 Motivating Example

We use Figure 1 as a running example throughout this section. The code in black is the original code. The code annotated with ‘+’ and ‘−’ indicates additions and deletions, respectively. Consider the two methods mA and mB: getLaunchConfigurations(ILaunchConfigurationType type) and getLaunchConfigurations(IProject project). These two methods perform similar tasks: (1) both iterate over all elements received by calling getAllLaunchConfigurations(), (2) both process the elements one by one, and (3) when an element meets certain condition, both add it to a predefined list. Suppose that Pat intends to apply similar changes to mA and mB. In mA, he wants to move the declaration of variable config out of the while loop and add some code that processes config inside the loop. Lines 4, and 6-10 in mA show Pat’s additions and deletions. In mB, he wants to perform a similar edit, but on the cfg variable instead of config. This example typifies a systematic edit. Even though the changes are similar, without assistance, Pat must manually edit both methods, which is tedious and error-prone. Using SYDIT, Pat instead applies the changes only to mA and specifies mB as a target place to apply similar changes. SYDIT derives an abstract, context aware edit script from the changes in mA, customizes it to create a concrete edit script for mB and then applies it.

3.1.2 Approach

This section overviews the five steps of SYDIT. The first three steps create an abstract, context-aware edit script from an old and new version of exemplar changed method mA, i.e., mA<sub>o</sub> and mA<sub>n</sub>. The last two steps apply the edit script to any user-selected method, mB, producing a modified method mB<sub>s</sub>.

Step 1: Program Differencing

SYDIT compares the syntax trees of the exemplar edit, mA<sub>o</sub> and mA<sub>n</sub>, using a modified version of ChangeDistiller [9], to generate a sequence of edits as Abstract Syntax Tree (AST) node additions, deletions, updates, and moves, described as follows:

insert (Node u, Node v, int k): insert u and position it as the (k + 1)<sup>th</sup> child of v.
delete (Node u): delete u.
update (Node u, Node v): replace u’s label and AST type with v’s while maintaining u’s position.
moves (Node u, Node v, int k): delete u from its current position and insert it as v’s (k + 1)<sup>th</sup> child.
The resulting sequence of syntactic edits is $\Delta_A = \{e_i|e_i \in \text{insert } (u, v, k), \text{delete } (u), \text{update } (u, v), \text{move } (u, v, k)\}$. We use a total order for $e_i$ to ease relative positioning of edits. For our example, the inferred edit script $\Delta_A$ between $mA_o$ and $mA_n$ is:

1. update ("ILaunchConfiguration config = (ILaunchConfiguration) iter.next();", "ILaunchConfiguration config = null;")
2. move ("ILaunchConfiguration config = null;", "protected List getLaunchConfigurations(IProject project) {", 2)
3. insert ("config = (ILaunchConfiguration) iter.next();", "while (iter.hasNext()) {", 0)
4. insert ("if (!config.inValid()) {", "while (iter.hasNext()) {", 1)
5. insert ("then", "if (!config.inValid()) {", 0)
6. insert ("config.reset()", "then", 0)

**Step 2: Edit Context Extraction**

For each edit, SYDIT extracts relevant context from both the old and new versions using control, data, and containment dependence analysis. By context, we mean AST nodes on which an edited node depends or those dependent on the edited node. The extracted context nodes will serve as anchors to position edits correctly in a new target location in Step 4. We implement the analysis in Crystal static analysis framework [14]. For each $e_i \in \Delta_A$, the edited node and its parent node in $mA_o$’s AST are included as context. Besides, by default, we determine its extra relevant context nodes in $mA_o$ as follows:

- **insert** $(u,v,k)$: the algorithm computes nodes in $mA_o$ on which $u$ depends and then projects them to corresponding nodes in $mA_o$.
- **delete** $(u)$: the algorithm computes nodes in $mA_o$ on which $u$ depends.
- **update** $(u,v)$: the algorithm computes nodes on which $u$ depends in $mA_o$ and those on which $v$ depends in $mA_o$. It then projects context nodes found in $mA_o$ to corresponding nodes in $mA_o$, and unions them with nodes already found in $mA_o$.
- **move** $(u,v,k)$: the algorithm computes nodes in both $mA_o$ and $mA_n$ on which $u$ depends. The nodes in the new version help guarantee dependence relations after the move. It then projects the nodes found in $mA_n$ to corresponding nodes in $mA_o$ and finally unions the set with nodes already found in $mA_o$.

