Chapter 16
Recommending Program Transformations

Automating Repetitive Software Changes

Miryung Kim and Na Meng

Abstract Adding features and fixing bugs in software often require systematic edits which are similar but not identical changes to multiple code locations. Finding all relevant locations and making the correct edits is a tedious and error-prone process. This chapter presents several state-of-the art approaches to recommending program transformation in order to automate repetitive software changes. First, it discusses *programming-by-demonstration* (PBD) approaches that automate repetitive tasks by inferring a generalized action script from a user's recorded actions. Second, it presents edit location suggestion approaches that only recommend candidate edit locations but do not apply necessary code transformations. Finally, it describes program transformation approaches that take code examples or version histories as input, automatically identify candidate edit locations, and apply context awareness, customization program transformations to generate a new program version. In particular, this chapter describes two concrete example-based program transformation approaches in detail, Sydit and Lase. These two approaches are selected for an in-depth discussion, because they handle the issue of both recommending change locations and applying transformations, and they are specifically designed to update programs as opposed to regular text documents. The chapter is then concluded with open issues and challenges of recommending program transformations.

16.1 Introduction

Recent work observes that software evolution often requires *systematic and repetitive changes*. Developers apply similar but not identical changes to different contexts [23, 24, 34, 45]. Nguyen et al. [45] find that 17 to 45 % of bug fixes are recurring fixes that involve similar changes to numerous methods. Another class of

M. Kim (✉) • N. Meng
The University of Texas at Austin, Austin, TX, USA
e-mail: miryung@ece.utexas.edu; miryung@cs.utexas.edu; mengna09@cs.utexas.edu

M.P. Robillard et al. (eds.), *Recommendation Systems in Software Engineering*,
DOI 10.1007/978-3-642-45135-5_16,

systematic changes occur when application programming interface (API) evolution requires all the clients to update their code [17] or when developers refactor code to improve its internal structure. Cross-system bug fixes happen frequently among forked software products such as FreeBSD, NetBSD, and OpenBSD, despite the limited overlap of contributors [6, 54]. Manual application of systematic changes is tedious and error-prone. Developers must find all required change locations, rewrite those locations manually, and test the modifications. A failure to systematically extend software may lead to costly errors of omissions and logical inconsistencies.

For example, Fig. 16.1 shows a systematic change example drawn from revisions to `org.eclipse.debug.core` on 2006-10-05 and 2006-11-06, respectively. The unchanged code is shown in black, additions in blue with a blue"+," and deletions in red with a red"−." Consider methods `mA` and `mB`: `getLaunchConfigurations(ILaunchConfigurationType)` and `getLaunchConfigurations(IProject)`. These methods iterate over elements received by calling `getAllLaunchConfigurations()`, process the elements one by one, and add it to a predefined list when an element meets a certain condition.

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 to 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 to multiple methods cannot be applied using existing refactoring engines in integrated development environment, 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.

Existing source transformation tools automate repetitive changes by requiring developers to prescribe the changes in a formal syntax. For example, TXL [8] is a programming language designed for software analysis and source transformation. It requires users to specify a programming language's structure (i.e., syntax tree) and a set of transformation rules. TXL then automatically transforms any program written in the target language according to the rules. These tools can handle nontrivial *semantics-modifying* changes, such as *inserting a null-check before dereferencing an object*. However, it requires developers to have a good command of language syntax and script programming [3, 4, 19].

Refactoring engines in IDEs automate many predefined *semantics-preserving* transformations. When performing a refactoring task (e.g., rename method), developers only need to decide the refactoring type and provide all necessary information (e.g., the old and new name of the method) as input to enable the transformation. Then the refactoring engines automatically check predefined constraints to ensure that the transformation preserves semantics before actually making the transformation. Although some tools allow developers to define new refactoring types, specifying refactoring preconditions and code transformation from scratch is time consuming and error-prone.

Existing interactive text-editing approaches, such as a *search-and-replace* feature of a text editor, can help developers look for edit locations based on keywords or

```
public ILaunchConfiguration[] getLaunchConfigurations
  (ILaunchConfigurationType type) throws CoreException {
    Iterator iter = getAllLaunchConfigurations().iterator();
    List configs = new ArrayList();
  + ILaunchConfiguration config = null;
    while (iter.hasNext()) {
  - ILaunchConfiguration config = (ILaunchConfiguration)iter.next
      ();
  + config = (ILaunchConfiguration)iter.next();
  + if (!config.inValid()) {
  + config.reset();
  + }
        if (config.getType().equals(type)) {
          configs.add(config);
        }
    }
    return (ILaunchConfiguration[])configs.toArray
      (new ILaunchConfiguration[configs.size()]);
}
```

a

```
protected List getLaunchConfigurations(IProject project) {
   Iterator iter = getAllLaunchConfigurations().iterator();
 + ILaunchConfiguration cfg = null;
   List cfgs = new ArrayList();
   while (iter.hasNext()) {
 - ILaunchConfiguration cfg = (ILaunchConfiguration)iter.next();
 + cfg = (ILaunchConfiguration)iter.next();
 + if (!cfg.inValid()) {
 + cfg.reset();
 + }
      IFile file = cfg.getFile();
      if (file != null && file.getProject().equals(project)) {
         cfgs.add(cfg);
      }
    }
    return cfgs;
}
```

b

Fig. 16.1 Systematic edit from revisions of `org.eclipse.debug.core` [37]. (**a**) $\mathtt{mA}_o$ to $\mathtt{mA}_n$. (**b**) $\mathtt{mB}_o$ to $\mathtt{mB}_n$

regular expressions, and apply edits by replacing the matching text at each location with user-specified text. These approaches treat programs as plain text. Therefore, they cannot handle nontrivial program transformations that require analysis of program syntax or semantics.

