Copyright
by
Na Meng
2014

The Dissertation Committee for Na Meng
certifies that this is the approved version of the following dissertation:

Automating Program Transformations based on Examples of Systematic Edits

Committee:

Miryung Kim, Supervisor

Kathryn S. McKinley, Co-Supervisor

Don Batory

Dewayne Perry

Vitaly Shmatikov

Automating Program Transformations based on Examples of Systematic Edits

by

Na Meng, B.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

Dedicated to my father, mother, and Yu.

Acknowledgments

I would like to express my sincere gratitude to Kathryn McKinley and Miryung Kim for supporting and mentoring me. I will never forget the first meeting I had with Kathryn to cautiously ask whether she would like to take me as her student. Although she almost knew nothing about me except that my English is poor, she said "Yes" without much thinking and immediately asked me whether I had any financial support because she would like to provide me a Research Assistantship. I will also never forget the first time Miryung asked me whether I would like to get involved in her project during the beginning days of my PhD when I had a hard time to figure out what research to do for the following five years. Both advisors entered my life when I felt at loss and badly needed help. Ever since then they have provided me a lot of important guidance and enlightenment, which will affect my entire life. It is my great honor to work with two advisors who are diverse in their personality and expertise. Kathryn's vitality and optimistism always encourage me to aggressively push myself to limits and proactively face challenges head on, while Miryung's rigorousness and precision always remind me to fight hard for every single goal I want to achieve and pay attention to details in the process. Kathryn has expanded my horizon on research spanning compilers, memory management, programming languages, architecture and machine learning, while Miryung has led me delve into research on software evolution. Despite the diversity, both advisors exhaust their ability to cultivate my interest in research, inspire me to think creatively, share their expertise with me, polish up my writing, and compliment me a lot. I entered UT graduate school unconfident about

getting a Ph.D., and I leave ambitious to pursue an academic career because of them.

Don Batory, Perry Dewayne, and Vitaly Shmatikov have given helpful feedback and spent a lot of time reading long documents and attending long talks. Ira Baxter has provided valuable suggestion.

Jungwoo Ha and Todd Mytkowicz have been enthusiastic supporters, brainstorming novel research and sharing their research skills. Don Batory, Dewayne Perry, and Vitaly Shmatikov have given helpful feedback and spent a lot of time reading long documents and attending long talks. William Cook has been very supportive to my projects and has provided valuable advice and mentoring.

The student community at UT has been incredibly supportive. I am really grateful to have John Jacobellis and Lisa Hua as friends and colleagues. I am indebted to them and the following friends and colleagues for their technical and personal support: Byeongchol Lee, Dong Li, Alex Loh, Sooel Son, Baishakhi Ray, Tianyi Zhang, Myoungkyu Song, Yamini Gotimukul, Ray Qiu, Aibo Tian, Xin Sui, Yang Wang, Chen Qian, Wei Dong, Song Han, Xu Wang, Jian Chen, Xiaohu Shen, Xiuming Zhu, Hongyu Zhu, Dimitris Prountzos, John Thywissen, Katie Genter, Patrick MacAlpine, Ivan Jibaja, Katherine Coons, Bert Maher, Michael Bond, Suriya Subramaniam, Jennifer Sartor and Maria Jump. They have given me both technical and personal support.

I am thankful for guidance and help provided by Phyllis Bellon, Lindy Aleshire, Lydia Griffith, and Gem Naviar.

I am really thankful to my parents for all unconditional love and selfless support they have provided throughout my life.

My husband Yu has provided quite important support for my study and daily life while also being awesome and fun, and I'm grateful.

Automating Program Transformations based on Examples of Systematic Edits

Publication No. _____

Na Meng, Ph.D.
The University of Texas at Austin, 2014

Supervisors: Miryung Kim
Kathryn S. McKinley

Programmers make *systematic edits*—similar, but not identical changes to multiple places during software development and maintenance in order to add features and fix bugs. Finding all the correct locations and making the edits correctly is a tedious and error-prone process. Existing tools for automating systematic edits are limited because they do not create general purpose edit scripts or suggest edit locations, except for specialized or trivial edits. Since many similar changes occur in similar *contexts* (in code with similar surrounding dependent relations and syntactic structures), there is an opportunity to automate program transformations based on examples of systematic edits. By inferring systematic edits and relevant context from one or more exemplar changes, automated approaches can (1) apply similar changes to other locations, (2) locate code that requires similar changes, and (3) refactor code which undergoes systematic edits. This thesis seeks to improve programmer productivity and software correctness by automating parts of systematic editing and refactoring.

Applying similar, but not identical code changes, to multiple locations with similar contexts requires (1) understanding and relating common program context—a program's syntactic structure, control, and data flow—relevant to the edits in order to propagate code changes from one location to others, and (2) recognizing differences between locations in order to customize code changes for each location. Prior approaches for propagating nontrivial, general-purpose code changes from one location to another either do not observe the program context when placing edits, or do not handle the differences between locations when customizing edits, producing syntactic invalid or incorrectly modified programs. We design a novel technique and implement it in a tool called SYDIT. Our approach first creates an *abstract, context-aware* edit script which contains a syntax subtree enclosing the exemplar edit with all concrete identifiers abstracted and a sequence of edit operations. It then applies the edit script to user-selected locations by establishing both context matching and identifier matching to correctly place and customize the edit.

Although SYDIT is effective in helping developers correctly apply edits to multiple locations, programmers are still on their own to identify all the appropriate locations. When developers omit some of the locations, the edit script inferred from a single code location is not always well suited to help them find the locations. One approach to infer the edit script is encoding the concrete context. However, the resulting edit script is too specific to the source location, and therefore can only identify locations which contain syntax trees identical to the source location (false negatives). Another approach is to encode context with all identifiers abstracted, but the resulting edit script may match too many locations (false positives). To suggest edit locations, we use *multiple examples* to create a *partially abstract, context-aware* edit

script, and use this edit script to both find edit locations and transform the code. Our experiments show that edit scripts from multiple examples have high precision and recall in finding edit locations and high accuracy when applying systematic edits because the extracted common context together with identified common concrete identifiers from multiple examples improves the location search without sacrificing edit application accuracy.

For systematic edits which insert or update duplicated code, our systematic editing approaches may encourage developers in the bad practice of creating or evolving duplicated code. We investigate and evaluate an approach that automatically refactors cloned code based on the extent of systematic edits by factoring out common code and parameterizing any differences between them. Our investigation finds that refactoring systematically edited code is not always feasible or desirable. When refactoring is desirable, systematic edits offer a better way to scope the refactoring as compared to whole method refactoring. Automatic clone removal refactoring cannot obviate the need for systematic editing. Developers need tool support for both automatic refactoring and systematic editing.

Based on the systematic changes already made by developers for a subset of change locations, our automated approaches facilitate propagating general purpose systematic changes across large programs, identifying locations requiring systematic changes missed by developers, and refactoring code undergoing systematic edits to reduce code duplication and future repetitive code changes. The combination of these techniques opens a new way of helping developers automate tedious and error-prone tasks, when they add features, fix bugs, and maintain software. These techniques also have the potential to guide automated software development and maintenance activities based

on existing code changes mined from version histories for bug fixes, feature additions, refactoring, and software migration.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developers spend a lot of time and effort changing code when they fix bugs, add features, adapt the software to new environments, and refactor code to improve performance, correctness, and maintainability [83]. Even with all this effort, software still has a lot of errors due in part to increasing software complexity and time-to-market pressure during a development cycle. These errors cost the U.S. economy an estimated 60 billion dollars every year [90]. The magnitude of these costs shows there is a need for tools and techniques which improve software correctness and enhance programmer productivity during software development and maintenance.

Recent work observes that many changes are *systematic*—programmers add, delete, and modify code at different code locations in similar, but not identical ways [50, 75]. For example, Kim et al. find that on average, 75% of structural changes to mature software are systematic [50]. They find that these changes are not identical, but that their contexts have similar characteristics. For instance, these contexts include similar methods and accesses to the same field(s). Nguyen et al. find that 17% to 45% of bug fixes are recurring fixes that involve similar edits to numerous methods [75]. Finding all the correct locations and making these edits correctly is a tedious and error-prone process.

We call similar but not necessarily identical code changes to multiple code locations *systematic edits*. We call related code for an edit *context*.

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. Refactoring tools can automate a predefined set of semantics preserving transformations [25, 31, 78, 91, 96]. However, they do not provide an easy way for developers to define new program transformations, neither do they implement semantics modifying transformations. With source transformation languages and tools [14, 16, 35, 58, 95], users define program transformations by describing code patterns to match and edit actions in domain-specific languages. These tools require developers to plan edit operations in advance and manually encode everything from scratch. Simultaneous editing tools [72, 92] proactively replicate *identical lexical* edit actions, such as *move the cursor to the end of a line*, in one text fragment to another preselected or identified text fragment simultaneously. These tools put strict requirements on the way developers may edit text. Programming by demonstration tools either record a human expert's edit actions on several examples [59, 60] or look at several text change examples [36, 77] to automate repetitive text editing tasks. However, these tools process only plain text and do not observe syntactic structures in programs.

When making similar changes to multiple locations, we believe that fully-automated tool support for creating, locating, and applying general purpose systematic edits will help developers create more correct code and produce it faster. None of these tools have been used or evaluated on a wide range of real-world nontrivial examples. Furthermore, none of these techniques recommends code duplication removal refactoring, when witnessing repetitive

changes to multiple locations in order to reduce the need for systematic edits to multiple locations in the future.

1.1 Thesis Statement

This thesis seeks to exploit the observation that programmers often make similar code changes to program locations which contain similar but not identical contexts, such as similar program syntactic structures with differently used variables. It investigates how to automate program transformation based on examples of systematic edits. Our hypothesis is as follows:

> *By inferring systematic edits and relevant context from user-provided exemplar changes, we can (1) create accurate general purpose edit scripts that apply to multiple locations, (2) use the inferred edit script together with relevant context to find edit locations missed by developers, and (3) refactor code fragments based on the systematic edits they experience.*

Our goal is to provide automated program transformation tool support for similar code scattered across software systems and to help developers identify, locate, and apply similar changes consistently that fix bugs, add features, migrate software, and refactor code. We expect developers to examine and test the resulting code to ensure its correctness. We believe that automated tool support will improve programmer productivity, enhance software quality, and reduce software maintenance costs.

To verify our hypothesis, we investigate the following three questions:

- Can we create a generalized edit script from a single code change example and correctly apply it to other code fragments?
- Can we use the edit script from one example or do we need to infer an edit script from multiple examples to correctly locate places for systematic edits?
- Can we apply clone removal refactoring to eliminate the need of systematic edits?

To investigate the first question, we design and implement an approach that derives an *abstract, context-aware* edit script from *one* changed method (or function) and applies the script to user-selected target methods (Chapter 4). Each edit script consists of context, a code pattern represented as an abstract syntax subtree, and a sequence of edit operations. The code pattern abstracts away all concrete identifiers of used types, methods, and variables. The edit operations represent simple and complex code changes which add, delete, update, or move statements in single method bodies. When users select code locations to apply an edit script, the approach establishes matching between the edit context and each location, to customize the general program transformation for each location, and applies the result accordingly.

To investigate the second question, we use the edit script derived from a single method to find edit locations without user assistance. Unfortunately, such edit scripts are inadequate for identifying edit locations. They either report too many incorrect locations or miss many correct locations, because the contexts encoded in edit scripts from single examples are either too restrictive for structure matching or too relaxed for identifier matching. A restrictive context only recognizes edit locations whose program structures are very similar

to the original code or incorrectly filters out locations which are less structurally similar. Relaxed identifier matching mechanisms reduce the capability of differentiating and precisely matching identifiers used in edit context, and thus incorrectly recommends edit locations. To reduce false positives and false negatives when looking for edit locations with edit scripts, we design and implement another approach that derives a *partially abstract, context-aware* edit script from *multiple* exemplar changed methods (Chapter 5). The approach infers the *most specific generalization* of systematic edits by specifying common identifiers and code, and then abstracting variations between examples. It extracts common edits as well as relevant context and commonly used identifiers. The resulting systematic edit is effective in finding edit locations because the common context better characterizes edit locations and provides good indication of edit applicability, while abstracted identifiers and locations make the edit script flexible.

To investigate the third question, we design and implement a clone removal refactoring approach which leverages systematic edits to extract common code and parameterize any difference between similarly changed methods (Chapter 6). The refactoring extracts common code guided by a systematic edit; creates new types and methods as needed; parameterizes differences in types, methods, variables, and expressions; and inserts return objects and exit labels based on control and data flow. The evaluation with real-world systematic edits shows that scoping refactoring based on systematic edits significantly improves the applicability of automate clone removal, compared with method based refactoring. However, we cannot refactor many systematic edits due to reasons like language limitations and semantic constraints. Our version history examination shows that developers do not always refactor systematic

edits. Both observations show that automated refactoring cannot obviate the need for systematic editing.

In the next three sections, we overview related work and elaborate on our research questions, approaches, and results in more detail.

1.1.1 Automating Systematic Edits

To correctly apply similar edits to multiple locations, developers must understand the program context, which includes the syntactic program structure, and control and data dependence relations between edited code and unchanged code. Developers should be cautious about any difference between locations, because they may affect customization and application of the general program transformation.

Existing work either requires developers to manually prescribe systematic edits to apply [14, 16, 35, 58, 95] or provides limited support for general purpose systematic edit application [45, 82, 88, 99]. Some techniques simply propagate identical code changes in the source location to each target location without adaptation [72, 92].

Since developers write or change at least one method before editing other methods similarly, providing a code change example and locating other places for change is a natural interface. The challenge is that both the edits and relevant contexts are similar but different, meaning that the inferred systematic edit must tolerate differences between locations. Our hypothesis is that by extracting the edit relevant context using program containment, control and data dependence analysis, and abstracting identifiers of used types, methods, and variables, we can generalize a systematic edit applicable to different contexts. To verify the hypothesis, we investigate the following questions:

- How much context should we extract and which identifiers should we abstract to generalize a systematic edit from an exemplar edit?
- How can we customize and apply a general systematic edit to a concrete edit location?
- How similar are tool-generated versions to developer-created versions?

We design an approach which generates an *abstract, context-aware* edit script from an exemplar changed method (Chapter 4). It then customizes the edit script and applies the result to each user-selected target method. The approach is implemented in a tool called SYDIT.

Given an exemplar changed method, SYDIT performs syntactic program differencing [26] to infer the edit applied. It then uses program containment, control, and data dependence analysis to capture code relevant to the edit, regarding it as *edit relevant context*. SYDIT abstracts all identifiers used in both edit operations and relevant context to create an abstract, context-aware edit script. Next, it uses the relevant context as a code pattern to search for matches in target locations specified by developers and then customizes code changes for each location. The context refinement abstracts away code irrelevant to a systematic edit and thus makes the edit applicable to locations containing relevant context together with arbitrary irrelevant context. The identifier abstraction makes the edit applicable to locations using different identifiers.

Given a new target location specified by a user, SYDIT establishes context matching between the edit script and target location by finding a subtree in the target method which matches the extracted subtree. If there is a

match, SYDIT furthermore establishes an abstract-to-concrete identifier mapping. The established mappings facilitate SYDIT's customization of a systematic edit for a target location by repositioning each edit operation with respect to the matching subtree in the target and replacing every abstract identifier with corresponding concrete identifiers.

With a test suite consisting of 56 systematic editing tasks in five open source projects, we demonstrate that SYDIT produces syntactically valid transformations for 82% of these tasks. It perfectly mimics developer edits for 70%. Syntactic program differencing considers the human-generated version and the SYDIT-generated version 96% similar, indicating that developers need modest manual effort to correct SYDIT's version in most cases. When applying non-trivial systematic edits, which require modifying non-contiguous statements inside methods, SYDIT correctly handles 15 out of 23 cases. By exploring different ways to extract edit context, we find that including one direct containment, and one direct upstream control and data dependence for each edited statement leads to the most accurate code. By exploring different ways to abstract identifiers, we find that abstracting all identifiers for types, methods, and variables makes the inferred edit script most flexible and increases its applicability.

We show that SYDIT is able to infer a general program transformation from one example and correctly customize and apply the transformation to other code fragments. However, SYDIT requires developers to manually specify target locations. Our later experiments in Chapter 5 show that the inferred systematic edit is not suitable to search edit locations because it either wrongly matches too many locations or misses too many correct locations.

1.1.2 Finding Systematic Edit Locations

Prior work [66, 97] searches code based on matching patterns prescribed by developers but requires a lot of manual effort for pattern prescription. Some other techniques [63, 75] keep track of clone groups and their evolution history to recommend similar changes among clones. Although it can be very complex to maintain and propagate clone relevant information in large programs, the feature of code search is still limited to clones within one group.

We require developers to make some edits, in this case to at least two locations. We leverage these changes to infer the relevant context used for both location search and edit application. We do not want to use as many exemplar edits as possible. The more examples we require, the more manual edit burden is put on developers and the fewer potential edit locations are left for a tool to find out. We investigate using one exemplar edit by exploiting the edit-relevant context—code matching pattern—inferred by SYDIT. We find that the context is not well suited to find edit locations for two reasons. First, uniformly extracting context based on the program dependence relationship may include some relevant statements which are so specific to the given exemplar edit that they do not generalize to other edit locations. Second, uniformly abstracting identifiers of used types, methods, and variables does not allow the abstract context to specially characterize locations using certain identifiers, such as all locations accessing the field `fContainer` or invoking the method `isValid()`. Given one exemplar edit, SYDIT has a limited choice of ways to generalize a program transformation: it either generalizes everything, generalizes nothing, or generalizes something based on some metrics. No matter which choice, SYDIT has no insight about which program statement or identifier is shared among all edit locations or not and should get

generalized or not. Consequently, it can only generate an abstract edit script which may match one or some edit locations but not all of them. This limitation indicates the potential if developers can provide more than one example, the tool better infers and generalizes edit context for edit location search. To verify our hypothesis, we investigate the following questions:

- How can we abstract the edit-relevant context from multiple systematically edited examples?
- How can we use the abstract context to search for edit locations?
- What is the capability of the inferred edit script to find correct edit locations and correctly apply edits?

We design an approach which creates a *partially abstract, context-aware* edit script from two or more exemplar changed methods, and uses the script to automatically search for and identify edit locations, and then transforms the code (Chapter 5). The approach is implemented as a tool called Lase.

Given two or more exemplar changed methods, Lase performs syntactic program differencing to create an AST edit script for each method, and then combines them to infer the *most specific generalization* of the edit scripts. Lase extracts common AST edit operations, from all edit scripts based on the type and content of each edit operation. It omits edit operations which differ between examples and abstracts *only* discrepant identifiers and expressions. For instance, if two examples delete an `if` statement, but disagree on certain types, methods, variables, or expressions in the condition, Lase abstracts the discrepancy while keeping the ones that match as concrete identifiers, creating a partially abstract edit operation. For each method, similar to Sydit,

LASE uses control and data dependence analysis to capture the AST *change context* of extracted common edit operations; it also extracts *edit relevant context*, a subtree including both change context and changed statements. LASE then identifies common edit relevant context among methods with clone detection [47] and common embedded subtree extraction [65]. The identified common context together with the common edit constructs a partially abstract, context-aware edit script. LASE then uses the abstract context to find edit locations. It searches for methods within the whole program, which have a subtree that matches the context and contains all concrete identifiers encoded in the context. For each method containing the identifiers and matching the context, LASE customizes the edit script and applies the result.

To evaluate our approach, we explore using the edit scripts inferred by SYDIT from single exemplar edits to search for edit locations and using scripts from LASE that leverage two or more code change examples. We draw 37 systematic edits from SYDIT's test suite, and run LASE's matching algorithm with edit scripts learnt from either one or two examples. We compare their precision and recall when finding edit locations. The result shows that edits from two examples have significantly higher precision (94% vs. 74%) and recall (100% vs. 82%) than from one example, showing that the strategy of learning program transformation from multiple examples is crucial in automatically suggesting edit locations. We further evaluate LASE with 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 [80]. We evaluate LASE by learning edit scripts from the initial patches and determining if LASE correctly derives the subsequent, supplementary patches. On average, LASE identifies

edit locations with 99% precision and 89% recall. The accuracy of applied edits is 91%, i.e., the tool-generated version is 91% similar to the developer's version. Furthermore, LASE identifies and correctly edits 9 locations in the oracle test suite which developers confirmed that they missed, showing that LASE can detect and correct errors of omissions.

With these experiments, we show that LASE infers a program transformation from multiple examples, uses the resulting edit scripts to find edit locations with high precision and recall, and makes edits with high accuracy. The extracted common context and identified common concrete identifiers improve location search without sacrificing much edit application accuracy. Although LASE requires developers to provide at least one more example than SYDIT to better characterize the edit-relevant context, we believe that task is easier than finding every edit location by hand and thus is worth the developers' effort.