SYDIT allows users to configure the amount of context to extract for any edit by taking as input a parameter $k$—which controls the largest number of dependence hops between an edited node and its surrounding context, a flag $Up$—which controls whether to include the nodes on which any edited node is data or control dependent on, and a flag $Down$—which controls whether to include the nodes which are data or control dependent on any edited node. The default setting of SYDIT is $k = 1$, $Up = true$, $Down = false$. It means that we only include as context the surrounding nodes which any edited node is directly dependent on and those directly containing an edited node. We pick the default values based on the experiment results shown in Section 3.1.3. For the edit script of $mA$, using the default configuration, SYDIT extracts AST nodes corresponding to line 1-2 and 5-6 in $mA_o$ as relevant context.

**Step 3: Identifier and Edit Position Abstraction**

The edit script derived in Step 1 is described using concrete identifiers with respect to the concrete context in the exemplar changed method. In order to make the edit script more generalized and applicable to methods using different concrete identifiers or containing different contexts, SYDIT generalizes the edit script with identifier and edit position abstraction. With identifier abstraction, SYDIT replaces concrete identifiers of
variables, methods, and types with equivalent abstract representations: \( v^x \), \( m^x \), and \( t^x \) respectively. With edit position abstraction, SYDIT recalculates position of each edit with respect to the extracted context from Step 2, so that irrelevant context in the original syntax tree is ignored, taking no effect on positioning edits in a target method. The resulting abstract, context-aware edit script is shown in Figure 2, marked as ∆.

**Step 4: Context Matching**

Given a target method, SYDIT uses a context matching algorithm to find nodes in the method that match the context nodes in ∆ and induce one-to-one mappings between abstract and concrete identifier names. The established node mappings and identifier mappings will facilitate us to concretize the abstract edit script ∆ for the target method.

The mapping problem is similar to a labeled subgraph isomorphism problem, if we model both ∆’s edit context and the target method’s AST as graphs and label each node with the corresponding AST node’s abstract content based on identifier abstraction. However, we invented a specialized subtree matching algorithm customized to the needs of this problem instead of using an off-the-shelf algorithm because of two reasons. First, graph algorithms usually ignore the ordering relations between siblings under the same node, causing wrong matching results in some cases. Second, they explore graphs along one edge at a time, unable to correlate sibling matching results with each other, causing very poor performance.

The intuition behind our algorithm is, given an edit context \( AC \) and target method \( mB \), we first find candidate leaf matches between leaf nodes in \( AC \) and leaf nodes in \( mB \), and then use them to match inner nodes. For each path \( p_{AC} \) from a leaf to the root of \( AC \), we tries to find the best match in \( mB — p_B \)—for it so that every node on \( p_{AC} \) maps nodes on \( p_B \) in sequence. If in some cases, there is more than one candidate path in \( mB \) matching \( p_{AC} \), SYDIT tries to break the tie by referring to known sibling matches of the undecided node pairs. If every node in \( AC \) finds a unique correspondence in the target tree, SYDIT collects the identifier mappings based on the node matches and proceeds to the next step.

**Step 5: Concrete Edits Generation and Application**

To generate concrete edits for a target method, SYDIT replaces symbolic names used in the abstract edit script with corresponding concrete identifiers in the identifier mappings established in Step 4. It recalculates each edit position with respect to the concrete target method based on the node matches established in Step 4. These two processes of edit concretization are just the opposite operations from Step 2 for edit abstraction. Finally, SYDIT translates the resulting edits to a sequence of AST rewrite operations step by step using the Eclipse ASTRewrite API, applies the operations, and provides a suggested version for the given target method, as shown on the right hand side of Figure 1. By examining the suggested version, users can decide whether to take the suggestion or not. If they decide to take the suggestion, the method’s original version is automatically replaced with the suggested version. Then users can make some extra edits on top of that if they want.

![Figure 2: Edit script generated from getLaunchConfigurations(ILaunchConfigurationType type) by SYDIT](image-url)
### Table 1: \textsc{Sydit}'s effectiveness in terms of coverage, accuracy, and similarity for $k=1$, $Up=true$, $Down=false$

<table>
<thead>
<tr>
<th></th>
<th>Single node</th>
<th>Multiple nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Identical</td>
<td>Contiguous</td>
</tr>
<tr>
<td>examples</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>matched</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>compilable</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>correct</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>coverage</td>
<td>71%</td>
<td>100%</td>
</tr>
<tr>
<td>accuracy</td>
<td>71%</td>
<td>100%</td>
</tr>
<tr>
<td>similarity</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Abstract</strong></td>
<td>SA</td>
<td>CA</td>
</tr>
<tr>
<td>examples</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>matched</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>compilable</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>correct</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>coverage</td>
<td>100%</td>
<td>75%</td>
</tr>
<tr>
<td>accuracy</td>
<td>86%</td>
<td>50%</td>
</tr>
<tr>
<td>similarity</td>
<td>86%</td>
<td>95%</td>
</tr>
<tr>
<td><strong>Total coverage</strong></td>
<td>82%</td>
<td>(46/56)</td>
</tr>
<tr>
<td><strong>Total accuracy</strong></td>
<td>70%</td>
<td>(39/56)</td>
</tr>
<tr>
<td><strong>Total similarity</strong></td>
<td>96%</td>
<td>(46)</td>
</tr>
</tbody>
</table>