This chapter presents several state-of-the art approaches that overcome these limitations by leveraging *user-specified change examples*. First, it discusses

programming-by-demonstration (PBD) approaches that automate repetitive tasks by inferring a generalized action script from a user's recorded actions. However, these PBD approaches are not suitable for updating code as they are designed for regular text. Second, it presents edit location suggestion approaches that stop at only recommending candidate locations but do not apply necessary code transformations. Thus these approaches still require programmers to edit code manually. Finally, it describes program transformation approaches that take code change examples as input, automatically identify candidate edit locations and also apply context-aware, customized program transformations to generate a new program version. In particular, this chapter describes two concrete techniques in detail, Sydit [36, 37] and Lase [38]. We chose to describe these two in detail because Lase has the most advanced edit capability among the techniques that handle both issues of finding edit locations and applying transformation and Sydit is the predecessor of Lase.

Given an exemplar edit, Sydit generates a *context-aware, abstract edit script*, and then applies the edit script to new program locations specified by the user. Evaluations show that Sydit is effective in automating program transformation. However, the tool depends on the user to specify edit locations.

Lase addresses this problem by learning edit scripts from *multiple examples* as opposed to a single example [38]. Lase (1) creates context-aware edit scripts from two or more examples, uses these scripts to (2) automatically identify edit locations and to (3) transform the code. Evaluation shows that Lase can identify edit locations with high precision and recall.

There are several open issues and remaining challenges in recommending program transformations based on examples. First, it is currently difficult for developers to view recommended program transformations, especially when the recommendation spans across multiple locations in the program. Second, it is difficult for developers to check correctness of the recommended program transformations, because none of the existing techniques provide additional support for validating recommended edits. Third, the granularity of program transformations is limited to intra-function or intra-method edits at large, making it difficult to apply high-level transformations such as modifications to class hierarchies and method signatures. Finally, existing techniques are limited to automating homogeneous, repetitive edits, but leave it to developers to coordinate heterogeneous edits.

16.2 Motivation

Software Evolution Often Requires Systematic Changes. This insight arises from numerous other research efforts, primarily within the domain of crosscutting concerns and refactorings. *Crosscutting concerns* represent design decisions that are generally scattered throughout a program such as performance, error handling, and synchronization [21, 59]. Modifications to these design decisions involve similar changes to every occurrence of the design decision. *Refactoring* is the process of improving internal software structure in ways that do not alter its external

behavior. Refactoring often consists of one or more elementary transformations, such as "moving the `print` method in each `Document` subclass to its superclass" or "introducing three abstract `visit*` methods." Another class of systematic changes occur when the evolution of an application programming interface (API) requires all API clients to update their API usage code [17], though the details can vary from location to location [19]. A recent study of bug fixes shows that a considerable portion of bug fixes (17–45 %) are actually recurring fixes that involve similar edits [45]. Another study on code changes finds that on average 75 % of changes share similar structural-dependence characteristics, e.g., invoking the same method or accessing the same data field [23]. These studies indicate that systematic code updates are common and often unavoidable.

Manual Implementation of Systematic Changes Is Tedious and Error-Prone. Purushothaman and Perry [53] found that only about 10 % of changes (in one, large industrial system) involve a single line of code, but even a single line change has about a 4 % chance of resulting in an error; on the other hand, changes of 500 lines or more have nearly a 50 % chance of causing at least one defect. Eaddy et al. [11] find that the more scattered the implementation of a concern is, the more likely it is to have defects. Murphy-Hill et al. [43] find that almost 90 % of refactorings are performed manually without the help of automated refactoring tools. These refactorings are potentially error-prone since they often require coordinated edits across different parts of a system. Weißgerber and Diehl [64] find that there is an increase in the number of bugs after refactorings. Kim et al. [22] also find a short-term increase in the number of bug fixes after API-level rename, move, and signature change refactorings. Some of these bugs were caused by inconsistent refactorings. These studies motivate automated tool support for applying systematic edits.

Systematic Changes Are Generally not Semantics-Preserving and They Are Beyond the Scope and Capability of Existing Refactoring Engines. To investigate the challenges associated with refactorings, Kim et al. [25] conducted a survey with professional developers at Microsoft. They sent a survey invitation to 1,290 engineers whose commit messages include a keyword "refactoring" in the last 2 years of version histories of five MS products; 328 of them responded to the survey. More than half of the participants said they carry out refactorings in the context of bug fixes or feature additions, and these changes are generally not semantics-preserving transformations. In fact, when developers are asked about their own definition of refactoring, 46 % of participants did not mention preservation of semantics, behavior, or functionality at all. During a follow-up interview, some developers explicitly said, "Strictly speaking, refactoring means that behavior does not change, but realistically speaking, it usually is done while adding features or fixing bugs." Furthermore, over 95 % of participants in the study said that they do most refactorings manually; 53 % reported that refactorings that they perform do not match the types and capability of transformations supported by existing refactoring engines. This motivates a flexible, example-based approach for applying systematic program transformations.