1.1.3 Refactoring Systematically Edited Code

SYDIT and LASE provide tool support for creating, locating, and applying systematic edits to multiple locations. However, they may encourage developers in the bad practice of creating or maintaining duplicated code. Similar edits to multiple locations may indicate that developers should instead refactor code to eliminate redundancy and future systematic edits. We explore this question by designing a fully automated refactoring tool called RASE, which performs *clone removal*. RASE (1) extracts common code guided by a systematic edit; (2) creates new types and methods as needed; (3) parameterizes differences in types, methods, variables, and expressions; and (4) inserts return objects and exit labels based on control and data flow. Existing

clone removal refactoring tools [7, 41, 46, 57, 89] only implement some, but not all of the refactoring techniques in Rase. To our knowledge, the functionality makes Rase the most advanced automated clone removal refactoring tool.

To explore whether systematic edits are obviated by using automated refactoring, we refactor code based on systematic edits. Our hypothesis is similar edits applied to similar code are good indications for refactoring. To verify our hypothesis, we investigate the following questions:

- Can we better scope clone removal refactoring based on systematic edits than method based refactoring?
- How should we refactor to extract commonality and parameterize differences?
- Is it more desirable to refactor or to perform systematic edits?

We design and implement an automatic refactoring approach which combines *extract method* (pg. 110 in [28]), *add parameter* (pg. 275 in [28]), *introduce exit label*, *parameterize type*, *form template method* (pg. 345 in [28]), and *introduce return object* refactorings to extract and remove similar code (Chapter 6). The approach is implemented as a tool called Rase.

Rase takes as input two or more methods with systematic edits to scope its refactoring region. It extracts the maximum common AST subtree encompassing all edited statements in each method and creates a general representation by abstracting any differences in used types, methods, variables, and expressions. The general representation is called an *abstract refactoring template*. Based on the template, Rase creates an extracted method, generates new types and methods for abstract type and method identifiers, declares

extra method parameters for abstract variables and expressions, defines return objects if more than one variable is returned by the extracted method, and inserts exit labels if the extracted method contains non-local jumps such as `break` and `return`. Finally, it replaces the extracted code in each original location with customized invocations of the extracted method.

We evaluate Rase with real-world systematic edits and compare to method based clone removal. Rase successfully performs clone removal in 30 of 56 cases for systematic edits from method pairs (n=2) and 20 of 30 cases from method groups (n≥3). We find that scoping refactoring based on systematic edits (58%), rather than the entire method (33%), significantly improves the applicability of automated clone removal. Lack of automated refactoring feasibility in the other 42% cases and lack of manual refactoring in later versions after systematic edits indicate that automated refactoring does not obviate the need for systematic editing. We believe these results offer strong evidence that both systematic editing and clone removal refactoring are necessary for software evolution.

1.2 Contributions

In summary, this thesis makes the following contributions:

- Sydit is the first tool to apply nontrivial general-purpose abstract edits to similar user-specified code fragments given one code change example. It significantly improves the capabilities of the state-of-the-practice tools such as a line-based GNU patch or the search and replace feature in text editors. Our evaluation shows that Sydit matches edit context and applies an edit on 82% of cases in our test suite, makes correct edits

for 70%, and produces code on average 96% similar to developer-created version.

- SYDIT improves programmer productivity and software correctness by consistently propagating similar changes automatically.

- LASE is the first tool to learn a general-purpose partially abstract edit from multiple changed methods, to use the edit to find edit locations, and to perform customized program transformation at each found location. Our results show that LASE finds edit locations with 99% precision and 89% recall, and transforms code with 91% accuracy.

- LASE improves programmer productivity and software correctness by identifying edit locations missed by developers and changing multiple locations consistently.

- RASE is the first clone removal refactoring tool that takes methods with systematic edits as input and exploits systematic edits to improve method refactoring. It is the start-of-the-art in terms of its capability to factorize and generalize. In our evaluation, it automates refactoring for 30 out of 56 cases of systematically edited method pairs and 20 out of 30 cases of method groups.

- RASE improves programmer productivity and software correctness by leveraging systematic edits to automatically scope a refactoring region and creates an executable plan of commonality extraction and variation parameterization.

1.3 Impact

This dissertation seeks to influence tool developers and programmers. For tool developers, our work will guide them to build automated program transformation tools by propagating bug fixes, feature additions, adaptive changes, and refactorings in one or some code locations to other locations which contain similar contexts—similar program syntactic structures with similar definition and/or use patterns of variables—to ease software development and maintenance activities. For programmers, our work provides alternative automatic support to help them identify, understand, modify, and/or refactor multiple code fragments in need of similar changes, and facilitate consistent and efficient programming practice,s both of which can lead to higher quality software.

Chapter 2

Definitions

- An **AST edit operation** describes a tree edit operation manipulating a program's AST. We focus on four types of AST edit operations: statement insertion, statement deletion, statement update, and statement move. Each edit operation records the edit type, edit content—text of the AST node it manipulates, and edit position in an AST.
- An **AST edit script** consists of a sequence of AST edit operations. Every operation takes an AST as input, edits the tree, and outputs a new AST, and thus each succeeding edit operation works on the resulting tree output by its predecessor.
- **Context** means code in a method.
- **Edit relevant context** is a code snippet extracted from an edited method. The code snippet corresponds to a syntactically valid AST. Given an edit, edit relevant context includes both edited statements and unchanged statements which either have control or data dependence relationship with the edited ones.
- **Abstract edit relevant context** is a code template generated from edit relevant context with all identifiers of types, methods, and variables abstracted as opposed to using the actual concrete identifiers.

- **Partially abstract edit relevant context** is a code template generated from edit relevant context with some identifiers of types, methods, variables, and expressions abstracted.

- An **abstract, context-aware edit script** is derived from one changed method. The script consists of a sequence of AST edit operations and the relevant context. All concrete identifiers used in the edit script are abstracted. Each edit operation is positioned with respect to the abstract edit relevant context's AST.

- A **partially abstract, context-aware edit script** is derived from multiple changed methods. The script consists of a sequence of AST edit operations and partially abstract context. Some used concrete identifiers are abstracted. Each edit operation is positioned with respect to the partially abstract edit relevant context's AST.

- **Systematic edits** are similar, but not necessarily identical changes applied to multiple locations in software. In our research, a systematic edit is represented either as an abstract, context-aware edit script or a partially abstract, context-aware edit script, depending on how the edit is created.

Chapter 3

Related Work

This chapter elaborates prior work relevant to our exemplar change based systematic editing tools in Section 3.1, and that relevant to our common change based opportunistic refactoring tool in Section 3.2.

3.1 Exemplar Change based Systematic Editing

The related work includes program differencing, refactoring, source transformation languages, simultaneous editing, programming by demonstration, edit location suggestion, and automated program repair.

3.1.1 Program Differencing

Program differencing takes two program versions and matches names and structure at various granularities, e.g., lines [43], abstract syntax tree nodes [26, 101], control-flow graph nodes [4], and program dependence graph nodes [40]. For example, the ubiquitous tool *diff* computes line-level differences per file using the longest common subsequence algorithm [43]. JDiff computes CFG-node level matches between two program versions based on similarity in node labels and nested hammock structures [4]. ChangeDistiller computes syntactic differences using a hierarchical comparison algorithm [26]. It matches statements, such as method invocations, using *bigram string simi-*

larity, 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 [50, 51]. Our systematic editing tools extend ChangeDistiller and go beyond these approaches by deriving an edit script from program differences, abstracting the script, and then applying it elsewhere.

3.1.2 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure [28]. Refactorings may require applying one or more elementary transformations to multiple code locations. Refactoring engines in IDEs automate many commonly recognized refactoring types such as *rename a variable*. However, most refactoring tools are developer-driven, requiring developers to explicitly invoke the functionality and provide all necessary information for the program transformation [25, 30, 34, 73, 78, 85, 91, 96]. Few refactoring tools [27, 31] infer developers' refactoring intention by matching developers' edits with predefined refactoring edit patterns at runtime, and suggest refactoring completion automatically. However, all refactoring tools are confined to predefined, semantic-preserving transformations. There is no easy way for developers to define customized refactoring. In contrast, our systematic editing tools automatically infer and apply general purpose program transformations, including semantic-modifying transformations, based on code change examples. Our clone removal refactoring tool automatically recommends developers edit locations to apply refactoring and creates refactored versions based on

similar edits applied to similar code.

3.1.3 Source Transformation Languages

Source transformation languages and tools allow users to define source transformation by describing code patterns to match and update actions to take. The transformation engines then search for code matching the patterns and take specified update actions to transform the code. However, most of these tools [14, 16, 35, 58, 95] are difficult to use because they require developers to prescribe program transformation from scratch using manipulation primitives on a syntax tree or the grammar underlying a programming language, instead of on the source code familiar to common developers. iXj [12] is simpler to use because it allows programmers to define code transformations in a visualized way by selecting an expression to change, generalizing some of its subexpressions with wildcards, and describing changes based on the generalized representation. However, it only handles trivial *update* edits for expressions. SmPL [79] 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 [79] 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 both data and control dependence constraints that edits have on their surrounding context.

3.1.4 Simultaneous Editing

When users interactively demonstrate their edits in one context, simultaneous text editing tools [72, 92] automatically replicate the *identical lexical* edits on preselected code fragments. Compared with these tools, our systematic editing tools infer systematic edits by comparing two versions of a changed method, which allows developers to freely make edits as they like. CloneTracker [19] takes the output of a clone detector as input, maps corresponding lines in the clones, and then echoes edits in one clone to another upon a user's request. The Clever version control system [76] detects code clones and changes to them, and then recommends edit propagation among clones. Different from these tools, SYDIT and LASE extract edit relevant context using static analysis and abstract the inferred edit scripts so that they are applicable to locations containing different contexts or using different identifiers.

3.1.5 Programming by Demonstration

Programming by demonstration (PbD) is an end-user development technique for teaching a computer or a robot new behaviors by demonstrating the task to transfer directly instead of programming it through machine commands [17]. Such transformation tools either record human experts' edit actions on several examples [59, 60, 67, 100] or look at several text change examples [36, 69, 77] in order to synthesize a program automating repetitive text editing tasks. Specifically, Gulwani et al. [36] have applied program synthesis based on input/output examples to different domains, like spreadsheet table transformation [37], semantic string transformation [86], natural deduction problem and solution generation [2], etc.. Different from these text editing tools, our systematic editing tools do not treat programs as pure text. Instead,

we exploit program syntactic structures to characterize edit context so that the inferred program transformation is applicable to programs with similar but different syntactic structures.

3.1.6 Edit Location Suggestion

Code searching and bug finding tools are used to identify code fragments which often require similar edits. Query-based code search engines [66, 97] look for code fragments of interest based on queries written in domain-specific languages. For instance, the approach of Want et al. [97] allows developers to write a query for program dependence graph in order to find code locations sharing certain control or data dependence relations. PQL [66] supports users to write declarative rule-based queries for sequence of events associated with a set of related objects to correct erroneous execution on the fly. Some bug finding tools mine implicit API migration rules from software version history [74] or mine API usage rules from large software systems [64], detect code locations violating the mined rules, and report them as edit locations to enforce the detected rules. Some other bug finding tools detect code clones and establish mappings between them [63, 75] to suggest locations which contain inconsistent identifier mappings or update inconsistently with their peers so that developers edit these locations to solve the inconsistency. Although these tools suggest edit locations, none of them transform code.

3.1.7 Automated Program Repair

Automatic program repair generates candidate patches and checks correctness using compilation and testing. GenProg [62] generates candidate patches by replicating, mutating, or deleting code *randomly* from the existing

program. PAR [49] improves the approach by also including ten common bug fix patterns mined from Eclipse JDT's version history when generating candidate repairs. These approaches totally depend on full coverage of test cases and existence of candidate repairs in a buggy program. Some bug fixing tools look for concurrency bugs [45] or security bugs [88] based on patterns and generate fixes for them according to predefined bug fixing strategies. Specification-based approaches require developers to define constraints to express expected behaviors [98] or data structure properties [18, 21] of correct programs, run a program against the constraints, detect constraint violations, and generate candidate fixes based on predefined repair strategies. To remove specification burden on developers, ClearView [82] infers invariants from program successful runs, detects invariant violations in program failing runs, modifies program states accordingly to make failing runs succeed. Compared with these tools, SYDIT and LASE infer general-purposed edits from user-provided examples and only apply them to locations containing edit-relevant context.

3.2 Common Change based Clone Removal Refactoring

The related work includes clone removal refactoring, automatic procedure extraction and empirical studies of code clones.

3.2.1 Clone Removal Refactoring

Based on code clones detected by various techniques [44, 47, 56], many tools identify or rank program refactoring opportunities [6, 33, 38, 39, 94]. For instance, Balazinska et al. [6] define a clone classification scheme based on various types of differences between clones and automate the classification to help developers assess refactoring opportunities for each clone group ac-

cording to its category. Higo et al. [39] and Goto et al. [33] rank clones as refactoring candidates based on coupling or cohesion metrics. Others [38, 94] integrate evolution information in software history to rank clones that have been repetitively or simultaneously changed in the past. While these tools detect refactoring opportunities for clones, they do not automatically refactor code.

A number of techniques [7, 41, 46, 57, 89] automate clone removal refactorings by factorizing the common parts and by parameterizing their differences using *strategy* design pattern or a *form template method* refactoring. Similar to RASE, these tools insert customized calls in each original location to use newly created methods. Juillerat et al. [46] automate *introduce exit label* and *introduce return object* refactorings supported by RASE. However, for variable and expression variations, CloRT [7] and Juillerat et al.'s approach [46] define extra methods to mask the differences, while RASE passes these variations as arguments of the extracted method. CloRT was applied to JDK 1.5 to automatically reengineer class level clones and, similar to our results, the reengineering effort led to an increase in the total size of code due to the numerous but simple methods created. Hotta et al. [41] use program dependence analysis to handle gapped clones—trivial differences inside code clones that are safe to be factored out such that the *form template method* refactoring is applicable. Krishnan et al. [57] use PDGs of two programs to identify a maximum common subgraph so that the differences between the two programs are minimized and fewer parameters are introduced. Unlike RASE, none of these tools handle type variations, when performing generalization tasks.

3.2.2 Automatic Procedure Extraction

Komondoor et al. [54, 55] extract methods based on the user-selected or tool-selected statements in one method. The *extract method* refactoring in the Eclipse IDE requires contiguous statements, whereas these tools handle non-contiguous statements. Program dependence analysis identifies the relation between selected and unselected statements and determines whether the non-contiguous code can be moved together to form extractable contiguous code. Similar to Rase, Komondoor et al. [55] apply *introduce exit label* refactoring to handle exiting jumps in selected statements. Tsantalis et al. [93] extend the techniques by requiring developers to specify a variable of interest at a specific point only. They use a block-based slicing technique to suggest a program slice to isolate the computation of the given variable. These approaches are only focused on extracting code from a single method. Therefore, they do not handle extracting common code from multiple methods and resolving the differences between them as Rase does.

3.2.3 Empirical Studies of Code Clones

Many empirical studies on code clones [5, 11, 32, 48, 52] find that removing clones is not necessary nor beneficial. Bettenburg et al. [11] report that only 1% to 3% of inconsistent changes to clones introduce software errors, indicating that developers are currently able to effectively manage and control clone evolution. Kim et al. [52] observe that many long-lived, consistently changed clones are not easy to refactor without modifying public interfaces. Göde et al. [32] reveal that there is a significant discrepancy between clones detected by state-of-the-art clone detectors and clones removed by developers. In 66% cases, developers refactor only a small part of a larger clone.

These empirical studies show that the removal of code clones is not always necessary or beneficial. While these studies use longer version histories or larger programs than our evaluation, none of these studies, automatically perform refactorings as RASE does. RASE thus improves over their methodology by eliminating human judgement when determining the feasibility of edits.

Chapter 4

Automating Systematic Edits

When developers make similar but not identical changes to multiple locations, they usually first edit one location, run tests to check correctness of the edit, copy and paste the edit to other locations, and customize the edits as needed. The copy-paste-customize process to propagate changes is tedious and error-prone. Developers may wrongly customize edits and produce syntactically valid but semantically invalid programs. A crucial observation is that similarly changed locations usually share a syntactic structure template. For instance, locations which are similarly refactored to enumerate elements in a data structure usually contain a loop structure. We hypothesize that by extracting out the template, we can automatically edit multiple locations in similar ways when developers demonstrate the edit in one location. Existing clone detection tools [9, 10, 13, 29] cannot always help to identify the template because they all start with similar code chunks. Instead, our tool extracts the template based on a systematic edit. Starting with a systematic edit, our tool slices code with control and data dependence analysis to identify a code template relevant to the edit. The resulting template may not correspond to code clones because it can represent a sequence of non-contiguous code chunks. It is then used to search for a match in user-selected code, which determines how to customize and apply the demonstrated edit for the specific location.

Existing tools provide limited support for systematic editing. For ex-

ample, the search and replace feature in a text editor supports only simple text replacements. It cannot handle non-contiguous edits, nor edits requiring adaption to different contexts. Simultaneous editing tools [19, 72, 92] require developers to interactively demonstrate their edit in one context and the tool replicates *identical lexical* edits on the preselected code fragments. Similar to search-and-replace, these tools cannot automatically customize edits for different contexts. Source transformation languages and tools [23, 35, 58, 79, 95] automate program transformations based on the edit scripts prescribed by developers with certain domain-specific languages. However, learning to use a source transformation language and precisely describing each transformation are challenging for developers.

This chapter introduces a novel approach to investigate inferring a systematic program transformation from one change example, customizing the transformation to each user-selected code fragment and applying the result. We implement the approach in a tool called SYDIT. Given an exemplar changed method, SYDIT uses a syntactic differencing algorithm to represent code changes as an AST edit script. It then generalizes the edit script by extracting a context relevant to the edit based on program containment, control and data dependence analysis, and by abstracting identifiers of types, methods, and variables occurring in the edit. The generalized edit script is an abstract, context-aware edit script. For each user-specified target method, SYDIT customizes the edit script by establishing context matching between the edit script and the method in order to correctly place the code changes, and by establishing identifier mapping between the two, in order to correctly use existing identifiers in the target. Finally, it applies the customized edit to suggest a modified version for developers to review.

Our evaluation with SYDIT demonstrates that our approach is effective in correctly inferring, customizing, and applying systematic edits based on single examples. We use 56 systematic edit pairs from five large software projects as an oracle. For each pair, we use SYDIT to infer a systematic edit from one method and apply the edit to the other. 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. It mimics human programmers correctly on 70% (39/56) of the edits. To our knowledge, SYDIT is the first tool to perform nontrivial general-purpose abstract edits to similar but different contexts based on a single code change example.