3.1.3 Results

To evaluate \textsc{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 (the similarity is computed using equation (1)), and experience at least one common edit between two program versions. We draw this test suite directly from systematic updates performed by programmers in \texttt{jEdit}, Eclipse \texttt{JDT core}, Eclipse \texttt{debug}, Eclipse \texttt{core.runtime} and Eclipse \texttt{compare} plug-ins. In the experiments, \textsc{Sydit} takes as input a source exemplar edit, which consists of an old and a new program fragment, and generates an edit script. The programmer selects the target. \textsc{Sydit} generates concrete edits and applies them to the target. We measure \textsc{Sydit}'s effectiveness in terms of coverage, accuracy, and similarity. We also observe how different configurations of \textsc{Sydit} can affect its effectiveness.

\[\text{similarity}(mA, mB) = \frac{|\text{matchingNodes}(mA, mB)|}{\text{size}(mA) + \text{size}(mB)}\]  

(1)

We manually inspect and categorize the 56 examples based on: (1) whether the edits involve changing a single AST node or multiple nodes, (2) whether the edits are contiguous or non-contiguous, (3) whether the edit’s content is identical or abstract. Here identical means that two methods’ edit content have the identical concrete representation; however, abstract means two methods’ edit content have different concrete representations but the same abstract representation when replacing all concrete identifiers with abstract ones. Different categories indicate different difficulty levels in automating systematic editing tasks.

For each method pair \((mA_o, mB_o)\) in the old version which are changed similarly to become \((mA_n, mB_n)\) in the new version, \textsc{Sydit} generates an abstract, context-aware edit script from \(mA_o\) and \(mA_n\), and tries to apply the learnt edits to the target method \(mB_o\), producing \(mB_n\).

In Table 1, \textit{matched} is the number of examples for which \textsc{Sydit} matches the learned context to the target method \(mB_o\). The \textit{compilable} row is the number of examples for which \textsc{Sydit} produces a syntactically-valid program, and \textit{correct} is the number of examples for which \textsc{Sydit} replicates edits that are semantically identical to what the programmer actually did. Coverage is \textit{matched \textit{examples}} / \textit{examples}, accuracy is \textit{correct \textit{examples}} / \textit{examples}, and similarity is computed using equation (1) between \textsc{Sydit}'s output and the expected output.

This table uses our best configuration of $k=1$, $Up=true$, $Down=false$, i.e., for each edit, in addition to
including the edited node and its parent node, SYDIT also includes nodes which the edited node is directly control or data dependent on as context. For this configuration, SYDIT matches the derived abstract 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.

3.2 Lase

Although SYDIT can perform many systematic edits correctly, it depends on programmers to specify target methods to change. We may extend SYDIT to automatically search for edit locations based on the edit script it derives from a single exemplar changed method. However, the derived edit script is not always well suited to finding edit locations because of two reasons. First, the mechanism of learning from a single example cannot reveal which changes in the example should be generalized to other places and which should not. As a result, it may over specify the script, making the extracted edit context too specific to the example, failing to match places which it should have matched. Second, the full identifier abstraction used to generalize an abstract edit script out of a concrete one may over generalize the script, allowing the extracted edit context to match places which it should not have matched because the places use totally different concrete identifiers.

To address SYDIT’s limitation in automatically finding edit locations, we design and implement another approach to generate a partially abstract, context-aware edit script from two or more exemplar changed methods, find other edit locations based on the same script, and apply a customized edit script to each identified location. This approach is implemented in a tool called LASE (Locating and Applying Systematic Edits). When given multiple exemplar changed methods, LASE infers an edit script for each example and extracts the edit operations (insert, delete, update, and move) common to all scripts in order to distinguish systematic changes from some specific to some of the examples. Then LASE computes the context of the inferred edit operations by extracting the relevant context in each example and finding the largest common context between them using a novel algorithm which combines clone detection [15], maximum common embedded subtree extraction [22], and dependence analysis. Within these two processes, LASE uses concrete identifiers if they are common in all examples, but uses abstract ones if they are different. In this way, LASE can avoid full identifier abstraction by abstracting identifiers only when necessary. Next, LASE creates a systematic edit script by recalculating the position of each common operation with respect to the common context. Finally, LASE uses the script to find edit locations and transform code.