16.3 State-of-the Art Approaches to Recommending Program Transformations

This section describes state-of-the art approaches to recommending program transformations, compares these approaches using a unified framework, and discusses their strengths and weaknesses. We first discuss individual approaches and present comparison results in Tables 16.1–16.3. Table 16.1 shows the comparison of existing approaches in terms of input, output, edit type, and automation capability. Table 16.2 describes the comparison of existing approaches in terms of edit capability: the second column shows whether each technique can handle single line or multiple-line edits; the third column shows whether each technique handles a sequence of contiguous edits or non-contiguous edits; the fourth column shows whether it supports only replication of concrete edits or edits that can be customized to individual target contexts; and the last column shows whether the technique models surrounding unchanged code or not. Table 16.3 shows the comparison of existing approaches, in terms of evaluation subjects, programming languages, data set size, and assessment methods.

16.3.1 Programming-by-Demonstration

Programming-by-example [30] (PbE) is a software agent-based approach that infers a generalized action script from a user's recorded actions. SMARTedit [28] automates repetitive text edits by learning a series of functions such as *"move a cursor to the end of a line."* Like macro recording systems, SMARTedit learns the program by observing a user performing her or his task. However, unlike macro-recorders, SMARTedit examines the context in which the user's actions are performed and learns programs that work correctly in new contexts. Using a machine learning concept called *version space algebra*, SMARTedit is able to learn useful text-editing after only a small number of demonstrations. Similarly, Visual AWK [27] allows users to interactively generalize text edits.

Several approaches learn string manipulations or a skeleton of repetitive editing tasks from examples or demonstrations. For example, the Editing by Example (EBE) system looks at the input and output behavior of the complete demonstration [47]. EBE synthesizes a program that generalizes the transformation expressed by text change examples. The TELS system records editing actions, such as search-and-replace, and generalizes them into a program that transforms input into output [65]. TELS also uses heuristic rules to match actions against each other to detect loops in the user's demonstrated program. However, TELS's dependence on domain-specific heuristic rules makes it difficult to apply the same techniques to a different domain, such as editing Java programs. The Dynamic Macro system of Masui and Nakayama [32] records macros in the emacs text editor. Dynamic Macro performs automatic segmentation of the user's actions, breaking up the stream of actions into repetitive

Table 16.1 Comparison of existing approaches in terms of input, output, edit type, and automation capability

Tool	Input	Output	Type	Location	Xform
Visual AWK [27]	Recorded editing actions and text to modify	Modified version of the text	Textual	Semi-automatic	Semi-automatic
EBE [47]	Reference edit in terms of textual differences	Transformation program	Textual	Automatic	Automatic
TELS [65]	Recoded editing actions	A transformation program	Textual	Semi-automatic	Semi-automatic
Dynamic macro system [32]	Edit action log	Text editing macro	Textual	Automatic	Automatic
Cima [33]	Reference edit in terms of old and new versions of changed example(s)	Transformation rule	Textual	Automatic	Automatic
Simultaneous text editing [41]	Demonstration of text editing and text to modify	Modified version of all selected text	Textual	Manual	Automatic
Linked editing [60]	Demonstration of text editing and code clones to modify	Modified version of all linked code clones	Syntactic	Manual	Automatic
CloneTracker [10]	Demonstration of text editing and code clones to modify	Modified version of all linked code clones	syntactic	Automatic	Automatic
Clever [46]	Two consecutive revisions of subject system	Modified version of system	Syntactic	Automatic	Automatic
Trident [19]	Search terms and replacement terms	Modified version of system	Lexical/syntactic	Automatic	Automatic
Program synthesis [13]	Reference edit in terms of old and new versions of changed example(s) and text to modify	Modified version of text	Textual	Automatic	Automatic
Reverb [34, 35]	Demonstration of source editing	Similar locations in the project	Semantic	Automatic	Manual
DQL [62]	DQL query and codebase	Locations matching the query	Semantic	Automatic	Manual
PQL [31]	PQL query and codebase	Locations matching the query	Semantic	Automatic	Automatic
PR-Miner [29]	Codebase	Locations violating extracted general programming rules	Semantic	Automatic	Manual

(continued)

Table 16.1 (continued)

Tool	Input	Output	Type	Location	Xform
HAM [5]	Project's version history	Code changes which are likely to be crosscutting concern	Semantic	–	–
Find-concept [57]	Natural language-based query and codebase	Locations matching the query	Semantic	Semi-automatic	–
FixWizard [45]	Manually identified bug fixes and codebase	Locations in need of recurring bug fixes	Semantic	Automatic	Manual
LibSync [44]	Client programs already migrated to library's new version and those not yet	Locations to update with edit suggestions	Semantic	Automatic	Manual
iXj [4]	Sample code to replace together with abstract replacement code	Visualized transformation program	Syntactic	Automatic	Automatic
ChangeFactory [55]	Recorded editing actions	A transformation program	Semantic	–	–
spdiff [2]	Reference edit in terms of multiple old and new methods	Semantic patch	Semantic	Automatic	Automatic
Coccinelle [48]	Semantic patch and codebase	Modified version of codebase	Semantic	Automatic	Automatic
ClearView [50]	Normal executions and erroneous executions of program	Dynamically patched program	Semantic	Automatic	Automatic
Weimer et al. [63]	Faulty program and test suites	Fixed program	Semantic	Automatic	Automatic
Sydit [36,37]	Reference edit in terms of old and new method and target location	Modified version of target	Semantic	Manual	Automatic
Lase [18,38]	Reference edit in terms of multiple old and new methods and codebase	Modified version of codebase	Syntactic	Automatic	Automatic