4.1 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 4.1 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 `getAllLaunchConfigurations()`, 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 5, and 7-11 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

A_{old} to A_{new}

```
1. public ILaunchConfiguration[] getLaunchConfigurations (ILaunchConfigurationType
2.      type) throws CoreException {
3.     Iterator iter = getAllLaunchConfigurations().iterator();
4.     List configs = new ArrayList();
5. +   ILaunchConfiguration config = null;
6.     while (iter.hasNext()) {
7. -       ILaunchConfiguration config = (ILaunchConfiguration)iter.next();
8. +       config = (ILaunchConfiguration)iter.next();
9. +       if (!config.inValid()) {
10.+           config.reset();
11.+       }
12.        if (config.getType().equals(type)) {
13.            configs.add(config);
14.        }
15.    }
16.    return (ILaunchConfiguration[])configs.toArray(new ILaunchConfiguration
17.      [configs.size()]);
18.}
```

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.}
```

Figure 4.1: Systematic edit from revisions of `org.eclipse.debug.core`

to multiple methods cannot be applied using the search and replace feature because the edited text in different methods is different. These edits cannot be applied using existing refactoring engines in IDE, either, 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.

Using SYDIT, Pat makes changes only to `mA`, provides `mA` as a changed example, and specifies `mB` as a code fragment to change similarly. SYDIT infers a systematic edit from `mA`, applies it to `mB`, and suggests the resulting version for developers to review.

With more details, when given $\texttt{mA}_{old}$ and $\texttt{mA}_{new}$, SYDIT applies a syntactic program differencing algorithm to the two versions to represent the exemplar edit (lines 5, and 7-11) as a statement-level Abstract Syntax Tree (AST) together with a sequence of AST node insertions, deletions, updates, and moves. For each generated AST edit operation, it performs data and control dependence analysis to extract relevant statements, i.e., statements which contain, or are control or data dependent on by the edited statement. By extracting the relevant statements together with the edited statements, SYDIT abstracts away unchanged statements irrelevant to the exemplar edit. The extracted code is the *edit relevant context* of the exemplar edit. To generalize a systematic edit out of the demonstrated edit, SYDIT recalculates position of each edit operation with respect to the extracted code and abstracts identifiers of types, methods, and variables used in the code. The abstraction of both edit positions and identifiers creates an *abstract, context-aware* edit script, which is applicable to code fragments containing different contexts or using different identifiers. Figure 4.2 shows the generalized abstract edit script.

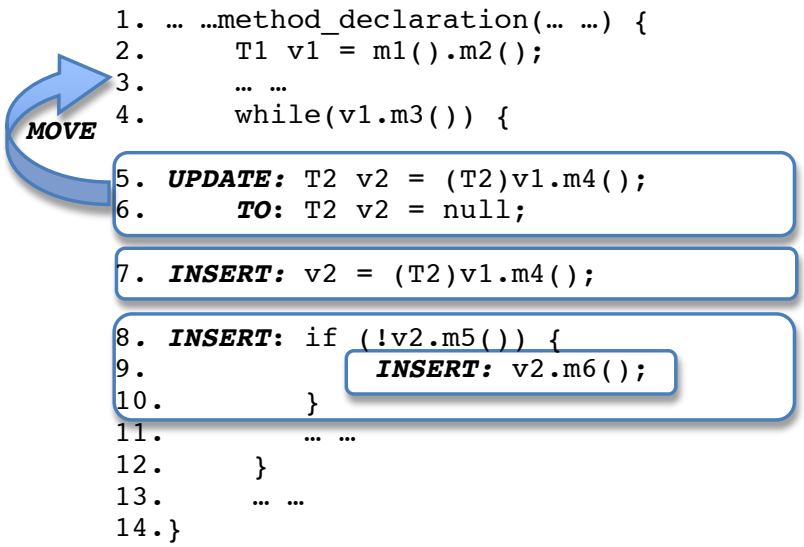

Figure 4.2: Abstract edit script

Currently, SYDIT does not guarantee to generate an edit script containing the fewest edits. For this example, a shorter edit script should have been one insertion for the variable declaration statement (line 5 in mA), one update for a variable assignment statement (line 7-8 in mA), and two other insertions (line 9-11 in mA), summing up to four edit operations, instead of five as shown in the generated edit script.

Next, SYDIT tries to apply the inferred edit script to user-specified method, mB. It looks for code snippet(s) in mB matching the extracted context (i.e., lines 2, 5 and 6 in mB). If there is a match, SYDIT recalculates position of each edit operation with respect to mB's context, and establishes mappings between abstract identifiers and concrete identifiers used in mB, in order to customize the edit script and apply the result.

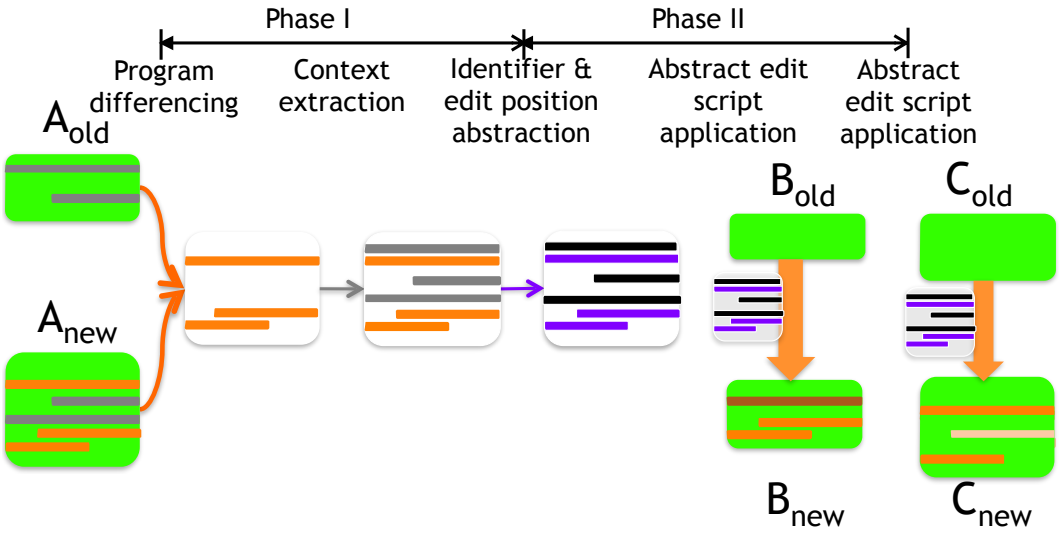

Figure 4.3: Two phases of SYDIT

4.2 Approach

This section describes the two phases of SYDIT, as shown in Figure 4.3. Phase I takes as input an old and new version of method `mA` as its exemplar edit, and creates an abstract, context-aware edit script from $\texttt{mA}_o$ and $\texttt{mA}_n$. Phase II applies the edit script to new contexts, such as `mB` and `mC`, producing a suggested version for each new context, $\texttt{mB}_s$ and $\texttt{mC}_s$. We first summarize the steps in each phase and then describe each step in detail.

Phase I: Creating Edit Scripts

- SYDIT compares two versions of an exemplar changed method, $\texttt{mA}_o$ and $\texttt{mA}_n$, and describes the differences as a sequence of statement-level AST node 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 containment, control and data 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 the identified context and by replacing all concrete identifiers of types, methods, and variables with abstract identifier names.

Phase II: Applying Edit Scripts

- SYDIT looks for a sub-context in ASTs of target methods, `mB` and `mC` for instance, to match the abstract context.

- If there is a match in `mB`, 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 $\texttt{mB}_s$. Similar process is done for `mC`.

4.2.1 Phase I: Creating Abstract Edit Scripts

This section explains how SYDIT creates an abstract, context-aware edit script from single changed example.

4.2.1.1 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 [26]. ChangeDistiller generates statement-level AST node insertions, deletions, updates, and moves. 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., expression statements), 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 set sizes. If the similarity is above an input threshold, σ, ChangeDistiller includes the two leaves 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 other loop nodes. Second, we allow unmatched leaf nodes to tentatively map to unmatched inner nodes using bigram string similarity if they fail both leaf node matching pass and inner node matching pass mentioned above in order to increase successful matching rate. The intuition behind is the more nodes we manage to match, the fewer edit operations are generated to represent the difference, and the more likely the resulting edit script is to be optimal in

terms of length. 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 as an inner node. Without mapping them together, we may end up with a redundant edit script consisting of a deletion of an empty `catch` clause and insertions for a non-empty `catch` clause, while a better edit script should have only included node insertions into the `catch` clause.

With AST node mappings established, SYDIT describes edit operations with respect to the original method $\texttt{mA}_o$ as follows:

delete (Node u): delete node u.

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 and insert it as the $(k+1)^{th}$ child of v.

update (Node u, Node v): replace u with v. This step replaces the AST node type and content of u with v's, but maintains all of u's relationship with its AST parent and children.

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 edit operations, although in reality there may be only partial order between them.

Figure 4.4 presents the mapping for our example, where 'O' is a node in the old version $\texttt{mA}_o$ and 'N' is a node in the new version $\texttt{mA}_n$. Below we show the concrete edit script Δ_A that transforms $\texttt{mA}_o$ into $\texttt{mA}_n$ for our example:

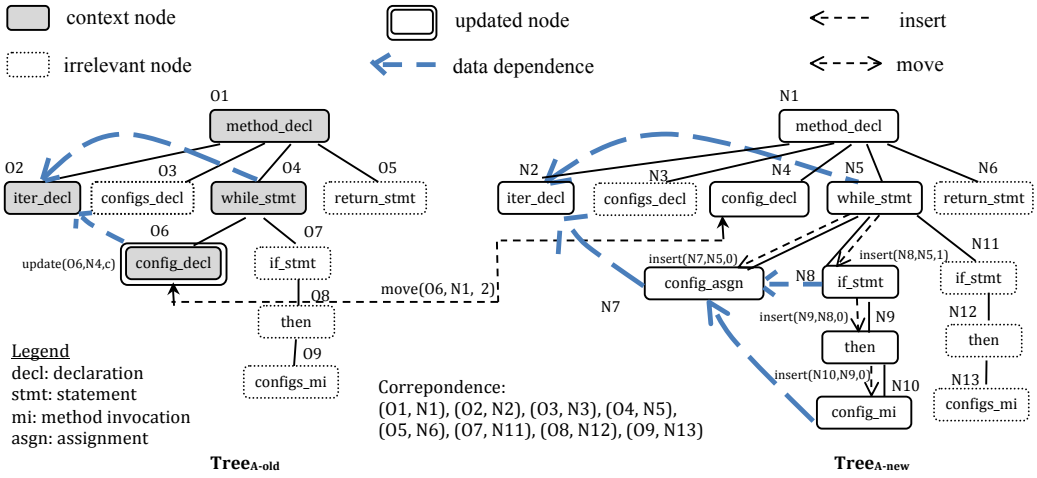

Figure 4.4: Syntactic edit extraction for mA

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();'

4.2.1.2 Extracting Edit Contexts

SYDIT identifies relevant statements in both the old and new versions for each edit operation and *projects* them to the old version in order to extract the edit-relevant context.

For each $e_i \in \Delta_A$, SYDIT analyzes $\texttt{mA}_o$ and $\texttt{mA}_n$ to find the unchanged nodes on which changed nodes in e_i are containment, control, or data dependent. 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 directly enclosing the edited statements. 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 operation 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 the context and thus may sacrifice coverage.

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

- *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.

- *Control dependence*: node y is control dependent on node x if x makes a decision about whether y executes or not. For instance, whether N7 executes or not depends on the decision made at the `while` condition of N5. Therefore, N7 is control dependent on N5.

- *Data dependence*: node y is data dependent on node x if y uses a variable whose value is defined in node x. For example, N10 in Figure 4.4 uses variable `config`, whose value is defined in N7. Therefore, N10 is data dependent on N7.

To extract the context for an edit script, we compute containment, control, and data 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 $\texttt{mA}_n$ onto corresponding nodes in the old version $\texttt{mA}_o$ 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 $\texttt{mA}_o$ that the deleted node, u, depends on.

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

move (u, v, k)**:** The algorithm finds nodes on which u depends in both $\texttt{mA}_o$ and $\texttt{mA}_n$. The nodes in the new version help guarantee dependence

relationships after the move. The algorithm projects the nodes from $\texttt{mA}_n$ onto corresponding nodes in $\texttt{mA}_o$ and then unions the two sets.

update (u, v)**:** The algorithm finds nodes on which u depends in $\texttt{mA}_o$ and those on which v depends in $\texttt{mA}_n$. It projects the nodes from $\texttt{mA}_n$ into corresponding nodes in $\texttt{mA}_o$ and then unions the two sets.

Consider insert (N7, N5, 0) from Figure 4.4. The inserted node N7 is control dependent on N5, and data dependent on N2 and N4. Projecting 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 $\texttt{mA}_o$, which yields C3 = {O1}. Finally, we union C1 and C3 to get the context node set {O1, O2, O4} for the move operation. Figure 4.4 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 4.3 shows how varying context affects SYDIT's coverage and accuracy.

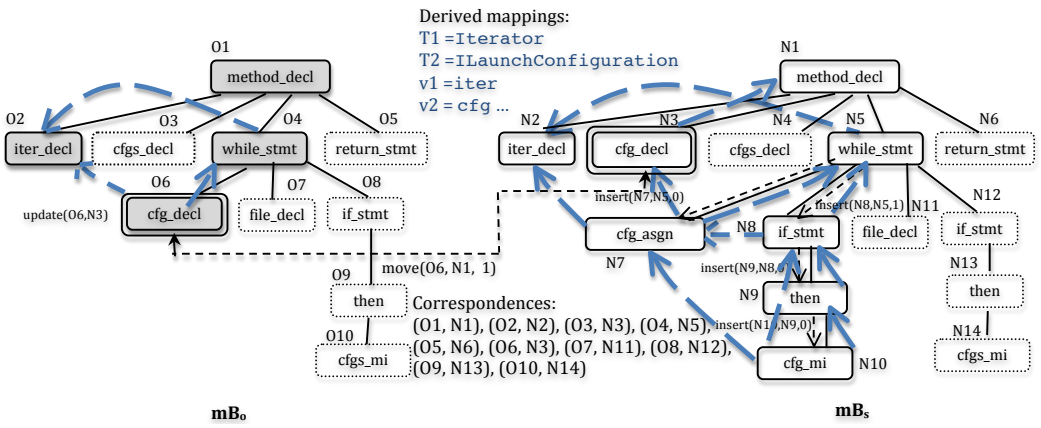

Figure 4.5: Syntactic edit suggestion for `mB`

4.2.1.3 Abstracting Identifiers and Edit Positions

At this point, the inferred edit script uses concrete identifiers of types, methods, variables, and positions edit operations with respect to the example method. To make the edit script more general and applicable to code fragments using different identifiers or containing different contexts, we abstract identifiers and edit operation positions.

To abstract identifiers, we replace all concrete identifiers of used types, methods, and variables with equivalent abstract representations: `T$`, `m$`, and `v$` respectively. Each unique concrete identifier corresponds to a unique abstract one. For example, we convert the concrete expression `!config.inValid()` in Figure 4.4 to `!v2.m5()` in Figure 4.6.

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 extracted 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

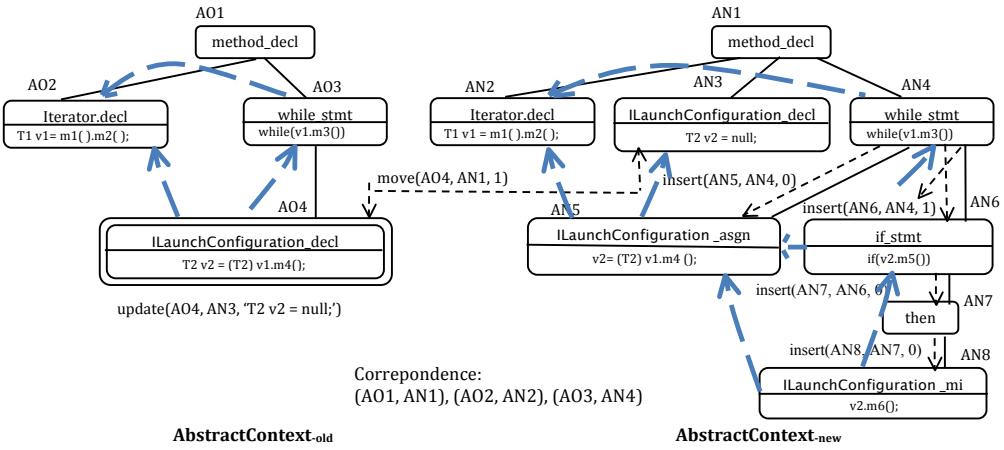

Figure 4.6: Abstract, context-aware edit script

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 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.

4.2.2 Phase II: Applying Abstract Edits

This section describes how SYDIT applies an edit script Δ to a method `mB`, producing a modified method $\texttt{mB}_s$ for suggestion.

4.2.2.1 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 identifiers. To simplify this process, we first abstract the identifiers in `mB`. We use the procedure as described in Section 4.2.1.3 to create $\texttt{mB}_{Abstract}$ from `mB`. For concision in this section, we simply use `mB` instead of $\texttt{mB}_{Abstract}$.

This problem can be posed as the labeled subgraph isomorphism problem. Although we experimented with an off-the-shelf implementation [68], 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 4.2.2.2 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, $x \in \texttt{AC}$, $y \in \texttt{mB}$, 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 `if`s 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`,

Algorithm 1: Matching Abstract Context to Target Tree

Input: AC, mB /* abstract context and target tree */
Output: M /* a set of node matches from AC to mB */
/* 1. create candidate leaf exact matches */
CL:= $\emptyset$; M := $\emptyset$; S:= $\emptyset$;
foreach *leaf node* $x \in$ AC, *leaf node* $y \in$ mB **do**
  CL:= CL $\cup$ $\{(x, y) \mid (x, y)$ is an exact match$\}$;
end
repeat
  /* 2(a). create node matches based on unique 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 $\cup$ $\{(u, v) \mid ((u, v)$ is a type match, $u \in p1$, $v \in p2$, and they appear in corresponding positions on $p1$ and $p2\}$;
    end
  end
  /* 2(b). select the best path match between candidate path matches */
  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_m with the maximum $pathMatchScore(p1, p2,$ M$)$;
    M:= M $\cup\{(u, v) \mid u \in p1$ and $v \in p2$ where $p2 = root_{mB} \rightsquigarrow y_m\}$;
  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_m with the maximum $LCSMatchScore(x, y,$ M$)$;
    M:= M $\cup\{(u, v) \mid u \in (root_{AC} \rightsquigarrow x)$ and $v \in root_{mB} \rightsquigarrow y_m\}$;
  end
  /* 3. establish symbolic identifier mappings */
  foreach $(u, v) \in$ M **do**
    S:= S $\cup$ $\{$(Tn, Tm), (vi, vj), and/or (mk, ml) that are supported by $(u, v)\}$;
  end
  $removeConflicts$(S, M);
  /* 4. relax constraints to add leaf matches */
  CL:= CL $\cup$ $relaxConstraints$(AC, M);
until CL *reaches fixpoint*;

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

```
t1 := p1's bottom-up iterator;
t2 := p2's bottom-up iterator;
while t1.hasPrev() ∧ 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;
```

(a) If there exists single 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 established match set M, we regard it as the best path match and add constituent (u, v) matches to M.

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() ∧ t2.hasPrev() do
    u := t1.prev();
    v := t2.prev();
    if (u, v) ∈ M then
        counter++;
    end
end
return counter;
```

(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 two 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 M contains a node match (u, v) where u precedes x as a sibling, while v is a sibling between y_2 and y_3. SYDIT prefers the path match $((root_{AC} \rightsquigarrow x), (root_{mB} \rightsquigarrow y_3))$, since this choice is consistent with an already established 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 aligns nodes [43], our approach matches leaf nodes based on established matches in M.

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;
```

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`.

Function removeConflicts(mappings S, matches M)

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

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`.

```
Function relaxConstraints(context AC, matches M)
    CL := ∅
    foreach leaf node z ∈ AC such that ∄(z, w) ∈ M do
        u := z;
        repeat
            u := parent(u);
        until u=null ∨∃(u, v) ∈ M;
        if u ≠null then
            CL := CL ∪{(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;
```

At any point in this process, if every node in the abstract context `AC` has a match in `M`, we proceed to derive concrete edits customized to `mB`, described in Section 4.2.2.3. 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.

4.2.2.2 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 identifiers of types, methods, and variables, 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 a subgraph in the target method's graph which is isomorphic to the abstract context's graph.

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 [68], 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.

4.2.2.3 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 4.2.1.3.

The substitution is based on the symbolic identifier mappings established in Section 4.2.2.1, e.g., `(T1, T2)`, and the abstract-concrete identifier mappings established in Section 4.2.1.3, e.g., `(T1, int)`, `(T2, int)`. For this specific case, each time `T1` occurs in Δ, it is directly mapped to `T2` and thus indirectly mapped to `int` in Δ_B via `T2`. Some edits in Δ use symbolic

identifiers that only exist in the new version, thus the corresponding concrete identifiers do not exist in the original code of $\texttt{mA}_o$ or $\texttt{mB}_o$. In this case, we borrow the concrete identifiers from $\texttt{mA}_n$. For example, the identifier `inValid` used in Figure 4.1 only exists in $\texttt{mA}_n$, but does not in $\texttt{mA}_o$ or $\texttt{mB}_o$. We thus just use the name from $\texttt{mA}_n$, storing it 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 4.5).

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 $\texttt{mB}_s$.

4.3 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` [52]. 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 ChangeDistiller. We identified method pairs `mA` and `mB` that share at least one common syntactic edit between the old and new versions and their content is at least 40% similar. We use the following similarity metric:

$$\texttt{similarity(mA, mB)} = \frac{|matchingNodes(mA, mB)|}{size(mA) + size(mB)} \tag{4.1}$$

Table 4.1: SYDIT's capabilities, coverage, accuracy, and similarity for k=1, upstream control and data dependences

	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)	

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 content means source and target methods do not use exactly the same type, method, or variable identifiers. To test this range of functionality in SYDIT, we chose at least 7 examples in

each category. Table 4.1 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 ($\texttt{mA}_o$, $\texttt{mB}_o$) in the old version that changed similarly to become ($\texttt{mA}_n$, $\texttt{mB}_n$) in the new version, SYDIT generates an abstract, context-aware edit script from $\texttt{mA}_o$ and $\texttt{mA}_n$ and tries to apply the learned edits to the target method $\texttt{mB}_o$, producing $\texttt{mB}_s$. In Table 4.1, *matched* is the number of examples for which SYDIT matches the learned context to the target method $\texttt{mB}_o$. 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 (4.1).

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 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

A_{old} to A_{new}

```
1. private void paintSides(GC g, MergeSourceViewer tp, Canvas canvas, boolean right) {
2.    ... ...
3. - g.setLineWidth(LW);
4. + g.setLineWidth(0 /* LW */);
5.    ... ...
6.}
```

B_{old} to B_{new}