We perform a thorough evaluation of LASE and its features on programmer applied systematic edits drawn from open-source programs. We use real-world repetitive bug fixes that required multiple check-ins in Eclipse JDT and SWT as an oracle. For these bugs, developers applied supplementary bug fixes because the initial patches were either incomplete or incorrect [29]. We evaluate LASE by learning edit scripts from the initial patches and determining if LASE correctly derives the subsequent, supplementary patches. On average, LASE identifies edit locations with 95% precision and 88% recall. The accuracy of applied edits is 91%—the tool-generated version is 91% similar to the developers’ version. In several confirmed cases, LASE identifies and performs edits on locations that the developers missed. We also evaluate LASE on a test suite of 37 systematic edits drawn from five Java open source projects. In these experiments, we find that the edit scripts inferred by LASE have significant fewer false positives and negatives than those inferred by SYDIT when they are used to find edit locations. Therefore, the partially abstract, context-aware edit scripts that LASE infers from multiple examples are critical to achieving high recall, precision, and accuracy in automatic program transformation.

3.2.1 Motivating Example

We use Figure 3 as a running example throughout this section. Consider the two methods mA and mB: 

\texttt{\texttt{textChanged(TEvent event)}} and \texttt{\texttt{updateActions()}}. These two methods perform similar tasks: (1) both iterate over all elements returned by \texttt{values()}, (2) both process elements one by one, (3) both cast each element to an object of certain type, and (4) when an element meets certain condition, both invoke the element’s
update() method. Besides, the two methods are changed in similar but not identical ways. The similarity is both methods add type check for the element iterated before casting it to an object of certain type (line 6-14 in mA and line 5-13 in mB). However, the difference is that in mA, two print statements outside the while loop are deleted (line 3-4 in mA), while in mB, one print statement inside the while loop is deleted, an extra type check and element processing are added (line 14-18 in mB).

Figure 3: Two exemplar changed methods to demonstrate a systematic edit for Lase

Suppose that programmer Pat already made the similar, but not identical changes to both methods. He wants to apply similar changes to other places in his project. Without any assistance, Pat must manually locate all other places to change and edit them one by one. This process is tedious and error-prone; because Pat may miss some locations to change or change some locations in an incorrect way. Using Lase, Pat instead provides the two already changed methods as examples. Lase derives a partially abstract, context-aware edit script from the changes in both mA and mB, identifies other edit locations based on the script, and transforms each location by applying the script.

3.2.2 Approach

This section overviews the six steps of Lase. The first four steps create a partially abstract, context-aware edit script from multiple changed methods $M$. The fifth step uses the edit-relevant context of the script to search for edit locations. The sixth step applies the edit script to each found location.

Step 1: Program Differencing

For each exemplar changed method $m_i \in M = \{m_1, m_2, \ldots, m_n\}$, where $n \geq 2$ is the number of exemplar methods, Lase compares the AST of $m_i$’s old and new versions and creates a sequence of node edit operations $\Delta_i$ using the program differencing algorithm mentioned in Step 1 (Program Differencing) of Section 3.1.2.

Step 2: Common Edit Operation Identification

Lase identifies common edit operations in $\{\Delta_1, \Delta_2, \ldots, \Delta_n\}$, by iteratively comparing the edits pairwise using a Longest Common Edit Operation Subsequence (LCEOS) algorithm, as shown in equation (2), which is a modified version of the Longest Common Subsequence (LCS) algorithm [13]. We do not require exact equivalence between edit operations because systematic edits are often similar, but not exactly the same. We define the comparison function $equivalent(e_p, e_q)$ for inexact matches between two edit operations in two ways: $concreteMatch(e_i, e_j, t_s)$ and $abstractMatch(e_i, e_j)$.

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

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

10
$LCEOS(s(\Delta_i, p), s(\Delta_j, q)) = \begin{cases} 
0 & \text{if } p = 0 \text{ or } q = 0 \\
LCEOS(s(\Delta_i, p - 1), s(\Delta_j, q - 1)) + 1 & \text{if equivalent}(e_p, e_q) \\
\max(LCEOS(s(\Delta_i, p), s(\Delta_j, q - 1)), LCEOS(s(\Delta_i, p - 1), s(\Delta_j, q))) & \text{if equivalent}(e_p, e_q) 
\end{cases}

(2)

$s(\Delta, i)$ represents the edit operation subsequence $e_1, ..., e_i$ in $\Delta$.