Table 16.2 Comparison of existing approaches in terms of edit capability

Tool	Multiple vs. Single	Contiguous vs. non-contiguous	Abstract vs. concrete	Context modeling
Visual AWK [27]	Single	Contiguous	Concrete	No
EBE [47]	Single	Contiguous	Concrete	No
TELS [65]	Single	Contiguous	Concrete	No
Dynamic macro system [32]	Single	Contiguous	Concrete	No
Cima [33]	Single	Contiguous	Concrete	Yes
Simultaneous text editing [41]	Single	Contiguous	Concrete	No
Linked editing [60]	Multiple	Non-contiguous	Concrete	Yes
CloneTracker [10]	Multiple	Non-contiguous	Concrete	Yes
Clever [46]	Multiple	Non-contiguous	Concrete	Yes
Trident [19]	Single	Non-contiguous	Abstract	No
Program synthesis [13]	Single	Contiguous	Concrete	No
Reverb [34, 35]	–	–	–	–
DQL [62]	–	–	–	–
PQL [31]	Single	Contiguous	Abstract	Yes
PR-Miner [29]	–	–	–	–
HAM [5]	–	–	–	–
Find-concept [57]	–	–	–	–
FixWizard [45]	–	–	–	–
LibSync [44]	–	–	–	–
iXj [4]	Single	Contiguous	Abstract	No
ChangeFactory [55]	–	–	–	–
spdiff [2]	Single	Contiguous	Abstract	No
Coccinelle [48]	Single	Contiguous	Abstract	No
ClearView [50]	Multiple	Non-contiguous	Abstract	Yes
Weimer et al. [63]	Single	Contiguous	Concrete	No
Sydit [36, 37]	Multiple	Non-contiguous	Abstract	Yes
Lase [18, 38]	Multiple	Non-contiguous	Abstract	Yes

subsequences, without requiring the user to invoke the macro-editor explicitly. Dynamic Macro performs no generalization and relies on several heuristics for detecting repetitive patterns of actions. The Cima system [33] learns generalized rules for classifying, generating, and modifying data, given examples, hints, and background knowledge. It allows a user to give hints to the learner to focus its attention on certain features, such as the particular area code preceding phone numbers of interests. However, the knowledge gained from these hints is combined with a set of hard-coded heuristics. As a result, it is unclear which hypotheses Cima is considering or why it prefers a certain inferred program over another. In general, these PBD approaches are not suitable for editing a program because they do not consider a program's syntax, control, or data dependencies.

Simultaneous text editing automates repetitive editing [41]. Users interactively demonstrate their edit in one context and the tool replicates *identical, lexical* edits on the preselected code fragments. Simultaneous text editing cannot easily handle

Table 16.3 Comparison of existing approaches in terms of evaluation

Tool	Subjects	Lng.	Data size	Evaluation
Visual AWK [27]	–	–	–	–
EBE [47]	–	–	–	–
TELS [65]	Text	–	3 text editing tasks	Count how many interactions needed between the system and its user to bake a correct transformation program
Dynamic macro system [32]	Text	–	Unknown	Compare results against user experience for 1 year
Cima [33]	Text	–	5 text editing tasks	User study
Simultaneous text editing [41]	Text	–	3 text editing tasks	User study
Linked editing [60]	Text	–	1 code clone linking task and 2 text editing tasks	User study
CloneTracker [10]	5 open source projects	Java	5 clone groups	User study
Clever [46]	7 open source projects	Java	7 open source projects	Compare results against version histories
Trident [19]	1 API migration	Java	1 open source project	Case study
Program synthesis [13]	Text	–	Unknown	User study
Reverb [34, 35]	2 open source projects	Java	2 open source projects	Compare results against manually determined locations
DQL [62]	2 open source projects	Java	4 searching tasks	Compare results against those of a clone detection tool and a text search tool
PQL [31]	6 open source projects	Java	3 searching tasks	Manually check results
PR-Miner [29]	3 open source projects	C	Top 60 reported rule violations	Manually check results
HAM [5]	3 open source projects	Java	Top 50 aspect candidates	Manually check results
Find-concept [57]	4 open source projects	Java	9 searching tasks	User study
FixWizard [45]	5 open source projects	Java	1414 bug fixes	Manually check results
LibSync [44]	6 pairs of API–client open source projects	Java	67 suggested locations together with adaptive changes	Manually check results

iXj [4]	1 open source project	Java	1 program transformation task	User study
ChangeFactory [55]	–	–	–	–
spdiff [2]	Linux and client programs	C	4 known patches	Compare results against known patches
Coccinelle [48]	Linux and client programs	C	26 known patches	Compare results against known patches
ClearView [50]	Firefox	C	10 known bugs	Test the tool's ability to successfully survive security attacks
Weimer et al. [63]	10 open source projects	C	10 known faulty programs	Run each generated program against the standard test suites
Sydit [36,37]	5 open source projects	Java	56 example method pairs	Compare the results against version histories
Lase [18,38]	2 open source projects	Java	61 program transformation tasks	Compare results against version histories