```
1. private void paintCenter(Canvas canvas, GC g) {
2.    ... ...
3.    if (fUseSingleLine) {
4.       ... ...
5. -     g.setLineWidth(LW);
6. +     g.setLineWidth(0 /* LW */);
7.       ... ...
8.    } else {
9.       if(fUseSplines){
10.         ... ...
11.-        g.setLineWidth(LW);
12.+        g.setLineWidth(0 /* LW */);
13.         ... ...
14.      } else {
15.           ... ...
16.-          g.setLineWidth(LW);
17.+          g.setLineWidth(0 /* LW */);
18.      }
19.   }
20.   ... ...
21.}
```

Figure 4.7: A non-contiguous identical edit script (NI) for which SYDIT cannot match the change context (`org.eclipse.compare`: v20060714 vs. v20060917)

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 4.7 (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. SYDIT does not automate the systematic edit because it always tries to find the only *one* best match for the edit's relevant context or code

pattern. If there is more than one best match, such as three, SYDIT gives up and concludes that it cannot apply the systematic edit because there is ambiguity in context matching process. This may be counterintuitive because it should have allowed more than one best match and applied the same edit to all matching locations. Additional user assistance would solve this problem.

A_{old} to A_{new}

```
1. public IActionBars getActionBars() {
2. + IActionBars actionBars = fContainer.getActionBars();
3. - if (fContainer == null) {
4. + if (actionBars == null && !fContainerProvided) {
5.     return Utilities.findActionBars(fComposite);
6.   }
7. - return fContainer.getActionBars();
8. + return actionBars;
9. }
```

B_{old} to B_{new}

```
1. public IServiceLocator getServiceLocator() {
2. + IServiceLocator serviceLocator = fContainer.getServiceLocator();
3. - if (fContainer == null) {
4. + if (serviceLocator == null && !fContainerProvided) {
5.     return Utilities.findSite(fComposite);
6.   }
7.   - return fContainer.getServiceLocator();
8.   + return serviceLocator;
9. }
```

B_{old} to $B_{suggested}$

```
1. public IServiceLocator getServiceLocator() {
2. + IServiceLocator actionBars = fContainer.getServiceLocator();
3. - if (fContainer == null) {
4. + if (actionBars == null && !fContainerProvided) {
5.     return Utilities.findSite(fComposite);
6.   }
7. - return fContainer.getServiceLocator();
8. + return actionBars;
9. }
```

Figure 4.8: A non-contiguous, abstract edit script for which SYDIT produces edits equivalent to the developer's (`org.eclipse.compare`: v20061120 vs. v20061218)

Figure 4.8 shows a complex example (from `org.eclipse.compare` v20061120 vs. v20061218) that SYDIT handles well. Although the methods $\texttt{mA}_o$ and $\texttt{mB}_o$ use different identifiers, SYDIT successfully matches $\texttt{mB}_o$ to the abstract context `AC` derived from `mA`, creating a version $\texttt{mB}_s$, which is semantically equivalent to the manually crafted version $\texttt{mB}_n$.

In addition to these 56 pairs, we collected six examples that perform similar edits on multiple contexts—on at least 5 different methods. Table 4.2 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

Table 4.2: Replicating similar edits to multiple contexts

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

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 4.7. 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 $\texttt{mA}_o$ and $\texttt{mA}_n$ and then apply it back to $\texttt{mA}_o$. We then compare the SYDIT generated version with $\texttt{mA}_n$. Similarly, we derive an edit script from $\texttt{mB}_o$ and $\texttt{mB}_n$ and compare the application of the script to $\texttt{mB}_o$ with $\texttt{mB}_n$. 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 4.3 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 4.4 explores how different context characterization strategies af-

Table 4.3: SYDIT's context characterization

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

fect 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 4.2.1.2).

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 4.4 reports on the effectiveness of identifier abstraction for types (T), methods (M), and variables (V). 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. Sur-

Table 4.4: SYDIT's sensitivity to 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%

prisingly, 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 statement-level AST edit script 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 4.5 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 to AST node mismatches, which produce incorrect updates or moves, and consequently SYDIT's coverage, accuracy and

Table 4.5: SYDIT's sensitivity to input threshold σ used in ChangeDistiller

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%

similarity decrease.

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 $\sigma = 0.5$ for ChangeDistiller, which together achieve 82% coverage, 70% accuracy, and 96% similarity.

4.4 Summary

SYDIT improves developer efficiency and program quality by automating systematic edits in user-selected targets and presenting suggested versions for developers to check and improve if necessary. It does not guarantee the correctness of generated code. It extracts context relevant to a systematic edit in a way to observe control and data dependence relationships between the edit and surrounding context. Therefore, by establishing matches between a user-selected method and the relevant context's code pattern, it enforces all constraints the edit has on target code. This enforcement provides two guarantees: (1) the edit produces a syntactically valid program; and (2) the edit alters program syntax and semantics in an expected way, as demonstrated by the example. However, if applying the systematic edit is not sufficient to produce a semantically correct program, SYDIT has no idea what extra edit to add.

On the other hand, SYDIT exploits existing identifiers instead of creating new identifiers to concretize wildcards in a systematic edit. Therefore, when the edit introduces some new identifiers, SYDIT will not create new identifiers to concretize them for target code, producing an incorrect program. Developers must choose a name in this case.

SYDIT's dependence on developers for edit method selection may still burden programmers, especially in cases when finding edit locations is more challenging than applying edits. If developers forget to provide target methods, SYDIT cannot locate them. We can extend SYDIT to locate code for editing by searching for code matching the abstract context in a systematic edit. However, our evaluation in the next chapter shows that the abstract context generalized from one exemplar edit is not sufficient to precisely find edit locations. The reason is that SYDIT creates the abstract context by uniformly generalizing control or data relevant context of every edit operation and abstracting every concrete identifier in an exemplar edit, although developers may only want to generalize some of them. Given a single exemplar edit, SYDIT has no clue how developers want to generalize the context versus use concrete. The experience with SYDIT leads us to a more challenging and interesting problem about how to find locations to apply systematic edits, which is to be discussed in the next chapter.

Chapter 5

Finding Systematic Edit Locations

This chapter addresses the problem of both correctly identifying locations in need of systematic edits and making correct edits to each location. Existing tools either suggest code locations or transform code, but not both, except for specialized or trivial edits. For instance, code search either requires developers to write code pattern queries [66, 97], or automatically mines API usage patterns from large software or version control systems [22, 64] to find edit locations matching or violating the desired code patterns. However, these tools do not transform code. Bug fixing tools [45, 88] integrate expertise about bug patterns and fixing strategies for certain security or concurrency bugs, automatically search for locations matching the patterns and fix bugs based on predefined strategies. However, they do not allow users to customize code patterns or transformation strategies. Chapter 4's approach applies program transformation to user-selected edit locations but does not automatically find the edit locations.

This chapter introduces a novel approach to use multiple change examples to infer a systematic edit, whose inherent edit-relevant context serves as a template to correctly identify edit locations. We implement the approach in a tool called Lase. It generalizes a systematic edit by extracting common edit as well as relevant context shared between exemplar edits.

Developers specify two or more example edits by hand. Lase performs

syntactic program differencing to represent code changes in each method as a statement-level AST edit script. It then identifies the longest common edit operation subsequence between edit scripts. Any uncommon edit operation specific to some examples is filtered out because they do not generalize to all examples. Next, LASE compares the identified edit operations from different scripts to create a general representation. If edit operations agree on identifier usage for types, methods, and variables, LASE uses these concrete identifiers in the resulting edit script. Otherwise, it creates wildcards `T$`, `m$`, `v$`, separately, to abstract away any divergence. Similar to SYDIT, based on the identified common edit, LASE leverages control and data dependence analysis to extract edit relevant context in each method. It exploits common subtree extraction and clone detection techniques to identify code commonality between examples. By combining the information of edit relevant context per method and code commonality among methods, LASE decides the common edit relevant context. This context and the common edit construct a partially abstract, context-aware edit script. The commonality of both edit and relevant context makes sure that LASE only extracts and generalizes information demonstrated by all examples, which is more likely to generalize to edit locations which are not provided by developers but waiting for LASE to correctly suggest them out.

LASE then uses the inferred systematic edit to find candidate edit locations by establishing matches between the code pattern and every method in the whole project or software system. If a method includes all concrete identifiers used in the pattern and has a code snippet matching the pattern's skeleton, the method is selected as a candidate. Again, we use common subtree extraction technique to decide whether a method contains a subtree matching

the code pattern's AST. For each candidate, LASE customizes and applies the systematic edit to transform code.

Our evaluation with LASE demonstrates that our approach is effective in both correctly finding edit locations and correctly making edits. We use 24 repetitive bug fixes that require multiple check-ins from two open source projects as an oracle. For these bugs, developers applied repetitive bug fixes because the initial patches were either incomplete or incorrect [80]. We evaluate LASE by learning edit scripts from the initial patches and determining if LASE correctly derives the subsequent, supplementary patches. On average, LASE identifies edit locations with 99% precision and 89% recall. The accuracy of applied edits is 91%, i.e., the tool-generated version is 91% similar to the developer's version. To our knowledge, LASE is the first tool to learn a general purpose abstract edit script from multiple changed methods, and to use an edit script for both location search and code transformation.

5.1 Motivating Example

This section uses a motivating example (see Figure 6.1) drawn from revisions to `org.eclipse.compare` on 2007-04-16 and 2007-04-30 to show SYDIT's work flow (see Figure 5.2) and compare it to LASE's work flow (see Figure 5.4). Figure 6.1 shows three methods with similar changes: `mA`, `mB`, and `mC`. The changes to method `mA` delete two print statements (line 3-4), insert a local variable declaration `next` for each enumerated element (line 6), and insert a type check for the element before it is processed (line 7).

Figure 5.2 shows SYDIT's work flow to automate systematic editing. In the figure, gray bars represent edit context, red bars represent deleted code, and blue bars represent inserted code. When a user shows SYDIT an exem-

A_{old} to A_{new}

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

B_{old} to B_{new}

```
1. public void updateActions () {
2.   Iterator iter = getActions().values().iterator();
3.   while(iter.hasNext()){
4. -   print(this.getReplacedText());
5. +   Object next = iter.next();
6. +   if (next instanceof MVAction){
7. -   MVAction action = (MVAction)iter.next();
8. +   MVAction action = (MVAction)next;
9.     if(action.isDependent())
10.      action.update();
11.+  }
12.+  if (next instanceof FRAction){
13.+    FRAction action = (FRAction)next;
14.+    if(action.isDependent())
15.+      action.update();
16.+  }
17.  }
18.  print(this.toString());
19.}
```

C_{old} to C_{new}

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

Figure 5.1: A systematic edit to three methods based on revisions to `org.eclipse.compare` on 2007-04-16 and 2007-04-30

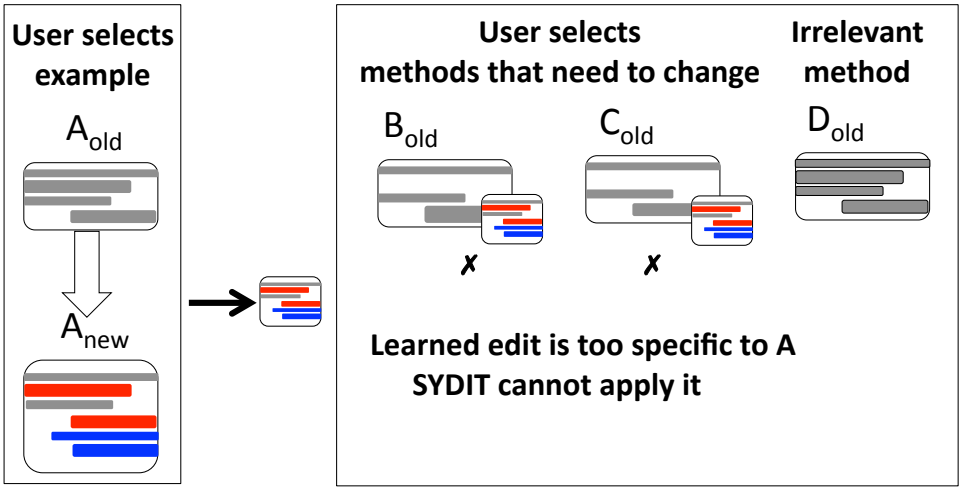

Figure 5.2: SYDIT learns an edit from one example. A developer must locate and specify the other methods to change.

```
1. … … method_declaration(… …){
2.    T$0 v$0 = v$1.m$0().m$1();
3.    DELETE: m$2(v$2.m$3());
4.    DELETE: m$2(v$2.m$4());

3.    while(v$0.m$5()){

4.       UPDATE: T$1 v$3 = (T$1)v$0.m$6();
5.           TO: T$2 v$4 = v$0.m$6();
6.       if(v$3.m$7()){
7.        … …
8.       }
9.       INSERT: if(v$4 instanceof T$1){
10.          INSERT: T$1 v$3 = (T$1)v$4;
11.          … …
12.              }
```

MOVE

Figure 5.3: Edit script from SYDIT abstracts all concrete identifiers. Gray marks edit context, red marks deletions, and blue marks additions.

plar changed method, e.g., $\texttt{mA}_o$ and $\texttt{mA}_n$, and all target methods to change systematically, e.g., mB and mC, SYDIT is expected to generate a general edit script, customize the script to each target method, and apply the result. For

this example, SYDIT manages to infer an edit script as shown in Figure 5.3. However, it cannot apply any edit to the target methods because the inferred script includes deleting two statements (lines 3-4), which operations are overly specific to `mA` and prevent the script from being applied to `mB` and `mC`.

When using the edit script inferred by SYDIT to find systematic edit locations, we still suffer from the *over specification* problem mentioned above since the edit script may fail to match methods it should have matched. On the other hand, we may also suffer from another problem—*over generalization* when the edit script succeeds to match irrelevant methods, such as `mD`, which it should have not matched. The reason for this is SYDIT's full identifier abstraction strategy makes the edit script too flexible, causing it to match types, methods, or variables in many unrelated methods, and consequently would incorrectly apply edit scripts to too many methods.

LASE seeks to generate an edit script that serves double duty, both finding edit locations and accurately transforming the code. It learns from two or more example edits to solve the problems of over generalization and over specification. Although developers may also want to directly create or modify a script, since they already write and edit code, we think providing multiple examples is a natural interface.

Figure 5.4 shows the work flow of LASE. The developer specifies two exemplar changed methods, `mA` and `mB`. LASE infers the edit script shown in Figure 5.5 from the examples. It uses the edit script to find matching locations, and applies customized script to each location. Using multiple examples requires new algorithms to identify common changes and context, and to abstract or omit differences. None of these algorithms are necessary when learning from a single example.

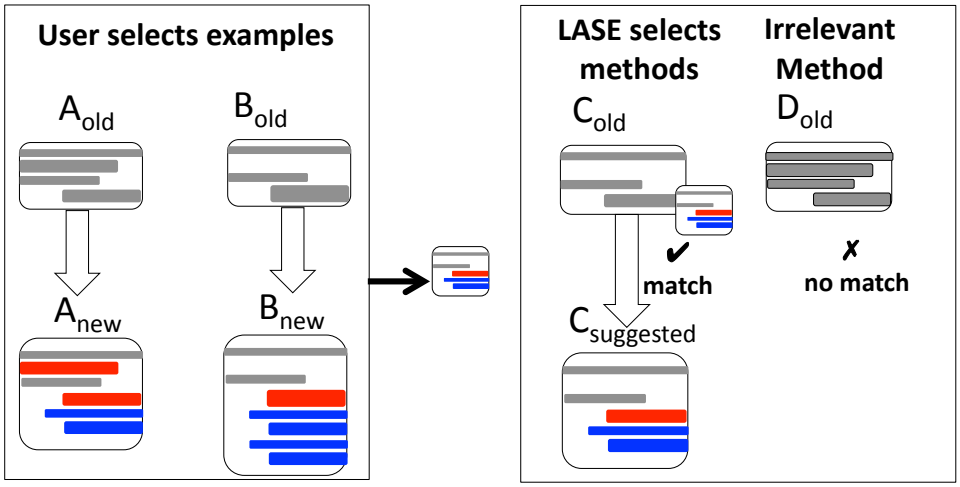

Figure 5.4: LASE learns an edit from two or more examples. LASE locates other methods to change.