LASE first applies concreteMatch($e_i$, $e_j$, $t_s$), which takes as input two concrete edit operations, $e_i$ and $e_j$, and a threshold $t_s$. It compares $e_i$ and $e_j$ based on edit type and edited node $u$’s AST label, i.e., a string representation of the AST node. If both operations are of the same type and the labels’ bi-gram string similarity [2] is above the threshold $t_s$, the function returns true. By default, we set $t_s$ to 0.6 to increase the number of matches between different but similar edits. If two edit operations fail the concreteMatch test, LASE applies abstractMatch($e_i$, $e_j$), which first converts all identifiers of types, methods, and variables in the edited node $u$’s label to abstract identifiers $t\$, $m\$, and $v\$. If the edit types and their labels’ abstract representation match, abstractMatch returns true. The result is the set of concrete edits that are equivalent and common to all exemplar methods, but their identifiers and AST types may not match.

**Step 3: Identifier Abstraction in Common Edit Operations**

LASE abstracts identifiers as needed. When all the edits agree on an identifier, LASE uses the identifier as it is. If one or more edits use a different identifier, LASE abstracts the identifier. For example, Figure 3 shows $e_A =$ delete(MVAction action=(MVAction)e.next()) matches with $e_B =$ delete(MVAction action=(MVAction)iter.next()). When LASE detects the divergent identifiers $e$ vs. iter, it generalizes them by creating a fresh abstract variable identifier $v\$, substituting it for the original identifiers, and creating $e =$ delete(MVAction action=(MVAction)$v\$.next()). LASE records the pairs ($e$, $v\$), (iter, $v\$) in a map. It then substitutes $v\$ for all instances of $e$ in $m_A$ and $\Delta_A$ and all instances of iter in $m_B$ and $\Delta_B$ to enforce a consistent naming for all edit operations and edit context. The result is a list of partially abstract edit operations $\Delta$.

**Step 4: Common Edit Context Extraction**

LASE extracts the common edit context $C$ for $\Delta$ from the exemplar methods with a novel algorithm combining clone detection, maximum embedded subtree extraction, and program dependence analysis.

LASE first finds the largest common unchanged context shared among all methods by aligning each two methods’ unchanged context based on the common edit operations they share and using text-based clone detection [15] to find clones in each aligned code segment pair. This process reveals all possible common text shared between each two methods in sequence, $C_{text}$.

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

Because clone detection treats programs as structureless text, it may mismatch two AST nodes under unmatched parent nodes. To solve the problem, LASE enforces structural context matching by using an off-the-shelf Maximum Common Embedded Subtree Extraction (MCESE) algorithm [22] to extract the largest common forest structure shared between each two methods’ ASTs, as shown in equation (3). This algorithm traverses each tree in pre-order, indexes nodes, and encodes the tree structure into a node sequence. By computing the longest common subsequence between each two sequences and reconstructing trees from the subsequence, LASE find the largest common embedded subtree(s), $C_{sub}$.
MOVE v$0 as Iterator type or AST node type. For instance, because C with a method C.

Step 5: Edit Location Search

The difference is, for each edited node, LASE includes all nodes transitively control or data dependent on it as edit relevant context $C_{dep}$. $C = C_{sub} \cap C_{dep}$ is the final edit context extracted for $\Delta$. If $C_{dep}$ is empty, LASE simply uses $C_{sub}$ as $C$ in order to find edit locations. LASE combines $\Delta$ and $C$ to create a partially abstract, context-aware edit script, $\Delta_P$, where each edit operation in $\Delta$ is described with respect to context $C$. The resulting script is shown in Figure 4.

Step 5: Edit Location Search

Given $\Delta_P$, LASE searches for methods containing $\Delta_P$’s context $C$ and suggests them as edit locations. Because $C$ is partially abstract, it contains both concrete and abstract identifiers. When LASE matches $C$ with a method $m$, it matches concrete identifiers by name and matches abstract identifiers by identifier type or AST node type. For instance, $\text{Iterator}$ in $C$ only matches $\text{Iterator}$ in $m$. An abstract name, such as v$0$, matches any variable, while u$0:\text{FieldAccessOrMethodInvocation}$ matches FieldAccess or MethodInvocation AST nodes. LASE reuses the MCESE algorithm described in Step 4 to find the maximum common context between $C$ and $m$, but redefines the $\text{equivalent}(i,j)$ function to compare concrete identifiers based on string equality and abstract identifiers based on identifier type and AST node type equality. If the common context between $C$ and $m$ contains correspondence for each node in $C$, LASE suggests $m$ as an edit location $m_f$.