similar yet different edits because its capability is limited in instantiating a *syntactic*, context-aware, abstract transformation. Linked Editing [60] applies the same edits to a set of code clones specified by a user. CloneTracker [10] takes the output of a clone detector as input and automatically produces an abstract syntax-based clone region descriptor for each clone. Using this descriptor, it automatically tracks clones across program versions and identifies modifications to the clones. Similar to Linked Editing, it uses the longest common subsequence algorithm to map corresponding lines and to echo edits in one clone to other counterparts upon a developer's request. The Clever version control system detects inconsistent changes in clones and propagates *identical* edits to inconsistent clones [46]. Clever provides limited support in adapting the content of learned edits by renaming variable names suitable for target context. However, because Clever does not exploit program structure, when abstracting edits, it does not adapt the edit content to different contexts beyond renaming of variables. Trident [19] aims to support refactoring of dangling references by permitting the developer to specify lexical and syntactic constraints on search terms and replacement terms, locating potential matches and applying requested replacements; an iterative process is supported allowing the developer to back out of a given requested change atomically.

Program synthesis is the task of automatically synthesizing a program in some underlying language from a given specification using some search techniques [13]. It has been used for a variety of applications such as string manipulation macros [14], table transformation in Excel spreadsheets [16], geometry construction [15], etc. The synthesizer then completes the program satisfying the specification [58]. However, these program synthesis approaches do not currently handle automation of similar program changes in mainstream programming languages such as Java because they do not capture control and data flow contexts nor abstract identifiers in edit content.

In summary, the programming by demonstration approaches can learn edits from examples, but they are mostly designed for *regular text documents* instead of *programs*. Thus they cannot handle program transformations that require understanding program syntax and semantics.

In Table 16.1, the top one-third compares the above-mentioned PBD approaches in terms of inputs, outputs, and automation capability. Column **Type** describes the type of edit operations: textual edit vs. syntactic edits vs. semantic edits. Column **Location** describes whether each technique can find locations automatically, semiautomatically, or manually. Column **Transformation** describes whether each technique can apply transformations automatically, semiautomatically, or manually. Table 16.1 shows that most PBD approaches can handle only textual edits or they are very limited in terms of syntactic program editing capability.

In Table 16.2, the top one-third compares the above-mentioned PBD approaches in terms of edit capability. Column **Multiple vs. Single** shows whether each technique can apply multiline or single line edits. Column **Contiguous vs. Non-contiguous** describes whether each technique can only apply contiguous program transformations or also apply transformations separated with gaps. Column **Abstract vs. Concrete** describes whether each technique can apply

customized abstract edits to different edit locations or simply apply identical concrete edits. Column **Context Modeling** describes whether each technique models the surrounding unchanged code relevant to edit operations in order to position edits correctly. The symbol "—" is recorded when a technique does not apply any edit automatically. Table 16.2 shows that most PBD approaches handle only concrete edits and are unable to apply edits customized to fit target program contexts.

Furthermore, as shown in Table 16.3, some techniques do not have any user study or only have done a study involving a handful of editing tasks. When the symbol "—" is recorded for **Subjects**, **Data size**, and **Evaluation** means no empirical study is reported. "**—**" recorded for **Lng.** (Language) means the technique targets plain text instead of any specific programming language.

16.3.2 Edit Location Suggestion

Code matching and example search tools can be used to identify similar code fragments that often require similar edits. Reverb [34] watches the developer make a change to a method and searches for other methods in the project where the syntax and semantics are similar to the original ones of the exemplar; however, it does not apply the transformations. DQL [62] helps developers to locate code regions that may need similar edits; developers can write and make queries involving dependence conditions and textual conditions on the system-dependence graph of the program so that the tool automatically locates code satisfying the condition. PQL [31] is a high-level specification language focusing on specifying patterns that occur during a program run. The PQL query analyzer can automatically detect code regions matching the query. Similarly, PR-Miner [29] automatically extracts implicit programming rules from large software code and detects violations to the extracted programming rules, which are strong indications of bugs. While all these tools could be used to identify candidate edit locations that may require similar edits, none of these tools help programmers in automatically applying similar changes to these locations. There are other tools that are similar to PQL and PR-Miner, such as JQuery [9] or SOUL [40]. While they can be used to find edit locations via pattern matching, they do not have a feature of automatically applying program transformations to the found code snippets. Similarly, Castro et al. diagnose and correct design inconsistencies but only *semiautomatically* [7].

Concern mining techniques locate and document crosscutting concerns [5, 57]. Shepherd et al. [57] locate concerns using natural language program analysis. Breu and Zimmermann [5] mine aspects from version history by grouping method-calls that are added together. However, these tools leave it to a programmer to apply similar edits, when these concerns evolve. We do not exhaustively list all concern mining techniques here. Please refer to Kellens et al. [20] for a survey of automated code-level aspect mining techniques. In Chap. 5, Mens and Lozano [39] discuss techniques that recommend edit locations based on mined source code patterns.

FixWizard identifies code clones, recognizes recurring bug fixes to the clones, and suggests edit locations and exemplar edits [45]. Yet, it does *not generate syntactic edits*, nor does it support abstraction of variables, methods, and types. LibSync helps client applications migrate library API usages by learning migration patterns [44] with respect to a partial AST with containment and data dependencies. Though it suggests example API updates, it is *unable* to transform code. These limitations leave programmers with the burden of manually editing the suggested edit locations, which is error-prone and tedious.

In summary, the middle parts of Tables 16.1–16.3 show the comparison of the above-mentioned edit location suggestion techniques. These techniques can be used to find changed locations automatically, but leave it to developers to manually apply necessary transformations. While the evaluation of some techniques involves real open source project data, none evaluates them in the context of a user applying similar program transformations to the found locations.