```
1. … … method_declaration(… …){
2.    Iterator v$0 = u$0:FieldAccessOrMethodInvocation
                              .values().iterator();
3.    while(v$0.hasNext()){
4.       UPDATE: MVAction action = (MVAction)v$0.next();
5.            TO: Object next = v$0.next();
6.       if(action.m$0()){
7.        … …
8.       }
9.       INSERT: if(next instanceof MVAction){
10.           INSERT: MVAction action = (MVAction)next;
11.           … …
12.               }
```

MOVE

Figure 5.5: Edit script from LASE abstracts identifiers that differ in the examples and uses concrete identifiers for common ones. Gray marks edit context, red marks deletions, and blue marks additions.

LASE first finds the longest common edit operation subsequence among exemplar edits to filter out operations specific to only a single example. Notice that LASE omits the deleted `print` statements from `mA` because the edits are not common to `mA` and `mB`. LASE extracts the context for each common edit and then determines the largest common edit-relevant context. This algorithm combines clone detection, maximum common embedded subtree extraction on the Abstract Syntax Tree (AST), and dependence analysis. Finally, if type, method, and variable identifiers agree, LASE uses these concrete identifiers. Otherwise LASE abstracts the discrepant identifiers in both edit operations and context. For example in Figure 5.5, LASE uses `Iterator` because it is common to `mA` and `mB`. Since field access `fActions` in `mA` and method invocation `getActions()` in `mB` match but differ, LASE generalizes them to an abstract identifier `u$0:FieldAccessOrMethodInvocation`.

5.2 Approach

This section summarizes LASE's three phases and formalizes our terminology. Each subsection then describes one phase in detail.

Phase I: Generating an Edit Script. Generating an edit script from multiple examples has four steps.

1. *Generating Syntactic Edits.* For each exemplar changed method $\mathtt{m}_i \in M=\{\mathtt{m}_1, \mathtt{m}_2, \ldots, \mathtt{m}_n\}$, LASE compares the old and new versions of $\mathtt{m}_i$ and creates an edit: $E_i = [e_1, e_2, \ldots, e_k]$ where e_i is an insert, delete, move, or update operation of AST statements.

2. *Identifying Common Edit Operations.* LASE identifies the longest common edit operation subsequence E_c such that $\forall 1 \leq i \leq n$, $E_c \subseteq E_i$, and

E_c preserves the sequential order of operations in each E_i.

3. *Generalizing Identifiers in Edit Operations.* When a common edit operation $e \in E_c$ uses distinct type, method, and variable identifiers in different methods, LASE replaces the concrete identifiers with abstract ones, resulting in E. Otherwise, it uses the original concrete identifiers.

4. *Extracting Common Edit Context.* LASE finds the largest common context C relevant to E using code clone detection, maximum common embedded subtree extraction, and dependence analysis. LASE abstracts identifiers in the context C as well as the edits E.

The result of this process is a *partially abstract, context-aware edit script* Δ_P.

Phase II: Finding Edit Locations. LASE uses the edit script's context C to search for methods M_f that match C.

Phase III: Applying an Edit. For each $\mathtt{m}_f \in M_f$, LASE specializes Δ_P to $\mathtt{m}_f$ by mapping abstract identifiers and abstract edit positions in Δ_P to concrete ones in m_f, producing Δ_f. LASE applies this concrete edit script to $\mathtt{m}_f$ and suggests the resulting method $\mathtt{m}_f\mathtt{'}$ to the developer.

5.2.1 Phase I: Learning from Multiple Examples

This section explains how LASE generates a partially abstract, context-aware edit script from multiple exemplar edits.

5.2.1.1 Generating Syntactic Edits

For each exemplar changed method $\mathtt{m}_i \in M$, LASE exploits the syntactic program differencing algorithm used in SYDIT to create an AST edit script which may consist of statement-level AST node insertions, deletions, updates, and moves.

5.2.1.2 Identifying Common Edit Operations

LASE identifies common edit operations in $\{E_1, E_2, \ldots E_n\}$ by iteratively comparing the edits pairwise using a Longest Common Edit Operation Subsequence (LCEOS) algorithm, which is similar to the classic Longest Common Substring algorithm [43], as shown in Equation (5.1).

$$LCEOS(s(E_i, p), s(E_j, q)) =$$

$$\begin{cases} 0 & \text{if } p = 0 \text{ or } q = 0 \\ LCEOS(s(E_i, p-1), s(E_j, q-1)) + 1 & \text{if } equivalent(e_p, e_q) \\ max(LCEOS(s(E_i, p)), s(E_j, q-1)), LCEOS(s(E_i, p-1), s(E_j, q))) & \text{if } !equivalent(e_p, e_q) \end{cases} \tag{5.1}$$

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

We do not require exact equivalence between edit operations because systematic edits are not necessarily identical. We define the comparison function $equivalent(e_p, e_q)$ in two ways: $concreteMatch(e_i, e_j, t_s)$ and $abstractMatch$ (e_i, e_j). LASE first applies $concreteMatch(e_i, e_j, t_s)$ to compare e_i and e_j using their edit types and *labels*. Labels are string representations of AST nodes with identifiers and operators. If two operations have the same node type and their labels' bi-gram string similarity [1] is above the threshold t_s, the function returns true. We use $t_s = 0.6$ to include more matches. LASE also matches AST node types inexactly, tolerating mapping `return` statement to `expression` statement, and `while` to `for` to `do`. If LASE fails to find

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

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

5.2.1.3 Generalizing Identifiers in Edit Operations

LASE next generalizes identifiers as needed. When all edit operations agree on a concrete identifier, LASE uses the concrete identifier. If one or more edit operations use a different identifier, LASE generalizes the identifier. For example, Figure 6.1 shows e_A = **update(**`MVAction action = (MVAction) e.next()`**)** matches with e_B = **update(**`MVAction action = (MVAction) iter.next()`**)**. When LASE detects the discrepant variable names `e` vs. `iter`, it generalizes them by creating a fresh abstract identifier `v$0`, substituting it for the original identifiers, and creating e = **update(**`MVAction action = (MVAction)v$0.next()`**)**. LASE records the pairs (`e`, `v$0`), (`iter`, `v$0`) in a map. It then substitutes `v$0` for all instances of `e` in m_A and E_A, and all instances of `iter` in m_B and E_B to enforce a consistent naming for all edit operations and edit context. If some subsequent common edit operations are inconsistent with the current mappings, LASE omits them, resulting in a list of partially abstract edit operations E.

5.2.1.4 Extracting Common Edit Context

This section explains how LASE extracts the common edit context C for E from the exemplar methods with clone detection and then refines this context based on consistent name usage, AST subtree extraction, and program dependence analysis.

Finding Common Text with Clone Detection For all matched edits $\{E_1, E_2, \ldots, E_n\}$, LASE extracts relevant unchanged *context* code. LASE first finds common text by dividing the methods into three parts using their common edits as anchors. Given matching edits E_1 on AST nodes $n_1, n_2 \in m_1$ and E_2 on $n_1', n_2' \in m_2$, let n_1 precede n_2 in m_1. LASE divides the methods into code preceding n_1 and n_1', code between the two matching nodes, and code after them. LASE next compares each of these three segment pairs with a clone detector [47]. This step reveals all possible common *text* shared between each pair of methods. This common text over approximates context. The steps below refine the context based on name mapping, common embedded subtree extraction, and dependence analysis.

Generalizing identifiers Because clone detection uses text similarity, it does not guarantee that type, method, and variable identifiers in one method are mapped consistently in other methods. LASE collects all identifier mappings between the two methods' clone pairs. If there are conflicting mappings, LASE retains the context statements for the most frequent mappings and excludes any inconsistent statements from the context. If different concrete identifiers map consistently with each other, LASE generalizes them to a fresh

abstract identifier and substitutes it for the identifiers in all context statements and edit operations to create an abstract common context C_{abs}.

Extracting Common Subtree(s) with MCESE Because clone detection uses text matching, the AST structure of two nodes may not match, e.g., two matching nodes may have different parent nodes. To solve this problem, LASE uses an off-the-shelf Maximum Common Embedded Subtree Extraction (MCESE) algorithm [65] to find the largest common forest structure, as shown in Equation 5.2. This algorithm traverses each AST in pre-order, indexes nodes, and encodes the tree structure into a node sequence. By computing the longest common subsequence between the two sequences and reconstructing trees from the subsequence, LASE finds the largest common embedded subtree(s), C_{sub}. It then excludes all the other statements from the context.

$$
\begin{aligned}
&MCESE(s,t) \\
&= \begin{cases} 0 & \text{if } s \text{ or } t \text{ is empty} \\ max \begin{cases} MCESE(head(s), head(t)) & \text{if } equivalent(s[0], t[0]) \\ +MCESE(tail(s), tail(t)) + 1, & \\ MCESE(head(s)tail(s), t), & \\ MCESE(s, head(t)tail(t)) & otherwise \end{cases} \end{cases}
\end{aligned} \tag{5.2}
$$

Consider $\mathtt{m}_A$'s and $\mathtt{m}_B$'s AST in Figure 5.6. LASE traverses $\mathtt{m}_A$'s AST in pre-order, indexes nodes, and encodes the tree into node sequence s = [1, 2, -2, 3, -3, 4, -4, 5, 6, -6, 7, 8, 9, -9, -8, -7, -5, 10, -10, -1], where "−" marks finishing the traversal of current node. Indexes X and $-X$ mark the boundaries of the subtree rooted at X's node. Similarly, LASE creates sequence t = [1, 2, -2, 3, 4, -4, 5, -5, 6, 7, 8, -8, -7, -6, -3, 9, -9, -1] for $\mathtt{m}_B$. We then use Equation (5.2) to find the longest common subsequence between them, which corresponds to

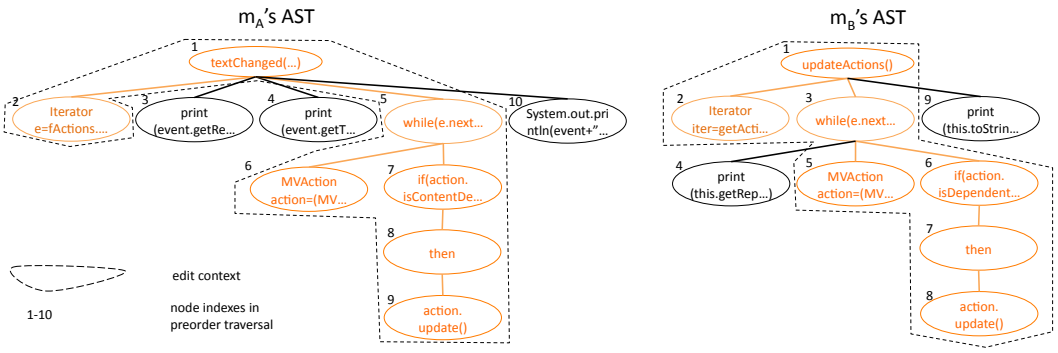

Figure 5.6: $\mathtt{m}_A$'s and $\mathtt{m}_B$'s AST

subsequence [1, 2, -2, 5, 6, -6, 7, 8, 9, -9, -8, -7, -5, -1] of s and [1, 2, -2, 3, 5, -5, 6, 7, 8, -8, -7, -6, -3, -1] of t. The reconstructed trees out of these sequences are colored with orange and circled with dash lines.

In the equation, $head(s)$ returns the sequence of nodes sub-rooting at $s[0]$ (excluding $s[0]$ and -$s[0]$), while $tail(s)$ returns the subsequence following -$s[0]$. For instance, given a sequence s = [1, 2, -2, -1, 3, -3], $head(s)$ = [2, -2], $tail(s)$ = [3, -3]. The function $equivalent(i, j)$ checks string equality between the two nodes' labels.

Refining Edit Context with Dependence Analysis The common text extracted between any two methods may include irrelevant code, i.e., code that does not have any control or data dependence relations with edited code. Blindly including them as edit context puts unnecessary constraints on potential edit locations, causing false negatives during edit location search. LASE thus further refines the extracted context based on control and data dependences.

LASE performs control and data dependence analysis and then determines direct and transitive dependences between edit operations and context

statements in each changed method. For each edit operation, LASE unions all the unchanged AST statements that are the source of dependences with the edit as relevant context. Finally, it intersects the identified edit-relevant context in each method to produce C_{dep}. If $C = C_{sub} \bigcap C_{dep}$ is not empty, LASE sets the context of E to C, omitting unrelated statements. If C_{dep} is empty, LASE sets $C = C_{sub}$, since matching an empty context is not useful for detecting edit locations. E and C define a *partially abstract, context-aware edit script*, Δ_P, where each edit operation in E is positioned with respect to context C.

5.2.2 Phase II: Finding Edit Locations

Given an edit script Δ_P, LASE searches for methods containing Δ_P's context C. Based on our assumption that methods containing similar edit contexts are more likely to experience similar changes, LASE suggests them as edit locations.

Because C is partially abstract, it contains both concrete and abstract type, method, and variable identifiers. When LASE matches C with a method `m`, it matches concrete identifiers exactly and abstract identifiers by identifier type (`T$`, `m$`, or `v$`) or AST node type. For instance, `Iterator` in C only matches `Iterator` in `m`. An abstract identifier, such as `v$0`, matches any variable, while `u$0_FieldAccessOrMethodInvocation` only matches `FieldAccess` or `MethodInvocation` AST nodes. LASE reuses the MCESE algorithm from Section 5.2.1.4 to find the maximum common context between C and `m`, but redefines the $equivalent(i, j)$ function to compare concrete identifiers based on string equality and abstract identifiers based on identifier type or AST node type. If each node of the common context C matches a node in

method m, LASE records identifier mappings between them and then suggests m as an edit location m_f.

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

5.2.3 Phase III: Applying the Edit

To apply the edit to a suggested location m_f, LASE must customize the edit Δ_P for m_f. For this process, we slightly modify the edit customization algorithm of SYDIT. The customization algorithm replaces all abstract identifiers in Δ_P with the corresponding concrete identifiers from m_f based on the identifier mappings established in Phase II. LASE retains all the concrete identifiers in Δ_P. Similar to SYDIT, LASE positions each edit operation concretely in the target method in terms of AST node positions. The result is Δ_f, which fully specifies each edit operation as an AST modification with concrete labels and node positions. LASE applies this customized, concrete edit script and suggests the resulting version to developers.

5.3 Evaluation

This section evaluates LASE's precision and recall when finding correct edit locations and its accuracy when applying edits. We use two oracle test suites. One test suite consists of multiple systematic edits that fix the same bug in multiple commits, drawn from two open-source programs, Eclipse `JDT` and Eclipse `SWT`. The other one contains 37 systematic edits from five representative medium or large Java open-source programs (`jEdit`, Eclipse `JDT`, Eclipse plugins of `compare`, `core.runtime`, and `debug`). We explore LASE's sen-

sitivity to (1) multiple examples vs. one example, (2) example choice, and (3) strategies for identifier abstraction and context. LASE matches context against *all* methods in the entire program reasonably quickly to search for edit locations, taking 28 seconds on average.

5.3.1 Precision, recall, and accuracy with an oracle data set

To measure precision, recall, and accuracy, we use an oracle test suite based on edits to Eclipse `JDT` and Eclipse `SWT` identified by work on supplementary bug fixes [80, 84]. They find bug fixes spanning multiple commits to understand characteristics of incomplete or incorrect bug fixes, using the bug ID and clone detection. The work illustrates that developers miss locations that they need to change when initially fixing a bug, further motivating our work. We select systematic edits from these programs. If a bug is fixed more than once and there exist clones of at least two lines in bug patches checked in at different times, we manually examine these methods for systematic changes. We find 2 systematic edits in Eclipse `JDT` and 22 systematic edits in Eclipse `SWT`, as shown in Table 5.1, where the first two rows are from `JDT`, while the rest are from `SWT`. The table groups the examples into two sets based on whether LASE refines the context with program dependence analysis or not (see Section 5.2.1.4). The first 17 edits have non-empty C_{dep}, and thus $C = C_{sub} \bigcap C_{dep}$. The last 7 edits have empty C_{dep} and thus $C = C_{sub}$.

We use these patches as an oracle test suite for correct systematic edits and test if LASE can produce the same results as the developers given the first two fixes in each set of systematic fixes. Since the developers may not be perfect, there may be incorrect edits or missing edits for which we cannot control. Indeed, we confirmed with developers that LASE found 9 methods in

Table 5.1: LASE's effectiveness on repetitive bug patches to Eclipse

Index	Bug (patches)		m_i	Edit Location Σ	✓	P %	R %	A %	Operations E	C	A_E %
1	73784	(1)	4	4	4	100	100	53	7	2	29
2	82429	(2)	16	13	12	92	75	81	9	9	100
3	114007	(3)	4	4	4	100	100	100	6	6	100
4	139329	(3)	6	2	2	100	33	74	6	3	50
5	142947	(6)	12	12	12	100	100	100	1	1	100
6	91937	(2)	3	3	3	100	100	95	5	3	60
7	103863	(5)	7	7	7	100	100	100	34	34	100
*8	129314	(3)	3	4	4	100	100	100	2	2	100
9	134091	(4)	4	4	4	100	100	73	24	24	100
10	139329	(3)	3	4	3	75	100	100	1	1	100
11	139329	(3)	3	3	3	100	100	88	12	12	100
12	142947	(6)	9	9	9	100	100	83	6	6	100
13	76182	(2)	6	6	6	100	100	90	6	6	100
14	77194	(3)	3	3	3	100	100	97	13	13	100
15	86079	(3)	3	3	3	100	100	100	25	25	100
*16	95409	(3)	7	9	9	100	100	78	4	4	100
17	97981	(2)	4	3	3	100	75	100	3	3	100
Average			**6**	**5**	**5**	**98**	**93**	**89**	**10**	**9**	**91**
18	74139	(3)	5	5	5	100	100	100	1	1	100
19	76391	(3)	6	3	3	100	50	100	3	3	100
20	89785	(3)	5	5	5	100	100	95	5	3	60
21	79107	(2)	3	2	2	100	67	92	4	4	100
22	86079	(4)	4	2	2	100	50	100	8	8	100
23	95116	(4)	5	4	4	100	80	100	3	3	100
*24	98198	(2)	9	15	15	100	100	95	3	3	100
Average			**5**	**5**	**4**	**100**	**78**	**97**	**4**	**4**	**94**
Total Average			**6**	**5**	**5**	**99**	**89**	**91**	**8**	**7**	**92**

* LASE suggests edits *missed* by developers.

3 fixes (starred in Table 5.1) and applied correct edits that they missed! When LASE produces the same results as developers do in later patches, it indicates that LASE will help programmers detect edit locations earlier, reduce errors of omission, and make systematic edits.

We give LASE as input two random changed methods in the first patch. If there is only one changed method in the first patch, we randomly select the second one from the next patch. LASE generates an edit script from these two examples, finds edit locations, customizes the edit for each location, and applies the customized edit to suggest a new version. Table 5.1 shows the results. The table lists the **Bug** identifier, the number of **patches**, and number of methods $\mathbf{m}_i$ that developers changed. For each **Edit Location**, we present Σ: the number of methods that LASE identifies as change locations; ✓: the number of methods correctly identified; precision **P%**: the percent of correctly identified edit locations compared to all found locations; recall **R%**: the percentage of correct locations out of all expected locations; and accuracy **A%**: the syntactic similarity between the tool-suggested version and the expected version, only for edited methods. The **Operation** columns present **E**: the number of edit operations shared among repetitive fixes for the same bug, i.e., operations we expect LASE to infer; **C**: the number of operations correctly inferred by LASE; and $\mathbf{A}_E$: the percentage of operations correctly inferred over expected operations.

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

7 edit locations, and correctly applies customized edits with 100% accuracy. In three edit cases (8, 16, and 24), LASE suggests 9 edits that developers *missed*. Note that the number of methods correctly identified for each is larger than the number of methods developers changed. We confirmed all these omission errors with the Eclipse developers and mark the cases with an asterisk in Table 5.1. These results indicate that LASE will help developers make systematic edits consistently and help reduce errors of omission.

LASE cannot guarantee 100% edit application accuracy for four reasons. First, the inferred edit is sometimes a subset of the exemplar edits and LASE cannot suggest edits specific to a single location. For instance in edit case 2, LASE infers all 9 edit operations shared among repetitive fixes for the same bug, but it misses some specific edits and does not achieve 100% accuracy. Second, abstract identifiers may not have corresponding concrete identifiers in the edit location. For example, if an abstract identifier is only used by inserted statements, LASE cannot decide how to concretize it. Third, based on string similarity, LASE's AST differencing algorithm cannot always infer edits operations correctly. For instance, if `trailingComments != null` is updated to `trailingPtr >= 0` in one method, and `rComments != null` is updated to `rPtr >=0` in another method, the inferred operation for the former is an update operation while the inferred operations for the later include an insert and a delete since the two strings are not similar enough for LASE to infer an update operation. When LASE compares an update operation to the insert and delete operations, the edit types do not match and it does not extract a common edit operation. Fourth, LASE's LCEOS algorithm cannot always find the best longest common edit operation subsequence between two sequences because it does not enumerate or compare all possible longest common sub-

Table 5.2: LASE's effectiveness when learning from multiple examples

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

sequences to choose the best one. Although each of these problems occurred, none occurs frequently.

The number of exemplar edits influences effectiveness. To determine how sensitive LASE is to the number and choice of exemplar edits, we randomly pick 6 cases in the oracle data set and enumerate subsets of exemplar edits, e.g., all pairs of two exemplar methods. We evaluate the precision, recall, and accuracy for each choice of exemplars and calculate the average for each cardinality to determine how sensitive LASE is to the choice and number of exemplar edits.

Table 5.2 shows that precision **P** does not change as a function of the number of exemplar edits for these examples, except for case 12, where two exemplars are the most accurate. Recall **R** is more sensitive to the choice and

number of exemplar edits, increasing as a function of exemplars. The more exemplar edits provided, the less common context is likely to be shared among them, and the easier it is to match. However, the context will still be specific, resulting in high precision. Precision can go down when more diverse examples are given, but this case (case 12) only occurred once in these tests.

In theory, Accuracy **A** can vary inconsistently with the number of exemplar edits, because it strictly depends on the similarity between edits. For instance, when exemplar edits are diverse, LASE extracts fewer common edit operations, which lowers accuracy. When exemplar edits are similar, adding exemplar methods may not decrease the number of common edit operations, but may induce more identifier abstraction and result in a more flexible edit script, which increases accuracy.

5.3.2 Sensitivity of Edit Scripts

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

We measure precision and recall of edit location suggestion and accuracy of edit application on the SYDIT test suite. This suite contains 56 pairs of exemplar changed methods. We remove the simple cases, e.g., edits on initially empty methods or only one statement, resulting in 37 pairs. For each pair,

we extend the oracle set of exemplar edits as follows. We first apply LASE to infer the systematic edit demonstrated by both methods and search for edit locations in the program's original version. Then we manually examine all found locations. If a location is indeed edited similarly in the next version but not in the known pairs, we include it in the oracle set.

Table 5.3 shows the results. On average, learning from one example has lower precision and recall when looking for edit locations as compared to learning from two examples, but has higher accuracy when suggesting edits for correctly identified locations. Several reasons explain these results.

- Inferring a common context from two examples results in a mix of concrete and abstract identifiers. Searching with a partially abstract context is more precise than a fully abstract context, which matches more methods. The partially abstract context recalls more than a concrete context does, which matches fewer methods.

- Using two examples reduces the edit to a common subset, so the derived edit is likely to be less accurate for any one target location, since it may lack some edit operations.

- LASE includes all nodes transitively depended on by any edited node in the inferred context, deriving a more precise context as compared to SYDIT's context, which is based on direct dependence relations.

- LASE matches context differently than SYDIT.

Table 5.4 shows the average sensitivity of LASE to the abstraction and context algorithms by comparing their average precision, recall, and accuracy on all 37 systematic edits.

Table 5.3: Learning from one example versus multiple examples

ID	m_i	Two Examples Σ	✓	P %	R %	A %	One Example Σ	✓	P %	R %	A %
1	5	6	5	83	100	100	10	5	50	100	100
2	2	3	2	67	100	80	7	2	29	100	100
3	5	7	5	71	100	100	277	1	0	25	100
4	2	3	2	67	100	96	596	2	0	100	97
5	5	6	5	83	100	100	5	3	60	60	100
6	2	73	2	3	100	100	3354	2	0	100	100
Average	**4**	**18**	**4**	**62**	**100**	**94**	**708**	**3**	**23**	**81**	**100**
7	2	2	2	100	100	92	24	2	8	100	83
8	2	2	2	100	100	100	76	2	3	100	100
9	3	3	3	100	100	100	4	2	50	67	100
10	2	2	2	100	100	100	8	2	25	100	100
11	2	2	2	100	100	100	3	2	67	100	100
12	2	2	2	100	100	100	2	1	50	50	100
13	2	2	2	100	100	96	5	1	20	50	100
14	2	2	2	100	100	99	3	2	67	100	100
Average	**2**	**2**	**2**	**100**	**100**	**98**	**16**	**2**	**36**	**83**	**98**
15	2	2	2	100	100	100	2	2	100	100	100
16	2	2	2	100	100	100	2	2	100	100	100
17	2	2	2	100	100	100	1	1	100	50	100
18	2	2	2	100	100	96	1	1	100	50	100
19	2	2	2	100	100	100	2	2	100	100	100
20	2	2	2	100	100	100	2	2	100	100	100
21	2	2	2	100	100	100	2	2	100	100	100
22	2	2	2	100	100	75	1	1	100	50	100
23	4	4	4	100	100	100	4	4	100	100	100
24	2	2	2	100	100	100	2	2	100	100	100
25	2	2	2	100	100	86	1	1	100	50	100
26	2	2	2	100	100	87	1	1	100	50	100
27	5	5	5	100	100	100	5	5	100	100	100
28	2	2	2	100	100	100	1	1	100	50	100
29	2	2	2	100	100	74	2	2	100	100	99
30	2	2	2	100	100	88	1	1	100	50	100
31	2	2	2	100	100	100	2	2	100	100	100
32	2	2	2	100	100	100	2	2	100	100	100
33	2	2	2	100	100	84	1	1	100	50	100
34	6	6	6	100	100	100	6	6	100	100	100
35	6	6	6	100	100	100	6	6	100	100	100
36	6	6	6	100	100	100	6	6	100	100	100
37	6	6	6	100	100	100	6	6	100	100	100
Average	**3**	**3**	**3**	**100**	**100**	**95**	**3**	**3**	**100**	**83**	**100**

Table 5.4: Comparison between LASE and its variants

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

LASE $_{AbsAll}$ differs from LASE by abstracting all identifiers instead of only abstracting identifiers when necessary. Therefore, LASE $_{AbsAll}$'s inferred context is more general than context inferred by LASE and it matches more methods, causing more false positives and lower precision.

LASE $_{SigCon}$ learns from a single example and uses all concrete identifiers which makes LASE $_{SigCon}$'s inferred edit very specific to the example. Consequently, LASE $_{SigCon}$'s derived context is too specific to find all edit locations. Its average recall is just 60% with many false negatives. In many cases, LASE $_{SigCon}$'s context can only find the method from which it is inferred and cannot detect any other edit location. In contrast, LASE has 100% recall on these examples.

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

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

MCESE is much better at identifying the correct locations.

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

5.4 Summary

LASE helps developers perform systematic editing in multiples locations efficiently and correctly. Similar to SYDIT, LASE does not guarantee the correctness of transformed code partially because (1) some systematic edits are inadequate to produce correct programs, and (2) LASE does not synthesize identifiers to concretize newly introduced identifiers. The systematic edit LASE produces for multiple similarly changed methods captures the shared commonality of edit and relevant context. As a result, the common edit may contain a subset of edit operations needed for all methods, missing edits for individual locations. The common relevant context may serve as a necessary condition instead of a sufficient condition for edit application, and thus requires extra programmer effort to make the generated program correct. LASE focuses on how to infer, locate, and apply systematic edits based on multiple exemplar edits. It does not explore how to check the correctness of a systematically edited program or rectify an incorrect program.

When making similar edits to multiple locations, programmers may need help in both finding edit locations and applying the edits. However,

providing a single exemplar edit is not enough for any tool to assist in both tasks because one exemplar edit only tells what is changed, without indicating which parts to generalize and how to generalize them. Programmers need to provide at least two examples to demonstrate the generalization, which is critical to finding edit locations. Although developers are required to provide more than one exemplar edit for LASE, they benefit by receiving more accurate suggestions on edit locations and edit applications.

Systematic edits on multiple locations may indicate a good opportunity for clone removal refactoring so that duplicated code is removed and redundant code changes are eliminated. Blindly automating systematic edits without the awareness of potential refactoring opportunities can produce code clones, and compromise software maintainability, readability, and correctness. The next chapter explores the relationship between systematic editing and refactoring.

Chapter 6

Refactoring Systematically Edited Code

When developers make similar changes to multiple locations, systematic editing tools automate the process to reduce the programming burden. On the other hand, similar changes may indicate that developers should instead refactor code to eliminate redundancy. Automating the editing process may encourage developers to duplicate code or maintain duplicated code. If programmers should always refractor, then systematic editing tools may be encouraging poor practices. To examine whether automated refactoring can obviate systematic edits, we design and implement a new automated refactoring approach Rase, which takes as input two or more methods with systematic edits to scope target code, and then performs *clone removal*.

Rase combines *extract method* (pg. 110 in [28]), *add parameter* (pg. 275 in [28]), *introduce exit label*, *parameterize type*, *form template method* (pg. 345 in [28]), and *introduce return object* refactorings to extract and remove similar code. It creates an *abstract refactoring template* that abstracts differences in identifiers of types, methods, variables, and expressions from multiple locations. Based on this template, as well as control and data flow, Rase creates new types and methods; inserts and assigns return objects and exit labels; adds parameters to the new extracted method; and introduces customized calls to it.

To our knowledge, Rase implements state-of-the-art refactoring with

respect to its capability to factor and generalize code. Existing clone removal refactoring tools [7, 41, 46, 57, 89] only implement some, but not all of the refactoring techniques in Rase. Furthermore, prior work that does study clone removal [5, 11, 32, 48, 52] *did not* actually construct an automated refactoring tool to investigate the refactoring of systematically changed code. The lack of automation in prior work introduces the possibility of subjectivity bias. By automating refactoring, our study substantially improves on prior methodology for determining the feasibility of refactoring.

We evaluate Rase on 56 real-world systematically edited method pairs (n=2) from prior work [70, 71] and 30 systematically edited method groups (n≥3) drawn from two open source projects. Rase automatically refactors 30 of 56 method pairs (54%) and 20 of 30 (67%) method groups when scoped with systematic edits. Rase applies sophisticated refactorings with all six techniques and in multiple different combinations of up to four techniques at once. On average, Rase automatically applied 41 lines of edits in our examples, ranging from 6 to 285, with modest code size increases of up to 18 lines of code, and reductions of up to 149 lines. Not surprisingly, Rase is most effective at reducing code size for multiple methods. Manual transformation to attain the same results would be as cumbersome as inserting or deleting 285 and 211 lines of code, which reduces the resulting code by 47 and 149 lines—all of which Rase automates for developers. These results add to the evidence that removing common code with variations is challenging in practice and needs automated tool support.

We compare Rase scoped by systematic edits to Rase scoped by methods: scoping with systematic edits improves the feasibility of automatic clone removal compared to method-level scoping: Rase scoped by systematic edits

for method pairs refactors 54% compared to 34% for method-level only refactoring, and for method groups, increases opportunities for refactoring to 67% compared to 30% for method-level only refactoring. Systematic edits thus are a good clue for refactoring, rather than being obviated by method refactoring. However, RASE cannot automate refactoring in 46% of pairs and 33% of groups mainly because of language limitations, semantic constraints, and lack of common code. We manually check software version history after systematic edits and find that in many cases, systematically edited methods are unrefactored. They either co-evolve, diverge, or stay unchanged. Our tool evaluation and repository observation indicate that developers need to perform both systematic editing and refactoring during software evolution, and that automated tools can help them for both tasks.

6.1 Motivating Example

This section overviews our approach with an example based on revisions to `org.eclipse.compare.CompareEditorInput` on 2006-11-20 and 2006-12-18. In Figure 6.1, the two methods perform very similar input processing and experience similar edits: adding a variable declaration and updating statements. However, the changes involve using different type, method, variable names: `IActionBars` vs. `ISLocator`; `getActionBars` vs. `getServiceLocator`; `findActionBars` vs. `findSite`; `offset` vs. `offset2`; and `actionBars` vs. `sLocator`.

Given two changed methods, RASE invokes LASE to create an abstract edit script, as shown in Figure 6.2. With the edit script, RASE identifies edited statements related to the systematic changes in the new version of each method. For Figure 6.1, RASE identifies lines 9-10 and 14 in the