Step 6: Concrete Edits Generation and Application

For each suggested location $m_f$, LASE customizes $\Delta_P$ to create a concrete edit script and applies it to suggest a modified version, $m_s$, for developers to review. For this step, we slightly modify the edit customization and edit application algorithms in Step 5 (Concrete Edits Generation and Application) of Section 3.1.2. The customization algorithm replaces all abstract identifiers in $\Delta_P$ with corresponding concrete identifiers.
from \( m_f \) using the \textit{equivalent}(i, j) function defined above. The application algorithm does not use Eclipse \texttt{ASTRewrite} API to transform or generate code because the API can lead to high performance overhead. Instead, \texttt{LASE} applies the concrete edit script directly to \( m_f \) and then translates the resulting AST to code.

### 3.2.3 Results

We evaluate the precision and recall of \texttt{LASE} for finding the correct edit locations and the accuracy when \texttt{LASE} applies edits. We then explore the differences between edit scripts learnt from multiple examples and one example on a test suite of 37 systematic edits from other five Java open source programs.

To measure \texttt{LASE}'s precision, recall, and accuracy, we use data from Park et al.'s study on supplementary bug fixes [29]. They find a group of bugs that were fixed in multiple commits by performing clone detection among bug patches labeled with the same bug ID to find repetitive, similar bug fixes [31]. If a bug is fixed more than once and there are clones of at least two lines between its bug patches checked in at different times, we manually examine these methods for systematic changes. We find 2 systematic edits in Eclipse JDT and 22 systematic edits in Eclipse SWT.

We view the patches for each bug as an oracle of a systematic edit and test if \texttt{LASE} can produce the same results as developers when given the first two patches in the systematic bug fix set. Experiment shows that on average, \texttt{LASE} locates edit positions with respect to the oracle data set with 95% precision (the percentage of correctly identified locations out of all found locations), 88% recall (the percentage of correctly identified locations out of all expected locations), and performs edits with 91% accuracy (the similarity between tool-generated version and the expected version for any program). Since \texttt{LASE} produces the same results as developers did in many cases, we can conclude that \texttt{LASE} will help programmers detect edit locations earlier, reduce errors of omissions, and make systematic edits correctly.

To explore the difference between edit scripts learnt from multiple examples and one example, we use \texttt{LASE} to generate edit scripts from example pairs and \texttt{SYDIT} to generate edit scripts from single examples. We borrow the 56 exemplar edit test suite from \texttt{SYDIT}'s evaluation in Section 3.1.3, remove the simplest example pairs, getting 37 pairs of exemplar edits. Afterwards, we apply \texttt{LASE} to infer the systematic edit demonstrated by each exemplar pair and search for edit locations in the program's original version. If a suggested location by \texttt{LASE} is indeed edited similarly in the program's new version but not included in the test suite yet, we extend the test suite to include it as another example demonstrating the systematic edit. In this way, although \texttt{LASE} learns a systematic edit from only two examples, we compute its precision by comparing the correctly identified locations and all known locations in the extended test suite.

Experiment shows that on average, edit scripts learnt from one example has lower precision (77% vs. 94%) and lower recall (81% vs. 100%) when used to look for edit locations as compared to those learnt from two examples, but has higher accuracy (99% vs. 95%) when suggesting edits for correctly identified locations. Especially, in some cases, edit scripts learnt from one example have so poor precision (e.g., 0%) and recall (e.g., 25%) that they are not well suited for finding edit locations, and motivate using two or more examples.

### 4 Proposed Work

This section describes our plan to handle more sophisticated program transformations and transformation correctness checking. \texttt{SYDIT} and \texttt{LASE} focus on systematic edits composed of fine-grained changes contained within a single method, although in reality developers make many coarser-grained changes that span multiple methods and involve \textit{addition}, \textit{deletion}, \textit{update}, \textit{move} of types, methods, and fields. Extending our original approaches by including edit generation, edit location suggestion, and edit application for these changes can make our solution more complete. The planned extension will be covered in Section 4.1. Although it is difficult to guarantee correctness of the program transformation suggested by our approaches, we propose to provide a measure of confidence to developers in the edit scripts we apply. With the confidence provided, developers can decide which suggested changes they should pay more attention to before accepting the suggestions and which they should pay less given limited time and resource. The approach to automatically
check transformation correctness will be covered in Section 4.2. Finally, we conclude with a timetable for completing the proposed work in Section 4.3.

4.1 CoSydit: Generating Coordinated Systematic Edits from Examples

When programmers maintain their code, it is very often that they do not limit their changes to statements inside methods. Instead, they may insert, delete, update, and move the declarations of types (classes or interfaces), methods, and fields, so that all declaration changes together with statement changes make up a high-level, collaborative systematic edit script. Our proposed approach—CoSydit—aims to detect both declaration changes and statement changes already made by developers, understand the relations between them, search for edit locations where it can propagate part of the changes or supplement collaborative changes in order to complete the overall program transformation.