16.3.3 Generating and Applying Program Transformations from Examples

To reduce programmers' burden in making similar changes to similar code fragments, several approaches take code change examples as input, find change locations, and apply customized program transformations to these locations. These approaches are fundamentally different from source transformation tools or refactoring engines, because users do not need to specify the script of repetitive program transformations in advance. Rather, the skeleton of repetitive transformations is generalized from change examples. This section lists such approaches and discusses their capability.

Sydit takes a code change example in Java as input and automatically infers a generalized edit script that a user can use to apply similar edits to a specified target [36,37]. Their subsequent work Lase uses multiple change examples as input, automatically infers a generalized edit script, locates candidate change locations, and applies the inferred edit to these change locations [38]. Both Sydit and Lase infer the context of edit, encode edit positions in terms of surrounding data and control flow contexts, and abstract the content of edit scripts, making it applicable to code that has a similar control and data flow structure but uses different variable, type, and method names.

iXj [4] and ChangeFactory [55] provide interactive source transformation tools for editing a program. iXj does not generalize code transformation, though it has a limited capability of generalizing the scope of transformation. ChangeFactory requires a user to generalize edit content and location manually.

To support API migration, Lawall et al. [1, 2, 48] find differences in the API usage of client code, create an edit script, and transform programs to use updated APIs. Their approach is limited in two respects: the edit scripts are confined

to term-replacements and they only apply to API usage changes. While it uses control- and data-dependence analysis to model the context of edits [1], the inferred context includes only inserted and deleted API method invocations and control and data dependencies among them. Their context does not include unchanged code on which the edits depend. Thus, when there is no deleted API method invocation, the extracted context cannot be used to position edits in a target method. Sydit is more flexible, because it computes edit context that is not limited to API method invocations and it can include unchanged statements related to edits. Therefore, even if the edits include only insertions, Sydit can correctly position edits by finding corresponding context nodes in a target method.

Automatic program repair generates candidate patches and checks correctness using compilation and testing [50, 63]. For example, the approach of Perkins et al. [50] generates patches that enforce invariants observed in correct executions but are violated in erroneous executions. It tests patched executions and selects the most successful patch. Weimer et al. [63] generate their candidate patches by replicating, mutating, or deleting code *randomly* from the existing program and thus far have focused on single line edits.

In summary, the bottom one-third of Tables 16.1–16.3 summarizes the comparison of the above-mentioned techniques. While these techniques can be used to find edit locations and apply transformations, some can handle only single line edits or contiguous edits. Very few can go beyond replication of concrete edits.

In the next two sections, we discuss two concrete example-based program transformation approaches in detail, Sydit and Lase. These two approaches are selected to discuss in depth for two reasons. First, they are specifically designed for updating programs as opposed to regular text documents. Second, they handle the issue of both recommending edit locations and applying transformations. They also have strengths of modeling change contexts correctly and customizing edit content appropriately to fit the target contexts. We discuss Sydit first, because Lase extends Sydit by leveraging multiple edit examples instead of a single example.

16.4 SYDIT: Generating Program Transformations from a Single Example

This section describes Sydit [37], which generates an *abstract, context-aware* edit script from a single changed method and applies it to a user-specified target. To facilitate illustration, we use Fig. 16.1 as a running example throughout this section.

16.4.1 Generating an Edit Script from a Single Example

There are two phases in Sydit. Phase I takes as input an old and new version of method `mA` to create an *abstract, context-aware* edit script Δ. Phase II applies Δ to a target method, `mB`, producing a modified method $\texttt{mB}_s$.

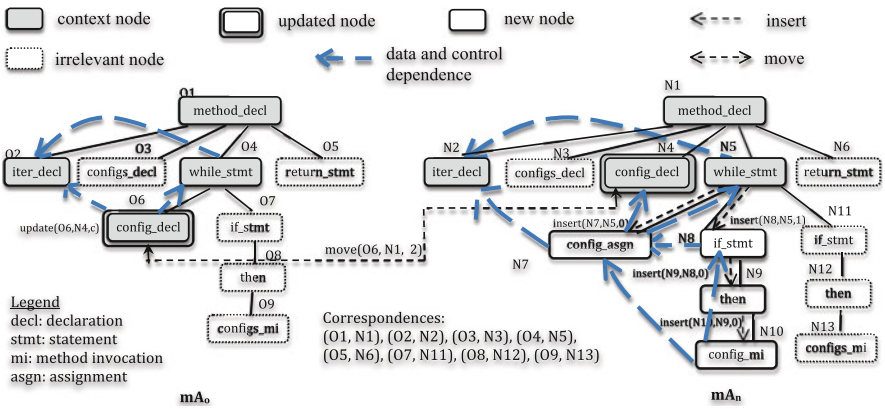

Fig. 16.2 Extraction of a syntactic edit from $\mathtt{A}_{old}$ and $\mathtt{A}_{new}$ and identification of its context [37]

Phase I: Creating Edit Scripts. Given $\mathtt{mA}_o$ and $\mathtt{mA}_n$, Sydit compares their syntax trees using a program differencing tool [12], to create an edit $\Delta_A = [e_1, e_2, \ldots, e_n]$, as a sequence of abstract syntax tree (AST) node additions, deletions, updates, and moves, described as follows:

insert (Node u, Node v, int k) Insert u and position it as the $(k+1)^{th}$ child of v.
delete (Node u) Delete u.
update (Node u, Node v) Replace u's label and AST type with v's while maintaining u's position in the tree.
move (Node u, Node v, int k) Delete u from its current position and insert it as the $(k+1)^{th}$ child of v.