```
1. public class CompareEditorInput {
2.   private ICompareContainer fContainer;
3.   private boolean fContainerProvided;
4.   private Splitter fComposite;
5.   public IActionBars getActionBars (int offset) {
6.     if (offset == -1)
7.       return null;
8. -   if (fContainer == null) {
9. +   IActionBars actionBars = fContainer.getActionBars();
10.+   if (actionBars==null&&offset!=0&&!fContainerProvided){
11.      return Utilities.findActionBars(fComposite, offset);
12.    }
13.-   return fContainer.getActionBars();
14.+   return actionBars;
15.  }
16.  public ISLocator getServiceLocator (int offset2) {
17.    if (offset2 > fComposite.getSize())
18.      return null;
19.-   if (fContainer == null) {
20.+   ISLocator sLocator = fContainer.getServiceLocator();
21.+   if(sLocator == null&&offset2!=0&&!fContainerProvided){
22.      return Utilities.findSite(fComposite, offset2);
23.    }
24.-   return fContainer.getServiceLocator();
25.+   return sLocator;
26.  }
27.}
```

Figure 6.1: An example of systematic changes based on revisions to `org.eclipse.compare.CompareEditorInput` on 2006-11-20 and 2006-12-18

`getActionBars` method and lines 20-21 and 25 in the `getServiceLocator`. RASE uses the ranges of edits to scope its automated factorization and generalization, extracting the maximum common contiguous code which encompasses all systematically edited statements. If similar edits are surrounded by cloned statements, RASE expands the refactoring scope to the entire method. In this way, we remove cloned methods by keeping only one copy of the common code. If similar edits are not surrounded by cloned statements, RASE starts from the common edited code, expands refactoring scope, and extracts as much commonality as it can. In Figure 6.1, lines 9-12, 14, 20-23, and 25 are marked for factorization and generalization. Note that RASE includes the unchanged

```
1. … …method_declaration(… …) {
2.     … …
3.     INSERT: T$0 v$0 = fContainer.m$0();
4.     UPDATE: if (fContainer == null) {
5.         TO: if (v$0==null && v$1!=0 && !fContainerProvided){
6.       … …
7.     }
8.     UPDATE: return fContainer.m$0();
9.         TO: return v$0;
10.}
```

Figure 6.2: Abstract edit script inferred by LASE

lines 11-12, 22-23 in order to extract syntactically valid `if` statements and statement groups.

Next, RASE creates an abstract refactoring template for extracted code by establishing matches between statements, expressions, and identifiers among multiple methods. This step matches statements based on their AST types, such as `ReturnStatement` or `ExpressionStatement`. It then matches expressions and identifiers used in matched statements. RASE uses the original code as is, if the matched statements are identical (e.g., `fContainer` vs. `fContainer`), and otherwise abstracts expressions or identifiers in the matched statements (e.g., `offset` vs. `offset2`).

```
1.   T$0 v$0 = fContainer.m$0();
2.   if (v$0==null && v$1!=0 && !fContainerProvided) {
3.     return Utilities.m$1(fComposite, v$1);
4.   }
5.   return v$0;
```

Figure 6.3: Abstract refactoring template of common code created by RASE

Figure 6.3 shows the resulting abstract template. Based on the template, RASE determines the extracted method's arguments and return variable, and performs additional refactoring transformations. Figure 6.4 shows

Newly created classes and methods through generalization

```
1. public abstract class TemplateClass<T0> {
2.   public T0 extractMethod(int v1, Splitter fComposite,
3.       ICompareContainer fContainer, boolean fContainerProvided){
4.     T0 v0 = m0(fContainer);
5.     if (v0 == null && v1 != 0 && !fContainerProvided) {
6.       return m1(fComposite, v1);
7.     }
8.     return v0;
9.   }
10.  public abstract T0 m0(ICompareContainer fContainer);
11.  public abstract T0 m1(Splitter fComposite, int v1);
12.}
13.public class ConcreteTemplateClass0 extends
14.     TemplateClass<IActionBars> {
15.  public IActionBars m0(ICompareContainer fContainer) {
16.    return fContainer.getActionBars();
17.  }
18.  public IActionBars m1(Splitter fComposite, int v1) {
19.    return Utilities.findActionBars(fComposite, v1);
20.  }
21.}
22.public class ConcreteClass1 extends TemplateClass<ISLocator> {
23.  public ISLocator m0(ICompareContainer fContainer) {
24.    return fContainer.getServiceLocator();
25.  }
26.  public ISLocator m1(Splitter fComposite, int v1) {
27.    return Utilities.findSite(fComposite, v1);
28.  }
29.}
```

Modifications to the original methods