In addition to meeting the conflicting requirements of generalization from example(s) mentioned in Section 1, the challenge to inferring a high-level, collaborative systematic edit script from scattered changes made by developers also includes that we need to correlate the scattered changes with each other, infer the high-level, collaborative program transformation pattern out of the partial and exemplar changes, locate places missing collaborative changes, and suggest them together with concrete edits to developers.

In the rest of this section, we will first discuss our proposed solution together with some problems that will need further exploration, and then present an evaluation plan.

4.1.1 Our Proposed Solution

We plan to design and implement a tool called CoSydit (Collaborative Systematic Editing), to infer and complete coarser grained systematic editing tasks, which may consist of both statement changes inside methods and declaration changes of types (classes or interfaces), methods, and fields. There will be basically five phases in the approach.

Phase I: Atomic Change Identification   Given two versions of a program, we identify all changes between them with existing program differencing tools, such as Eclipse compare plugin and ChangeDistiller [9]. The changes are categorized as declaration changes and statement changes. Declaration changes include: inserted types (IT), deleted types (DT), updated types (UT), moved types (MT), inserted methods (IM), deleted methods (DM), updated methods (UM), moved methods (MM), inserted fields (IF), deleted fields (DF), updated fields (UF), and moved fields (MF). Statement changes include: inserted statement (IS), deleted statement (DS), updated statement (US), and moved statement (MS). Declaration changes are inferred from the results provided by structural program differencing tools, such as Eclipse compare plugin; while statement changes are inferred with syntactic program differencing tools, such as ChangeDistiller, as mentioned in Section 3.1.2.

Note that although a class may be modified because two fields are inserted to it, we do not create a UT (Updated Type) change due to the two IF (Inserted Field) changes for it, because UT (Updated Type) is an atomic change independent of any other change. We only create a UT (Updated Type) when a type’s modifier, name, super type’s name, or interface name list is changed. When parsing declaration changes, we do not assume they are ordered. However, we assume statement changes identified within the same method are ordered, because different orders to apply the same statement changes can sometimes lead to different results.

Phase II: Change Relation Identification In order to correlate scattered changes with each other to create a high-level view about the program transformation, we need to do some analysis on relations between changes.

There may be many types of relations, two of which are dependence and coexistence. Dependence relation means the application of one change depends on that of another change due to compilation reasons or programming guidelines. For instance, a US (Updated Statement) change depends on an IM (Inserted Method) change if the statement after update invokes the newly defined method. An IM (Inserted Method)
change depends on an IT (Inserted Type) change if the method is to be inserted to the newly inserted class. Dependence relations can be inferred between declaration changes or between a declaration change and one or more statement changes. Coexistence relation means the application of two changes usually happen together. It is weaker than dependence relation. For instance, the content of `hashCode()` and that of `equals(Object)` is always changed together. In order to represent method content changes as a whole, we append a new type of changes to the changes mentioned above—content changed methods (CCM). Different from other types, CCM is not atomic and is composed of a list of statement changes inside methods. If a statement change, such as US (Updated Statement), inside a CCM depends on a declaration change, such as IM (Inserted Method), we will establish dependence relation between US (Updated Statement) and IM (Inserted Method), and between CCM and IM (Inserted Method). There may be other types of relations between these types of changes, and we will explore them. The output of this phase is groups of detected correlated changes.

**Phase III: Program Transformation Identification**  
Given the partial and exemplar changes made by developers, we need to identify the high-level program transformation developers intend to do based on the limited information. We may need to establish a program transformation pattern library, which maintains a set of high-level program transformation descriptions together with their corresponding correlated change patterns. For instance, one program transformation “method extraction” corresponds to one IM (Inserted Method), one or more IS (Inserted Statements), 0 or more DS (Deleted Statements), meaning that it creates a new method out of some existing code fragment and replaces each such code fragment with the corresponding method call.

Once we have correlated changes detected between a program’s two versions and a program transformation pattern library, we can map the detected changes against the correlated change pattern for each transformation. If all detected changes match a transformation’s change pattern, we can infer the transformation as the one which developers intend to fulfill. If there are two or more transformations inferred out of the detected changes, we may rank them based on some heuristics and suggest the first one to developers or list all of them for developers to decide. No matter which way we take, the output of this phase is the inferred program transformations for each group of detected correlated changes.

**Phase IV: Edit Locations Search**  
By comparing the change pattern of an inferred program transformation and the changes already done by developers, we can figure out the difference between them, and concretize the difference to edit locations missing changes. The difficulty of finding edit locations may depend on which program transformation we want to complete and what changes to supplement. For instance, if we perform a “method extraction” transformation, a piece of code has been extracted to define a new method, and some of its occurrence has been replaced by method calls; what we need to suggest is just the places where the code snippet has not been replaced by the method call. However, if we perform a “super class extraction” transformation, a new class is declared, and several classes are changed to extend it; it is hard to suggest what other classes should be changed in the same way because potentially every unchanged class can be a candidate. We will explore examples used in practice to develop more insight and to derive a solution.