For our example, the inferred edit Δ_A between $\mathtt{mA}_o$ and $\mathtt{mA}_n$ is shown below.

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

Figure 16.2 shows the edit in a graphical way. It indexes all nodes to simplify explanation. For each edit, Sydit extracts relevant context from both old and new versions of a changed method using control-, data-, and containment-dependence analysis. Here, the *context* relevant to an edit includes the edited nodes and nodes on which they depend. For instance, since the inserted node `N7` is contained by and control dependent on `N5`, data dependent on `N2` and `N4`, `N2`, `N4`, `N5`, and `N7` are

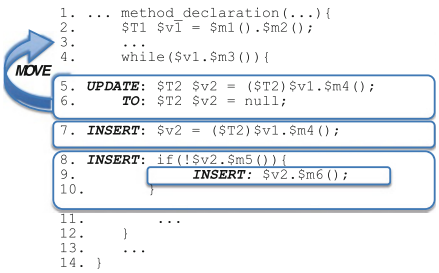

Fig. 16.3 Abstract edit script derived from Fig. 16.2 [37]

extracted as context relevant to the insert operation. The extracted context reflects the control-, data-, and containment-dependence constraints the exemplar changed method has on the derived edit. Given a target method, Sydit looks for the context's correspondence in the method to ensure that all underlying constraints are satisfied. If such a correspondence is found, Sydit infers that a similar edit is applicable to the method, ignoring statements irrelevant to the edit.

Sydit then creates an *abstract, context-aware edit script* Δ, by replacing all concrete types, methods, and variables with unique symbolic identifiers `$T`x, `$m`x, and `$v`x, where x is a number, and recalculating each edit operation's location with respect to its extracted context. This step generalizes the edit script, making it applicable to code using different identifiers or structurally different code. For instance, Fig. 16.3 shows a resulting abstract edit script derived from Fig. 16.2. It abstracts the `config` variable in `mA` to `$v2`. After removing all irrelevant statements, for the moved `ILaunchConfiguration` declaration, Sydit calibrates its source location as a child position 0 of `while` (i.e., its first AST child node), and target location as a child position 1 of the method declaration node.

Phase II: Applying Edit Scripts. When given a target method, Sydit looks for nodes in the method that match the abstract context nodes in Δ and induce one-to-one mappings between abstract and concrete identifiers. The node mapping problem can be rephrased as a subtree isomorphism problem, which looks for a subtree in the target method's AST matching the given context's tree. Sydit uses an algorithm specially designed to solve the problem [37]. The algorithm establishes node matches in a bottom-up manner. It first establishes matches for all leaf nodes in the context tree, and then does so for all inner nodes based on leaf matching result. If every node in the abstract context finds a unique correspondence in the target method's tree, Sydit infers that the abstract edit script can be customized to an edit script applicable to the method. It then establishes identifier mappings based on the node mappings. In our example, $\texttt{mB}_o$ contains a subtree corresponding to the abstract context for Δ, so Sydit can create a concrete edit script for $\texttt{mB}_o$ out of Δ. Since Sydit establishes a mapping between the abstract node `$T2 $v2 = null` and concrete node `ILaunchConfiguration cfg = null`, it aligns the identifiers used and infers mapping `$T2` to `ILaunchConfiguration`, `$v2` to `cfg`.

Sydit next proceeds to generate concrete edits for the target. With identifier mappings derived above, it replaces abstract identifiers used in Δ with corresponding

concrete identifiers found in the target method, such as replacing `$v2` with `cfg`. With node mappings derived above, it recalculates each edit operation's location with respect to the concrete target method. For example, it calibrates the target move location as child position 1 of $\texttt{mB}_o$'s method declaration node. After applying the resulting edit script to $\texttt{mB}_o$, Sydit produces a suggested version $\texttt{mB}_s$, which is the same as $\texttt{mB}_n$ shown at the bottom of Fig. 16.1.

16.4.2 Evaluation

Sydit is evaluated on 56 method pairs that experienced similar edits from Eclipse JDT Core, Eclipse Compare, Eclipse Core Runtime, Eclipse Debug, and jEdit. The two methods in each pair share at least one common syntactic edit and their content is at least 40 % similar according to the syntactic differencing algorithm of Fluri et al. [12]. These examples are then manually inspected and categorized 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* over types, methods, and identifiers. Table 16.4 shows the number of examples in each of the six categories. Note that there are only six categories instead of eight, since non-contiguous edits always involve multiple nodes.