```
1. public class CompareEditorInput {
2.   private ICompareContainer fContainer;
3.   private boolean fContainerProvided;
4.   private Splitter fComposite;
5.   public IActionBars getActionBars (int offset) {
6.     return new ConcreteTemplateClass0().extractMethod(offset,
7.          fComposite, fContainer, fContainerProvided);
8.   }
9.   public ISLocator getServiceLocator (int offset2) {
10.    return new ConcreteTemplateClass1().extractMethod(offset2,
11.         fComposite, fContainer, fContainerProvided);
12.  }
13.}
```

Figure 6.4: Code refactoring based on inferred systematic program transformation

the refactored version for our example. RASE performs a *parameterize type* refactoring because the type variation `T$0` must be handled to work correctly for the different type variables. The method variations `m$0` and `m$1` require a *form template method* refactoring to invoke the correct methods depending on the callers. The variation in the use of a variable name `v$1` requires the corresponding variable to be passed as a parameter to the extracted method. The variable wildcard `v$0` does not need such processing, because the variable is locally defined and used, and thus invisible to the extracted method's callers. RASE performs a static analysis to differentiate these cases.

6.2 Approach

RASE takes systematic changes or methods as input and extracts common code among the multiple methods. It works in two phases. Phase 1 determines the scope of refactoring for systematic edits or methods, analyzes variations, and outputs an *abstract refactoring template*. This template encodes the scope and content of code to extract and serves as a basis for determining concrete transformations. Phase 2 constructs an executable refactoring plan by handling type, method, variable, and expression variations and by analyzing control flow, data flow, and class hierarchy of original methods. RASE currently uses a combination of six different refactoring operations.

6.2.1 Abstract Template Creation

We use LASE to create an abstract edit script that describes the input systematic changes [70]. LASE represents the difference between before- and after- versions with AST node inserts, deletes, updates, and moves. LASE identifies the largest common subsequence edits and then extracts the common

context with control and data dependence analysis. It produces an abstract edit script, which includes both a code pattern containing abstract syntax subtrees and a list of tree edit operations to apply. In the abstract edit script, LASE uses the original concrete code when the different methods use the same concrete identifiers. When they use different identifiers, LASE abstracts the identifiers. Figure 6.2 shows an exemplar edit script.

RASE takes the output of LASE as an input and marks all edited statements. It marks the maximum common contiguous region of AST statements that enclose all edits. RASE requires

- The AST statements are contiguous.
- The AST statements either form a subtree or contiguous subtrees under the same parent node, such that there is only one entry to the region and the code snippet can be extracted as a method.

Starting with all edited statements in the new version, RASE first identifies all subtrees rooted at the edited statements. It then merges subtrees until there is a single subtree left or a sequence of adjacent subtrees under the same parent node. The merging algorithm picks two subtrees, T_1 and T_2, with the longest paths from the tree root, identifies the lowest common ancestor R between them, and substitutes R's subtree for T_1 and T_2 in the identified subtree set. RASE may include some unchanged code into the refactoring scope to extract syntactically valid statements. If there are no edited or added statements in the new methods, only deleted code, then we do not proceed with code extraction.

RASE then tries to create an abstract template to guide further refactoring. To successfully create such a template, RASE requires that (A) the

number of marked statements in the target methods is the same, and (B) the statements are syntactically similar between methods, although the statements may differ in their use of types, method invocations, variables, and expressions. Requirement (A) guarantees that the template reflects the code skeleton of marked statements in every method. Requirement (B) guarantees that we extract syntactically similar code.

RASE abstracts away the use of different type names, method invocations, variable names, and expressions by using wildcards `T$`, `m$`, `v$` and `u$` respectively. It attempts to establish mapping between each concrete identifier and the abstract version, making sure all methods consistently use and define these identifiers. If not, RASE does not refactor them. This analysis checks for semantic equivalence between the methods. RASE then uses the resulting **abstract template**, control and data flow analysis, and identifier maps to determine how to factorize and generalize the code, as described in the next section.

6.2.2 Clone Removal Refactoring

RASE next plans concrete refactoring transformations and then transforms code accordingly. RASE implements the following six types of refactoring operations:

- **extract method** extracts common code into a method.
- **add parameter** handles variations in different uses of variables and expressions.
- **parameterize type** handles variations in used types.
- **form template method** handles variations in method calls.

- **introduce return object** handles multiple output variables of extracted code.
- **introduce exit label** preserves control flow from the original code

Type Variations Given a type wildcard (`T$`) in the abstract template, RASE applies a **parameterize type** refactoring. It declares a generic type for the newly created class and modifies each original location to call the extracted method with type parameters. We define this new term because Fowler's catalog does not include it and current refactoring engines, such as Eclipse, do not support it. Figure 6.5 shows an example. When the target code differs in terms of type identifiers, RASE adds explicit type parameters to the new extracted method. The applicability of this refactoring is affected by language support for generic types. In our implementation for Java, the refactoring is not applicable when any parameterized type creates an instance by calling its constructors (e.g., `new T()`), to perform an `instanceof` check (e.g., `v instanceof T`) or gets the type literal (e.g., `T.class`), because Java does not support these cases. Even if developers may handle such cases manually with smart tricks, the resulting refactored code would have poor readability.

Method Call Variations Given a method wildcard (`m$`) in the abstract template, RASE applies the **form template method** (pg. 345 in [28]) refactoring. It creates uniform APIs that encapsulate the variations and changes in the extracted method to invoke these APIs instead. Figure 6.6 shows an example. RASE declares an abstract class which contains the extracted method and a sequence of abstract methods. Each abstract method corresponds to a

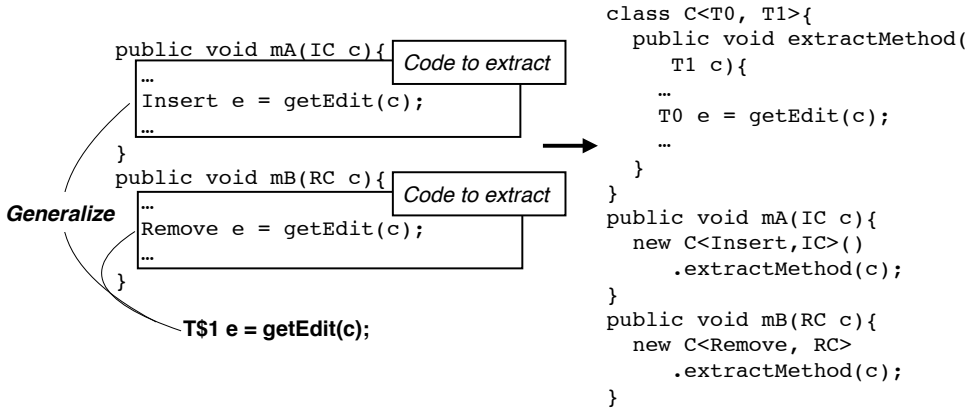

Figure 6.5: Parameterize type refactoring

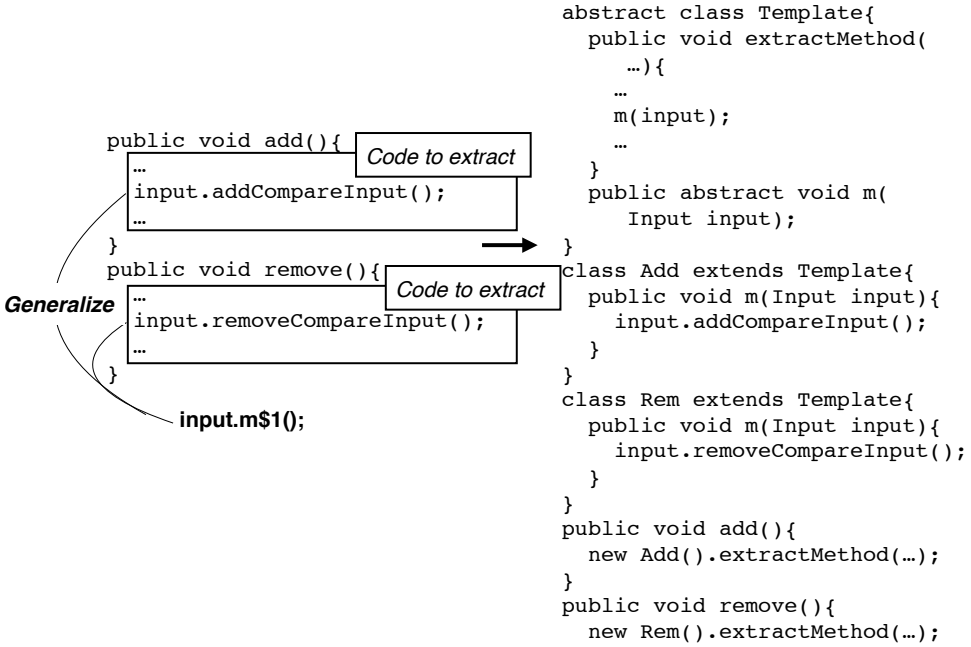

Figure 6.6: Form template method refactoring

method wildcard. For each original location, the refactoring declares a concrete class extending the abstract class so that all abstract methods are implemented to call the correct corresponding concrete methods. Each original location is modified to invoke the extracted method with the corresponding concrete class.

When a method wildcard represents a non-static method and is invoked via an object (e.g., `input.m$1()`), the corresponding method is declared to place the receiver object (e.g., `input`) as an argument (e.g., `m1(Input input)`). Then the actual method (e.g., `input.addCompareInput()` or `input.removeCompareInput()`) is invoked correctly inside each newly defined method. If any of the variant methods does not have a modifier `public` in its method declaration, the refactoring is not applied, because the method is not accessible by newly defined methods in the template class. If variant methods have different numbers of parameters, e.g., `foo(int offset)` vs. `bar(Object obj, boolean flag)`, the refactoring is not applied. Although it is possible to create a long method signature by merging different input signatures, we believe the resulting code is ugly and too hard to read.

Variable and Expression Variations Given variations in variable names and expressions, RASE uses an **add parameter** (pg. 275 in [28]) refactoring. We use data dependence analysis to identify variables which have local uses but no local definitions in the extracted code. We consider variable wildcards (`v$`) as candidates for input arguments of the extracted method. For each variable wildcard, we check whether it is purely local to the extracted code, meaning that it is declared, defined, and used only in the extracted code. If so, there is no need to promote the variable as an input parameter, since it is

invisible to caller methods. Assigning it a concrete identifier is enough. For example, RASE promotes the variable `v$1` shown in the template in Figure 6.3 to an input parameter but does not promote the variable `v$0` because it is defined and used only locally.

We consider expression wildcards (`u$`) as candidates for input arguments of the extracted method. Since the wildcards map to different AST node type expressions in different methods, each caller can pass appropriate expressions as arguments. For instance, if `u$` is mapped to `getConfiguration()` in one method, but is mapped to `fCompareEditorInput.getConfiguration()` in another method, RASE compares the types of both expressions. If the types are the same, it declares an input parameter with the common type. If the types are different, RASE records the type mapping and later applies **parameterize type** refactoring to accommodate the variation. Compared with prior work [7, 46], which solves expression variations by declaring new methods, our approach creates cleaner code by avoiding extra method declarations and invocations.

Return Value We use data dependence analysis to check which variables have local definitions and external uses. We convert them to output variables of the extracted method. When there is more than one such variable, we apply **introduce return object** refactoring. As shown in Figure 6.7, we encapsulate all return values into one object and insert code at each call site to read appropriate fields of the returned object. Fowler's catalog does not include this refactoring and current refactoring engines do not support it.

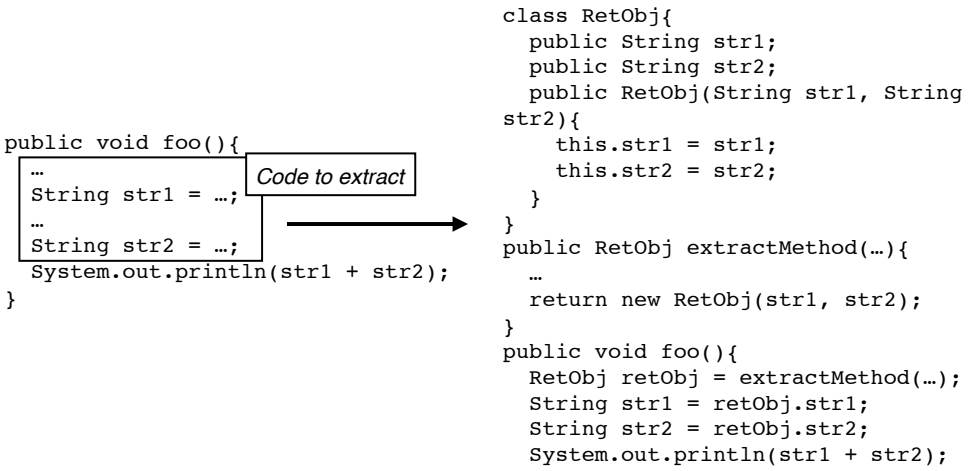

Figure 6.7: Introduce return object refactoring

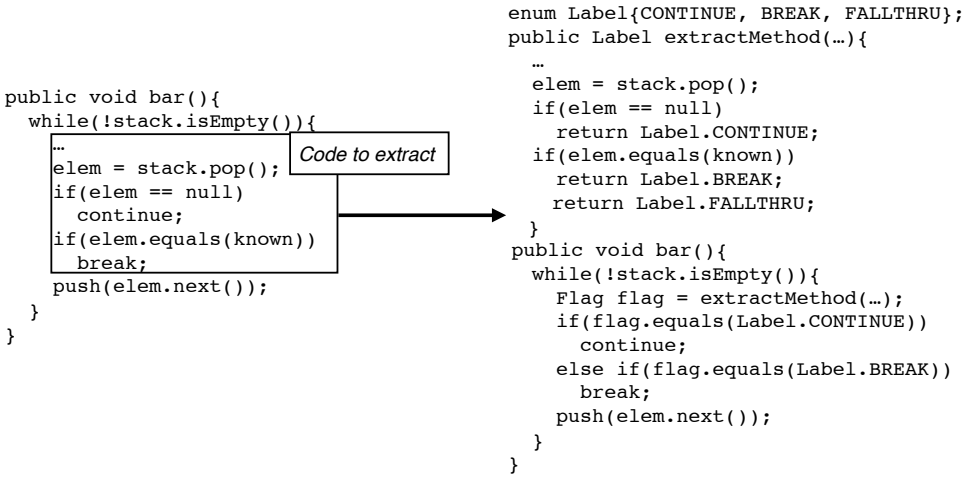

Figure 6.8: Introduce exit label refactoring

Control Flow We use control flow analysis to determine the statements that exit the code in addition to the fall-through exit in the extracted code, such as a `return`, `break`, or `continue`. These non-local jump nodes either terminate execution or jump execution from one location to another. Naively putting them in the extracted method may cause compiler errors or incorrect control flow. Rase applies **introduce exit label** refactoring to correctly implement non-local jumps. It replaces non-local jumps with exit label return statements and modifies each original location to interpret the returned labels. Fowler's catalog does not include this refactoring nor is it implemented in current refactoring engines, such as Eclipse. We borrow the approach from Komondoor and Horwitz's prior work on automated procedure extraction [55]. Figure 6.8 shows an example with multiple exits. Rase replaces a non-local jump statement with a return statement to terminate the execution of the current method and adds a return label indicating the exit type. Rase inserts code at each call site to handle non-local jumps correctly.

Placing Extracted Code We use class hierarchy analysis to discover the relation between classes declaring the originally edited methods. The relationships help Rase decide where to put the extracted method, and which input parameters or output variables to add. For instance, if the systematically changed methods are in the same class and when no method or type wildcards is in the abstract template, Rase places the extracted method in the same class. If the methods are in sibling classes extending the same super class, Rase puts the extracted method into their common super class. If the methods are in classes which do not have any type hierarchy relation, Rase must put the extracted method into a newly declared class. All fields that the extracted code reads from or writes to should be passed as input parameters and

output variables separately, since they may not be accessible to the extracted method defined by the newly defined class. RASE checks that all methods invoked by the extracted method are declared as `public` for correctness.

6.3 Evaluation

This section evaluates RASE with systematic editing tasks. It explores if automated refactoring eliminates the need for systematic editing; if systematic editing guides refactoring better or worse than methods alone. It also takes a first look at whether automated refactoring is desirable when it is feasible.

Our data set consists of 56 similarly changed method pairs and 30 similarly changed method groups. These real-world systematic editing tasks are drawn from version histories of `JEdit`, Eclipse `compare`, `jdt.core`, `core.runtime`, `debug`, `JFreeChart` and `elasticsearch`. The method pairs are drawn from prior evaluation of systematic editing [70, 71]. Each method pair have at least 40% syntactic similarity and share at least one common AST edit operation. Each method group contains at least three similarly changed methods.

We use four variants of refactoring for our evaluation. The default RASE refactors as much code as possible given a systematic edit. RASE$_{min}$ chooses the smallest amount of code that includes the systematic edit. RASE$_{MA}$ refactors the entire method after the edit and RASE$_{MB}$ refactors the entire method before the edit. We apply RASE and its variants to the test suites. For each refactoring task, we report if refactoring is automated. If so, we report the applied refactoring types, the number of edit operations, and the code size change. Table 6.1 and Table 6.2 present the results.

In the tables, each method pair has a unique identifier **ID**. The refactoring characterization includes edit operations (**edits**), refactoring **types**, and resulting code size change (Δ**code**). RASE applies the following six refactoring types: E: extract method, R: introduce return object, L: introduce exit label, T: parameterize type, F: form template method, and A: add parameter. N/A means RASE could not refactor. We omit pairs with no refactoring in any configuration. In Δ **code size** column, a positive number means refactoring increases the code size and a negative number means code size decreases.

6.3.1 Method Pairs

By extracting the maximum common code enclosing systematic edits, RASE (columns 2-4 on the left) automatically refactors 30 out of 56 cases. By comparing RASE against RASE$_{MA}$ (columns 5-7 in the middle) and RASE$_{MB}$ (columns 8-10 on the right), we find scoping refactoring based on systematic edits increases refactoring opportunities. RASE$_{MA}$ automates refactoring for only 19 cases, all of which are refactored by RASE. For these cases, RASE$_{MA}$ refactors the entire method because the changed methods are clones of each other. RASE refactors 11 more cases than RASE$_{MA}$ by limiting refactoring scope with systematic edits, when the methods are less similar to each other.

RASE$_{MB}$ refactors 18 cases, all of which are also refactored by RASE and RASE$_{MA}$. In these cases, the old versions of the method pair are clones and experience similar changes, producing clones afterward. Case 2 is not handled by RASE$_{MB}$, because the original version has no statements in the method and thus no clones can be extracted. These comparisons imply that systematic edits better scope refactoring and increase refactoring opportunities compared to applying clone removal to the entire methods either before or after

Table 6.1: Method pairs: Clone removal refactorings

ID	Rase edits	Rase types	Rase Δcode	Rase $_{min}$ edits	Rase $_{min}$ types	Rase $_{min}$ Δcode	Rase $_{MA}$ edits	Rase $_{MA}$ types	Rase $_{MA}$ Δcode	Rase $_{MB}$ edits	Rase $_{MB}$ types	Rase $_{MB}$ Δcode
2	15	E, A	-1	15	E, A	-1	15	E, A	-1		N/A	
4	6	E, A	2	6	E, A	2	6	E, A	2	6	E	2
6	14	E, F	10	14	E, F	10	14	E, F	10	31	E, F	11
9	77	E, R	-7	61	E	-15		N/A			N/A	
10	24	E	-4	20	E, L	8	24	E	-4	15	E	-1
11	20	E, F	8	20	E, F	8	20	E, F	8	14	E, F	10
12	31	E, F	11	31	E, F	11	31	E, F	11	14	E, F	10
13	38	E, F	2	32	E, F	4	38	E, F	2	29	E, F	5
18	42	E	-10	7	E	3		N/A			N/A	
19	61	E	-15	21	E, R	13	61	E	-15	61	E	-15
22	285	E, F	-47	285	E, F	-47	285	E, F	-47	288	E, F	-48
29	56	E, L, R	4	45	E, L, R	9		N/A			N/A	
32	9	E, A	1	6	E	2		N/A			N/A	
34	24	E, A	-4	24	E, A	-4		N/A			N/A	
35	9	E	1	9	E	1		N/A			N/A	
36	36	E, A	-8	36	E, A	-8		N/A			N/A	
38	16	E	0	12	E	0		N/A			N/A	
40	20	E, L	8	20	E, L	8		N/A			N/A	
45	6	E	2	6	E	2		N/A			N/A	
46	6	E, A	2	6	E, A	2	6	E, A	2	6	E	2
47	20	E, L, R	8	20	E, L, R	8		N/A			N/A	
48	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
49	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
50	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
51	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
52	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
53	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
54	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
55	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
56	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
Average	35.5		2.4	31.5		4.1	39.4		4.0	38.9		4.9
Total automated	30			30			19			18		

Cases IDs from Meng et al. Rase by default includes as much code as possible. Rase $_{min}$ chooses the smallest scope that includes the systematic edit. Rase $_{MA}$ refactors the entire method after the edit and Rase $_{MB}$ refactors the entire method before the edit. The **edits** column is AST statement edit operations for refactoring. Refactoring **types** are: E: extract method, R: introduce return object, L: introduce exit label, T: parameterize type, F: form template method, and A: add parameter. Rase uses all the refactoring types in many combinations. Both tables show Rase automates refactoring in many cases: 30 out of 56 pairs, and 20 out of 30 method groups, but not all. Systematic edits scope refactoring opportunities better (Rase and Rase $_{min}$) than methods (Rase $_{MA}$ and Rase $_{MB}$).

edits.

RASE$_{min}$ extracts the minimum common code enclosing systematic edits, as opposed to the maximum common code in default RASE. If we mark the minimum common code for extraction, we may have fewer variations between counterparts which require adding extra code for specialization. On the other hand, we may extract less code than the actual commonality shared between changed methods, leaving redundant code after refactoring. Comparing RASE$_{min}$ and RASE shows that RASE$_{min}$ performs differently from RASE in 8 cases. In 7 of the 8 cases, RASE$_{min}$ is less effective at reducing code size because less common code is extracted. However in case 9, RASE$_{min}$ eliminates more code, because the smaller extracted method does not include control flow jumps, which requires no extra code to interpret flow jumps.

In 6 out of the 30 cases, RASE only uses the *extract method* to perform its refactoring task. All the other cases need a combination of different types of refactoring. The code size change varies between an increase of 11 lines and a decrease of 47 lines. RASE's automated refactoring reduces the code size in 8 cases (27%) for the method pairs.

6.3.2 Method Groups

To explore whether our conclusions based on method pairs generalize to multiple similarly changed methods, we apply RASE to 30 systematically edited method groups. Each group contains at least 3 methods and at most 9 methods. We apply RASE$_{MA}$, RASE$_{MB}$ and RASE$_{min}$ to the same data set and compare refactoring capabilities. The results are mostly similar comparing Table 6.1 and Table 6.2. Column **#** in Table 6.2 shows the number of changed methods in each group. RASE refactors 20 out of 30 cases. Similar to Ta-

Table 6.2: Method groups: Clone removal refactorings

ID	#	Rase edits	Rase types	Rase Δcode	Rase $_{min}$ edits	Rase $_{min}$ types	Rase $_{min}$ Δcode	Rase $_{MA}$ edits	Rase $_{MA}$ types	Rase $_{MA}$ Δcode	Rase $_{MB}$ edits	Rase $_{MB}$ types	Rase $_{MB}$ Δcode
1	6	137	E, A, F, T	-7	137	E, A, F, T	-7	137	E, A, F, T	-7	137	E, A, F, T	-7
2	4	30	E	-10	30	E	-10	30	E	-10	10	E	2
4	3	17	E	-1	10	E, A	4		N/A			N/A	
5	7	36	E, T	-6	16	E	2		N/A			N/A	
6	8	42	E, T	-6	42	E, T	-6		N/A			N/A	
8	3	44	E, A, F	-4	44	E, A, F	-4	44	E, A, F	-4	32	E, A, F	2
9	5	58	E, L, R	18	58	E, L, R	18		N/A			N/A	
10	3	38	E, F	14	38	E, F	14	38	E, F	14	19	E, A, F	13
11	4	20	E	-4	10	E	2		N/A			N/A	
13	3	9	E	3	9	E	3		N/A			N/A	
15	3	32	E, A	-10	28	E	-8		N/A			N/A	
17	3	21	E	-3	9	E	3		N/A			N/A	
18	3	37	E	-11	37	E	-11	37	E	-11	25	E	-5
19	3	96	E, F	6	48	E, F, R	24	96	E, F	6	29	E, A, F, T	13
24	3	59	E, R	-1	59	E, R	-1	59	E, R	-1	8	E	2
25	3	26	E, R	14	26	E, R	14		N/A			N/A	
27	4	20	E, A	-4	20	E, A	-4		N/A			N/A	
28	3	24	E, T	0	24	E, T	0	24	E, T	0	12	E, A, T	0
29	9	211	E	-149	211	E	-149		N/A			N/A	
30	4	26	E, A	-6	26	E, A	-6	26	E, A	-6	15	E, A, T	7
Average		49.2		-8.4	44.1		-6.1	54.5		-2.1	31.9		3.0
Total automated		20			20			9			9		

ble 6.1, we observe that RASE automates refactoring more than RASE_{MA} and RASE_{MB}. RASE produces more concise code than RASE_{min} in 6 out of 30 cases. One difference from the method pair results is that RASE decreases code size more consistently and frequently, reducing code size in 14 of the 20 refactored cases (70%) and on average reducing code by 8 lines. This result is expected because the refactored code appears in just two methods with method pairs, whereas for method groups the refactored code originally appears in three or more methods.

6.3.3 Reasons for Not Refactoring

We examined by hand the 26 cases that RASE did not refactor and found four reasons, which Table 6.3 summarizes. For 7 cases, RASE failed to refactor due to Java's limited support for generic types. It is very difficult to convert some generalized statements like `v instanceof T$`, `T$.m$()`, and `v = new T$()`, into code that compiles.

For 5 cases, RASE did not refactor because some statements cannot be moved correctly into an extracted method. For instance, the super constructor `super(...)` is only valid in constructors and cannot be moved to any other method. Another example is, when attempting to put an extracted method into a newly declared class, calls to `private` methods by the extracted method are not semantically valid because `private` methods are only accessible for methods defined in the same class.

For 8 cases, no edited statement is identified in the new version of each changed method. RASE depends on LASE to create an abstract edit script representing the input systematic changes. If LASE fails to create such an edit script or the edit script only deletes statements from old versions, RASE

Table 6.3: Reasons Rase does not refactor 26 cases: Method pairs

Reason	number of cases
Limited language support for generic types	7
Unmovable methods	5
No edited statement found	8
No common code extracted	6

Table 6.4: Reasons Rase does not refactor 10 cases: Method groups

Reason	number of cases
Limited language support for generic types	2
Unmovable methods	0
No edited statement found	2
No common code extracted	6

cannot locate the code to extract based on the output of Lase, nor can it automate refactorings.

In 6 cases, the marked code snippets in different methods are not generalizable to create an abstract template. Four possible reasons explain this result. First, the code snippets contain different numbers of statements, which prevent Rase from creating an abstract template. Although some existing *clone removal* refactoring techniques [41, 57] leverage program dependence analysis and some heuristics to shift irrelevant variant code and put together extractable code, these techniques cannot handle all the cases in this category either, indicating the difficulty of fully automated factorization and parameterization. Second, the AST node types of some extracted statements do not match, such as `ExpressionStatement` vs. `ReturnStatement`. Third, the number of parameters in method calls do not match, such as `foo(v)`

vs. `bar(a, b, c)`. Although we can create a single method by merging input signatures of different methods, the resulting code may have poor readability. Fourth, there is at least one identifier mapping conflict. For instance, identifier `v` is mapped to an identifier `a` in one statement, but mapped to another identifier `x` in another statement. Ignoring these conflicts to apply refactoring would incorrectly modify the program semantics. Similar to Table 6.3, Table 6.4 shows that limited language support for generic types, no edited or added statement, and no common code extracted are the three main reasons why automated refactoring is infeasible.

In summary, the above sections make the following observations.

- Method based refactoring before or after systematic editing does not eliminate the need for systematic editing.
- Scoping refactoring based on systematic edits improves refactoring applicability over refactoring the entire methods, either before or after the edits.
- Extracting the maximum common code instead of the minimum common code creates a refactored version with a smaller code size.

6.3.4 Examination of software evolution after systematic edits

To understand how RASE's refactoring recommendations correlate with developer refactorings, we manually examine how developers evolved methods after these systematic edits by going through the version histories. The average time interval in the version history, after systematic edits in our test

suite, is 1.3 years. Table 6.5 shows the results for method pairs and Table 6.6 shows method groups. In the tables, the **Feasible** column corresponds to cases when RASE automates refactoring, while **Infeasible** corresponds to the other cases. The **Refactored** row shows cases when developers either by hand or with the help of some other tool refactored code later in the version history. The other rows show break down cases without developer refactoring. **Co-evolved** means the methods are systematically edited at least one more time in later versions, and may indicate that refactoring is desirable. **Divergent** means the methods evolved in divergent ways, and may indicate that refactoring is undesirable. For example, one method was deleted or only one method changed. **Unchanged** means the methods did not get changed after systematic edits, indicating that it may not be worthwhile to refactor because these methods are quite stable, or it is premature to state refactor desirability due to lack of information.

Table 6.5 shows that developers only refactored 4 out of 56 cases. RASE automates refactoring for all 4 of these cases. Additionally, RASE refactors 26 cases which were not refactored by developers. Among these 26 cases, 21 cases had no code changes. However, 9 of these 21 cases were specially crafted in prior work [71] to evaluate SYDIT and they do not have any version history. When code does not need to change to fix bugs or add features, developers are unlikely to aggressively remove clones. In 3 out of 26 cases, developers evolved code differently by either changing both methods differently or deleting one method while maintaining the other one. In 2 out of 26 cases, developers did not refactor code for some reason but actually similarly changed code once again. Such repetitive systematic edits on method pairs may indicate a desire for refactoring.

Table 6.5: Manual evaluation of version history after systematic edits: Method pairs

		Feasible	Infeasible
Refactored		4	0
	Co-evolved	2	6
Unrefactored	**Divergent**	3	10
	Unchanged	21	10

Table 6.6: Manual evaluation of version history after systematic edits: Method groups

		Feasible	Infeasible
Refactored		1	0
	Co-evolved	2	1
Unrefactored	**Divergent**	4	0
	Unchanged	13	9

There are 6 cases in which methods were co-evolved by developers but are not automatically refactored by Rase. The major reason is some methods either invoke certain methods which are not accessible by an extracted method, contain non-contiguous cloned code, or have conflicting identifier mappings between each other. It is not easy to automatically refactor these cases. If developers want to refactor them, they need to first apply some tricks to make the common code extractable.

Table 6.6 summarizes the version history for systematically edited method groups, and show similar results to the method pairs in Table 6.5. Except for the unchanged cases, developers refactored one case, which Rase also handles. Methods co-evolved in 2 cases and diverged in 4 cases, all of which Rase can refactor. There is one case where methods were co-evolved by only deleting

code, but Rase does not refactor because it does not suggest refactoring when the only change is deleted code.

Manually observing the version history reveals that there is no obvious correlation between Rase's automatic refactoring recommendation and manual refactorings done by developers for systematically edited methods, although there are cases when automatic refactoring overlaps with manual refactoring. When developers refactor or not, they do not base their decision solely on code similarity and similar edits, but also based on other factors. Although Rase cannot decide for developers whether to refactor, it conducts an executable refactoring plan when developers decide to refactor, which helps reduce developer burden of code transformation.