**Phase V: Edit Suggestion**  
When we suggest locations missing changes, we usually know for sure which changes are missing there. However, the thing we are not sure is how to create customized edit suggestion for each edit location based on the partial information at hand. Exemplar code changes can only tell what happens in other places while the inferred change pattern can only provide a general description of changes. In order to suggest an applicable edit to an edit location, we may need to extract more information about the change context, and explore how to convert an exemplar code change or customize a general change pattern based on that.
4.1.2 Proposed Evaluation Plan

To evaluate the approach, we will first construct a test suite consisting of collaborative systematic edits found in different versions of existing open source projects. By intentionally removing some of the changes, we can leave the rest changes for CoSYDIT to detect and learn from. And we will measure CoSYDIT’s performance based on the following criteria:

- In how many cases are detected code changes correlated as expected, compared to a by-hand analysis?
- In the cases where correlated changes are correctly identified, in how many cases is a program transformation correctly inferred out of the changes?
- In the cases where program transformations are correctly inferred, what is the precision and recall of edit location suggestion?
- In the cases where program transformations are correctly inferred, for all correctly suggested edit locations, what is the average similarity between tool-generated version and developer-created version?

We will also evaluate the approach by comparing any two versions of an open source project to infer collaborative edits and suggest some changes. By checking whether the changes actually occur later in the project’s version history or are validated by developers, we can measure CoSYDIT’s performance based on the criteria mentioned above. What is more important, we can show the potential assistance it can provide to developers in software maintenance.

4.2 Transformation Correctness Checking

Although we have tried various ways to infer the most appropriate edit script from example(s), we cannot guarantee the correctness of generated edit scripts because programs are complex and some of their behaviors are statically undecidable. However, we propose to explore if it is possible to provide confidence to developers in the edit scripts generated; so that developers are always warned to carefully check the scripts in which we have less confidence.

One possible approach is after customizing a systematic edit script ∆ for a code place and applying it to the place, we can regenerate a systematic edit script ∆′ out of the code place’s old and suggested versions. The approach is similar to Son et. al’s work [32]. By comparing how similar the two edit scripts are to each other, we can compute a confidence value for developers.

In order to evaluate whether the confidence value reflects the reality, we plan to construct a test suite of systematic edits. For each systematic edit, by learning a systematic edit from one or some examples and applying it to the remaining examples, we can compute a confidence value $C_i$ for each target and get an average value $\bar{C}$ between them. We will also manually check the correctness of each edit script application, set the real confidence value $RC_i$ to 0 if the application is wrong or 1 if correct, and calculate an average value $\bar{RC}$. By comparing corresponding $C$ values and $RC$ values, we can know how good our confidence values are.

This piece of work is not as challenging as the prior one mentioned above, but it will be interesting to explore some possible approaches to predict correctness of edit scripts generated. If we succeed in finding such an approach, we can save developers a lot of edit script checking time. However, if we fail to do that, we may still gain insight into correctness checking and how to generate correct edit scripts.

4.3 Timetable

This section summarizes the timeline to complete the work which is outlined in this section:

1. Design and implement CoSYDIT (11 months)
   (a) Atomic change identification (1 month)
Change relation identification (2 months)
Program transformation identification (2 months)
Edit location search (2 months)
Edit suggestion (2 months)
Evaluation (2 months)

2. Design and implement transformation correctness checking (3 months)

3. Wrap up and dissertation writing (3 months)

5 Summary

This proposal has presented the opportunity and challenge to automate systematic program transformation. By taking advantage of the opportunity and trying different ways to deal with the challenge, we explore three approaches to infer systematic edits from examples and one approach to check correctness of systematic edits:

1. We designed and implemented Sydit to infer a systematic edit from a single exemplar changed method and apply it to user-selected target methods.

2. We designed and implemented Lase to infer a systematic edit from multiple exemplar changed methods, so that the edit script can be used to locate target methods to change and furthermore change them.

3. We propose CoSydit to infer a systematic collaborative edit script from multiple exemplar heterogeneous changes to different entities (types, methods, and fields), find edit locations, and suggest edits.

4. We propose an approach to automatically check the correctness of any inferred edit script in order to reduce developers’ burden in edit suggestion review.

We show that systematic program transformation can be automated based on the change examples demonstrated by developers. It will significantly save developers time and efforts in constructing repetitive code changes from scratch and identifying all places in need of changes. It may furthermore relieve developers from change suggestion checking by providing automatic checking support.

References