To explore the reasons developers do not refactor while performing systematic edits, we randomly pick several examples and ask project owners for their expert opinion.

One developer is conservative about aggressive refactoring and merging commonality between methods: *"(I will not refactor the method pair because) this pair of methods is not a pain point during maintenance/evolution of JDT. That particular class is very stable, and the readability of the code as it is now outweighs potential benefits of refactoring. We have other duplications, that are more likely to cause pain, e.g., by being forgotten during maintenance. ... In these classes, potential gain might be greater, but then a refactoring to avoid redundancy would certainly introduce a significant amount of additional complexity. We don't typically refactor unless we have to change the code for some bug fix or new feature."*

Another developer refactors more proactively to reduce cloned code, but prefers reducing four duplicated methods to two, instead of the single

method that Rase suggests to simplify the class hierarchy. This feedback from developers illustrates that the decision to apply clone removal depends on many criteria in addition to code similarity and co-evolution events, including the effectiveness of cloned code in bug fix and feature additions, the software architecture, readability, and maintainability of the resulting refactored code.

Based on our experience with software version history and communication with developers, we envision Rase as a refactoring recommendation tool when developers think about refactoring duplicated code. It should serve to complement existing systematic editing tools because developers do not always aggressively reduce duplicated code but often maintain redundant code for various reasons.

6.4 Threats to Validity

Our results are based on 86 systematic editing examples. Further evaluation with more subject systems, longer version histories, and larger scope of systematic edits beyond the method level remains as future work.

The refactoring capability of Rase is affected by the systematic editing tool—Lase—it uses. Given multiple similarly changed methods, if Lase fails to generalize an abstract edit script for them, Rase cannot provide any refactoring suggestion. The six types of refactorings implemented in Rase do not cover all possible code transformations applicable to clone removal. However, it is the state-of-the-art in terms of the number of refactorings it automates.

Rase determines concrete refactoring transformations based on an abstract template without considering the global context such as the extent of code duplication across the entire codebase, the class relationship among meth-

ods with and without systematic edits. Therefore, Rase may not suggest the best possible transformation. However, Rase is the first automated tool that mechanically examines if systematic edits in multiple locations indicate refactoring opportunities. We believe that Rase will be useful and enable further research on recommendations and cost/benefit analysis of refactoring.

Our results focus on automated refactoring *feasibility* instead of *desirability*. We leave developers to decide whether to refactor or not. We believe that it is difficult to choose between systematically editing methods and reducing clones to edit a single copy. To assess refactoring desirability, the refactoring cost/benefit analysis must account for multiple factors such as how frequently future systematic edits will occur to fix bugs or add features, if complexity increases, if it is worthwhile to reduce future edits, and if the related methods are likely to change and/or diverge in the future.

6.5 Summary

Similar edits in similar code may indicate that refactoring should be applied to reduce code duplication. By designing and implementing Rase, an automated refactoring tool which combines six refactoring techniques, we show it is difficult to obviate systematic editing with automated refactoring. Furthermore version history examination reveals that developers do not always refactor code to eliminate systematic edits. These observations indicate that both systematic editing and automatic refactoring tools can help software development and maintenance.

While Rase automates clone removal based on systematic edits, the decision of whether to refactor or not depends on multiple complex factors such as readability, maintainability, and types of anticipated changes. Sys-

tematic edits serve as only one among those factors. Therefore, they are not sufficient to indicate clone removal opportunity. We envision Rase's automated refactoring capability will support further research on refactoring cost benefit analysis and refactoring opportunity recommendations.

Chapter 7

Conclusion

Developing and maintaining software is challenging and time-consuming. As computing becomes more ubiquitous and complex, more and more programmers open source their projects to share resources and computation. Developers are encouraged to reuse and modify existing software to build their own applications instead of coding everything from scratch. When codebases are widely shared among developers, code changes will also be shared intentionally or unintentionally. Different from open source projects which are conveniently available in software repositories, systematic edits—similar code changes to different locations—are not well recognized, organized, or utilized. Therefore, there is a great opportunity to pursue further research in systematic editing to help developers modify software more efficiently for better software reliability.

In the following sections, we will first summarize insights we have learnt by automating program transformation. We divide future work into two categories: (1) next steps for improving and evaluating systematic editing and refactoring tools (Section 7.2), and (2) broader implications of these results for open questions in software development (Section 7.3). Section 7.2 elaborates how we would like to generalize the definition of systematic edits and prompt developers for their expertise, in order to automate more diverse code changes and validate transformation correctness. Section 7.3 discusses interesting research questions about mining systematic edits and synthesizing code changes.

We finally conclude with the prospects of our research impact.

7.1 Summary

Based on the observation that many similar edits are applied to locations containing similar code snippets, we argue that automatic program transformation based on code change examples can improve programmer productivity and software correctness. We show that unlike most prior approaches, ours only require exemplar edit(s) from developers to propagate similar edits or to refactor similarly edited code. Our systematic editing approaches demonstrate that the edit relevant context, which is extracted from exemplar edit(s) based on control and data dependence analysis, enables the inferred edits to consistently modify program syntax and semantics. Although the approaches do not check or guarantee correctness of transformed code, our consistent code transformation capability facilitates developers to do correctness checking. Our refactoring approach demonstrates that scoping with systematic edits improves the feasibility of automatic clone removal compared to method-level scoping. Although it is not always feasible or desirable to refactor systematically edited code, our research takes the first step to explore the relationship between systematic editing and refactoring, and to furthermore predict refactoring desirability based on the relationship. Different from the usual refactoring tools, this approach can serve as an intelligent agent to notify developers of *clone removal* opportunities by providing a detailed refactoring plan and transformed code after witnessing similar edits applied to similar code.

7.2 Future Work

In the process of test suite construction, we find that there are many code changes which do not fall into the category of systematic editing. While some research has revealed some common bugs that cross projects, for example, not checking for null, forgetting to free resources, and incorrectly invoking libraries, most of these patterns are not sophisticated [49]. We observe different developers have different coding styles. In particular, different projects target different functionalities, and it is not surprising to find few systematic editing patterns that span different projects. For example, Kim et.al [49] mined common fix patterns from 62656 human-written bug fixes for Eclipse JDT, by identifying commonly shared single line bug fixes. Among the eight simple patterns they found, five patterns are single statement changes such as *Altering method parameters* and *Initializing an object*. The other three fix patterns add conditional checks before accessing certain objects, such as adding a null checker. The study corroborates our experience.

Nevertheless, as developers build their applications by reusing code from open source projects, frameworks, and libraries, they do have good reasons to share code changes. In order to find these common "niddles" in the vast sea code changes, we need a more diverse definition for "systematic editing", to effectively extract relevant code changes, specially identify their common features, and properly recommend applicable code changes to developers. The paradigm of our automatic program transformation will be that the system automatically collects human-written code changes, categorizes similar changes based on some metrics, characterizes each group of changes together with the edit relevant context, and finally locates code containing the relevant context in users' code to suggest corresponding code changes. Each possible definition

of "systematic editing" will provide a way to instantiate the paradigm by describing what kind of code changes are similar, what edit relevant context is shared, and how to modify code which contains the edit context but misses the edit.

There can be different ways to define systematic edits. In addition to the definition provided in Chapter 2, one possible definition can be code changes which syntactically diverge but implement the same algorithm. For each definition, we may explore different strategies to identify, recommend, and check code changes. For instance, for systematic code changes written in the same language which implement the same algorithm, we may use symbolic execution and a constraint solver to reason about their semantic equivalence, to infer preconditions of applying these changes, and later to recommend similar code changes to locations that meet the preconditions. Such code changes can be applied to fix bugs, refactor code, add new features, and parallelize sequential programs. Furthermore, for code changes written in multilingual programs which are semantically equivalent, we may also need a way to combine both static analysis and code generation for different programming languages in order to use the expertise encoded in programs developed in another language for reference. The approach will help maintain legacy systems, propagate semantically equivalent code changes across languages, and translate programs from one language to another language.

To help tools better understand program semantics and the purposes of code changes, developers may be required to provide more hints other than code via annotations, commit messages, or interactive inputs. For each multiline code change, the developer input can describe (1) the change's purpose, such as bug fix, feature addition, refactoring, and parallelizing, (2) the involved

patches and affected code regions, and (3) the oracles to demonstrate expected behaviors of the modified code when the semantics is not preserved. The developer input will not only enhance change understanding, but also guide automatic testing for modified code. For bug fixes and refactoring changes, tools may not need to create new test cases, because existing tests can be sufficient to check whether bugs are fixed or semantics are preserved without introducing new bugs. However, for feature addition changes, tools will automatically create new test cases or modify existing test cases to validate new features. The extra input may burden developers so much that developers are discouraged from submitting code changes in a short term. However, developers will benefit from the effort when having automatic programming support for higher quality code. On the other hand, we will also explore ways to ease the burden of specifying code changes.

Our existing systematic editing approaches do not check or guarantee correctness of transformed code, although they guarantee consistency of program transformations. There is a gap between consistency and correctness, which should be filled with code changes specific to each systematically edited location. The oracles, which we expect from developers describing correct program behaviors, may help tools create those specific changes. The oracles can be described in terms of input/output pairs, predicate annotations, or program skeletons. With input/output pairs, tools may leverage machine learning algorithms to train a fitting function [24], which can be applied to adjust the output of systematically edited code. With predicates, tools could check program status against the specified invariants, generate extra statements or modify existing statements if any invariant is violated [20]. Given possible program statements included in the resulting code, we can use pro-

gram sketching technique to correctly fill in the statement content [87].

Our existing refactoring approach identifies refactoring opportunities solely based on similar edits applied to similar code. However, with more diverse systematic editing definitions, we may recognize more refactoring opportunities for divergent edits applied to dissimilar code and produce more understandable refactored code. With developers' expertise input about code change types, we may better predict automatic refactoring applicability and desirability. For instance, when we detect refactoring code changes in a version history, we can check which bug fixes or feature additions are made before and after the refactoring. Some prior work [15, 53] shows that developers usually apply refactoring together with other types of changes, such as bug fixes and feature additions. By observing the evolution patterns between different types of changes, we can more precisely predict possible refactorings to apply and what they should be like.

7.3 Open Questions

In the thesis, we have explored automatic program transformation based on exemplar edits given by developers. One future direction would be to explore different ways to define and automate systematic edits based on annotations, commit messages, and oracles provided by developers. Below we list some more open issues and challenges for expanding and enhancing automatic program transformation.

How to recognize and leverage unknown code change patterns? A lot of existing bug finding and fixing tools identify and fix bugs based on predefined matching patterns or code change patterns [42, 45, 63, 88]. The change

patterns can be as simple as *replacing* `new Integer(*).toString()` *with* `Integer.toString(*)` [42], or as complicated as checking the access-control implementation in Web applications against predefined access-control policies to recommend candidate fixes for access-control errors of omission [88]. Generally speaking, these tools all have their separate "systematic edit" definition to automate code transformation. Different from our approach, instead of inferring patterns, these tools implement code patterns based on developers' experience of bug symptoms and solutions. Although coding very complicated patterns is challenging, time-consuming, and error-prone, we do not see an easy way to automatically mine the patterns. The reasons is that some patterns, such as access-control patterns, are application and implementation specific. They require a deep understanding of the application rather than the shallow semantics revealed by any complex inter-procedural analysis. Without the deep insight of developers, it is impossible for any automatic approach to identify or leverage sophisticated patterns. Compared with the uninferrable patterns, some simple patterns, such as replacing an API with another API, are inferrable. It is important and worthwhile to investigate the boundary between inferrable code change patterns and uninferrable ones. For the inferrable ones, we may design scalable algorithms to fully automate common code change identification and application. For the uninferrable ones, we may design tools to simplify the programming task of specifying patterns.

Is it possible to synthesize a general-purpose program given a high-level specification, and a well-organized code and change database? Theoretically, if every feature described in the specification has a corresponding code implementation in the database, and all possible interactions between features can be well captured by stored code changes, it may be possible to

automatically forge an arbitrary program out of a high-level specification, using a process similar to, but more generalized than, feature-oriented programming [8]. However, realistically, it is difficult to construct such a comprehensive database which covers everything to program and every possible interaction between programmed components. A very important part of developers' daily life is to invent programs to solve problems never posed or solved before. Therefore, it is unrealistic to assume the existence of such codebase. However, researchers have explored different ways to narrow down the problem space in order to make program synthesis applicable to specific programming scenarios. For instance, Gulwani et al. have developed a series of techniques to synthesize programs for self-defined domain-specific languages [2, 81, 86]. They limit the number of syntactic elements in each language and possible ways to program with the elements. By limiting the search scope, they make it possible to automatically enumerate all possible solutions for constraints described in the high-level specification and then pick the best one. Weimer et.al use genetic programming to synthesize candidate bug fixes for an executable program which contains a single bug [61, 99]. The buggy program is expected to pass some tests while failing other tests. They randomly generate candidate bug fixes by inserting, deleting, or updating code in the program until they find a program variant passing all tests. Genetic programming limits solution search scope to program elements contained by the buggy program and all possible combinations between them. However, the assumption that the solution lies in existing elements in the program is too strong to always lead to a valid bug fix. There may be other ways to define a specific problem and thus limit the solution space. It is worth furthermore investigation to explore principled ways to limit solution space with problem specialization. When tools can identify the scope of synthesizable programs and synthesize a program for

each problem in the scope, programmers will concentrate their time and efforts on problems that have never been solved before or hard to solve, relying on automatic program transformation to implement the other parts with high quality.

7.4 Impact

Our tools and techniques have implications for tool developers and programmers. Tool developers can adopt and improve our approach, building automated repair tools that can spread bug fixes within the same project and sometimes across different projects. They may also build new types of automated refactoring tools, which are no longer limited to conducting whatever developers command them to do, but instead proactively recommend refactoring opportunities to developers. Programmers may use these tools to apply systematic edits to multiple locations with less effort, and to achieve high precision and recall for edit location search. The result will be correct program transformations. We believe this automation will substantially improve programmer productivity.

Bibliography

[1] George W. Adamson and Jillian Boreham. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, 10(7-8):253–260, 1974.

[2] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[3] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *International Conference on Automated Software Engineering*, page (Tool Demo Paper), 2012.

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean 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.

[5] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *CSMR '07*, pages 81–90, 2007.

[6] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, page 292, 1999.

[7] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, page 326, 1999.

[8] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1992.

[9] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.

[11] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *WCRE*, pages 85–94, 2009.

[12] Marat Boshernitsan, Susan L. Graham, and Marti 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.

[13] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[14] S. Burson, G.B. Kotik, and L.Z. Markosian. A program transformation approach to automating software re-engineering. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 314 –322, 31 1990.

[15] Dongxiang Cai and Miryung Kim. An empirical study of long-lived code clones. In *FASE*, 2011.

[16] James R. Cordy, Charles D. Halpern, and Eric Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16:97–107, 1991.

[17] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration.* MIT Press, 1993.

[18] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. *SIGPLAN Not.*, page 18, 2003.

[19] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167, 2007.

[20] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *Proceedings of*

the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, 2007.

[21] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. juzi: a tool for repairing complex data structures. juzi: a tool for repairing complex data structures. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 855–858. ACM, 2008.

[22] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[23] Martin Erwig and Deling 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.

[24] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[25] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *OOPSLA '11*. ACM, 2011.

[26] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling—tree differencing for fine-grained source code change extrac-

tion. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[27] Stephen R. Foster, William G. Griswold, and Sorin Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *ICSE '12*, 2012.

[28] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[29] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[30] Alejandra Garrido and Ralph E. Johnson. Refactoring c with conditional compilation. In *ASE*, 2003.

[31] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *ICSE '12*, 2012.

[32] Nils Göde. Clone removal: Fact or fiction? In *IWSC*, pages 33–40, 2010.

[33] Akira Goto, Norihiro Yoshida, Masakazu Ioka, Eunjong Choi, and Katsuro Inoue. How to extract differences from similar programs? a cohesion metric approach. In *IWSC*, pages 23–29. IEEE, 2013.

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

[35] William G. Griswold, Darren C. Atkinson, and Collin 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.

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

[37] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 317–328, New York, NY, USA, 2011. ACM.

[38] Yoshiki Higo and Shinji Kusumoto. Identifying clone removal opportunities based on co-evolution analysis. In *IWPSE*, pages 63–67, 2013.

[39] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J. Softw. Maint. Evol.*, 20(6):435–461, 2008.

[40] Susan 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.

[41] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. *2011 15th European Conference on Software Maintenance and Reengineering*, 0:53–62, 2012.

[42] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

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

[44] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[45] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Symposium on Operating System Design and Implementation*, 2012.

[46] Nicolas Juillerat and Beat Hirsbrunner. Toward an implementation of the `form template method` refactoring. *SCAM*, 0:81–90, 2007.

[47] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE*, pages 654–670, 2002.

[48] Cory Kapser and Michael W. Godfrey. `cloning considered harmful` considered harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[49] Dongsun Kim, Jaechange Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *IEEE/ACM International Conference on Software Engineering (to appear)*, 2013.

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

[51] Miryung Kim, David Notkin, and Dan 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.

[52] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196, 2005.

[53] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[54] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *POPL*, pages 155–169, 2000.

[55] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *IWPC*, pages 33–, 2003.

[56] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE*, page 301, 2001.

[57] Giri Panamoottil Krishnan and Nikolaos Tsantalis. Refactoring clones: An optimization problem. *ICSM*, 0:360–363, 2013.

[58] David A. Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, 1995.

[59] J. Landauer and M. Hirakawa. Visual awk: a model for text processing by demonstration. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, VL '95, pages 267–, Washington, DC, USA, 1995. IEEE Computer Society.

[60] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. *Learning repetitive text-editing procedures with SMARTedit*, pages 209–226. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[61] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[62] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1), January 2012.

[63] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. pages 289–302, 2004.

[64] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium*

on Foundations of software engineering, pages 306–315, New York, NY, USA, 2005. ACM.

[65] Antoni Lozano and Gabriel Valiente. On the maximum common embedded subtree problem for ordered trees. In *String Algorithmics, Chapter 7. King's College London Publications*, 2004.

[66] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. pages 365–383, 2005.

[67] Toshiyuki Masui and Ken Nakayama. Repeat and predict: two keys to efficient text editing. In *CHI '94*, pages 118–130, 1994.

[68] Alexander Matzner, Mark Minas, and Axel Schulte. Efficient graph matching with application to cognitive automation. In Andy SchÃrr, Manfred Nagl, and Albert ZÃndorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 297–312. Springer Berlin / Heidelberg, 2008.

[69] David Maulsby and Ian H. Witten. Cima: An interactive concept learning system for end-user applications. *Applied Artificial Intelligence*, 11(7-8):653–671, 1997.

[70] Na Meng, Miryung Kim, and Kathryn McKinley. Lase: Locating and applying systematic edits. In *ICSE*, page 10, 2013.

[71] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

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

[73] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. *SIGPLAN Not.*, 1996.

[74] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. pages 302–321, 2010.

[75] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *ACM/IEEE International Conference on Software Engineering*, pages 315–324, 2010.

[76] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Clone-aware configuration management. In *ASE*, pages 123–134, 2009.

[77] Robert Nix. Editing by example. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 186–195, New York, NY, USA, 1984. ACM.

[78] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[79] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles 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.

[80] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. An empirical study of supplementary bug fixes. In *IEEE Working Conference on Mining Software Repositories*, pages 40–49, 2012.

[81] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[82] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles*, pages 87–102, 2009.

[83] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.

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

[85] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[86] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, 2012.

[87] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

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

[89] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, 2012.

[90] G. Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Diane Publishing Company, 2002.

[91] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 2011.

[92] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC*, pages 173–180, 2004.

[93] Nikolaos Tsantalis. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, 2011.

[94] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Ranking refactoring suggestions based on historical volatility. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, pages 25–34, Washington, DC, USA, 2011. IEEE Computer Society.

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

[96] Daniel von Dincklage and Amer Diwan. Converting java classes to use generics. *SIGPLAN Not.*, 2004.

[97] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 457–466, 2010.

[98] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[99] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *IEEE International Conference on Software Engineering*, pages 364–374, 2009.

[100] Ian H. Witten and Dan Mo. *TELS: learning text editing tasks from examples*, pages 183–203. MIT Press, Cambridge, MA, USA, 1993.

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