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 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 $\texttt{mB}_s$, which is compared against $\texttt{mB}_n$ to measure Sydit's effectiveness. In Table 16.4, "matched" is the number of examples for which Sydit matches the change context learnt from $\texttt{mA}$ to the target method $\texttt{mB}_o$ and produces some edits. 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, i.e., that $\texttt{mB}_s$ is semantically equivalent to $\texttt{mB}_n$.

$$\text{The "coverage" row is } \frac{\text{"matched"}}{\text{"examples"}}, \quad \text{and "accuracy" is } \frac{\text{"correct"}}{\text{"examples"}}.$$

The "similarity" measures how similar $\texttt{mB}_s$ is to $\texttt{mB}_n$ for the examples which Sydit can match learnt context and produce some edits. The results are generated using Sydit's default context extraction method, i.e., one source node and one sink node for each control- and data-dependence edge, in addition to a parent node of each edited node, since the configuration is evaluated to produce the best results. For this configuration, Sydit matches the derived edit context and creates an edit for 46 of 56 examples, achieving 82 % coverage. In 39 of 46 cases, the edits are semantically equivalent to the programmer's hand edit. Even for those cases in which Sydit

Table 16.4 Sydit's coverage and accuracy on preselected targets [37]

	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)

produces a different edit, the output and the expected output are often similar. On average, Sydit's output is 96 % similar to the version created by a human developer. While this preliminary evaluation shows accuracy for applying a known systematic edit to a given target location, it does not measure the accuracy for applying the edit to all locations where it is applicable because Sydit is unable to find edit locations automatically. The next section describes the follow-up approach (Lase) that leverages multiple examples to find edit candidates automatically.

16.5 LASE: Locating and Applying Program Transformations from Multiple Examples

Sydit produces code transformation from a single example. It relies on programmers to specify *where* to apply the code transformation, and it does not automatically find edit locations. This section describes Lase, which uses multiple edit examples instead of a single example to infer code transformation, automatically searches for edit locations, and applies customized edits to the locations [38].

16.5.1 Why Learning from Multiple Examples?

The edit script inferred by Sydit is not always well suited to finding edit locations for two reasons. First, the mechanism of learning from a single example cannot disambiguate which changes in the example should be generalized to other places while which should not. As a result, it simply generalizes every change in the example and thus may *overspecify* the script. The over-specification may make the extracted edit context too specific to the example, failing to match places where it should have matched. Second, the full identifier abstraction may *over generalize* the script, allowing the extracted edit context to match places that it should not have matched, because they use different concrete identifiers.

Lase seeks an edit script that serves double duty, both finding edit locations and accurately transforming the code. It learns from two or more exemplar edits given by the developer 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 make similar edits to more than one place, providing multiple examples could be a natural interface.

We use Fig. 16.4 as a running example throughout the section. Consider the three methods with similar changes: `mA`, `mB`, and `mC`. All these methods perform similar tasks: (1) iterate over all elements returned by `values()`, (2) process elements one by one, (3) cast each element to an object of a certain type, and (4) when an element meets a certain condition, invoke the element's `update()` method. Additionally, `mA` and `mB` also experience some specific changes, respectively. For instance, `mA` deletes two `print` statements before the `while` loop. `mB` deletes one `print` statement inside the `while` loop and adds an extra type check and element processing.

16.5.2 Learning and Applying Edits from Multiple Examples

Lase creates a *partially abstract, context-aware* edit script from multiple exemplar changed methods, finds edit locations using the extracted context in the edit script, and finally applies the edit script to each location. There are three phases in Lase. Phase I takes as input several changed methods, such as `mA` and `mB`, to create a *partially abstract, context-aware* edit script Δ_p. Phase II uses the extracted context in Δ_p to search for edit locations which can be changed similarly, such as `mC`. Phase III applies Δ_p to each found location and suggests a modified version to developers. Figure 16.5 shows the inferred edit script from `mA` and `mB` in Fig. 16.4. The details of Lase's edit generalization, location search, and edit customization algorithms are described elsewhere [38].

```
public void textChanged (TEvent event) {
   Iterator e = fActions.values().iterator();
- print(event.getReplacedText());
- print(event.getText());
   while(e.hasNext()) {
- MVAction action = (MVAction)e.next();
- if(action.isContentDependent())
- action.update();
+ Object next = e.next();
+ if (next instanceof MVAction) {
+ MVAction action =(MVAction)next;
+ if(action.isContentDependent())
+ action.update();
+ }
  }
  System.out.println(event + " is processed");
}
```

a

```
public void updateActions () {
  Iterator iter = getActions().values().iterator();
  while(iter.hasNext()) {
- print(this.getReplacedText());
- MVAction action=(MVAction)iter.next();
- if(action.isDependent())
- action.update();
+ Object next = iter.next();
+ if (next instanceof MVAction) {
+ MVAction action =(MVAction)next;
+ if(action.isDependent())
+ action.update();}
+ }
+ if (next instanceof FRAction) {
+ FRAction action = (FRAction)next;
+ if(action.isDependent())
+ action.update();}
+ }
  }
  print(this.toString());
}
```

b

```
public void selectionChanged (SEvent event) {
  Iterator e = fActions.values().iterator();
  while(e.hasNext()) {
- MVAction action=(MVAction)e.next();
- if(action.isSelectionDependent())
- action.update();
+ Object next = e.next();
+ if (next instanceof MVAction) {
+ MVAction action =(MVAction)next;
+ if(action.isSelectionDependent())
+ action.update();
+ }
  }
}
```

c

Fig. 16.4 A systematic edit to three methods based on revisions from 2007-04-16 and 2007-04-30 to `org.eclipse.compare` [38]. (**a**) $\texttt{mA}_o$ to $\texttt{mA}_n$. (**b**) $\texttt{mB}_o$ to $\texttt{mB}_n$. (**c**) $\texttt{mC}_o$ to $\texttt{mC}_n$

```
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

Fig. 16.5 Partially abstract, context-aware edit script derived from `mA` and `mB` [38]

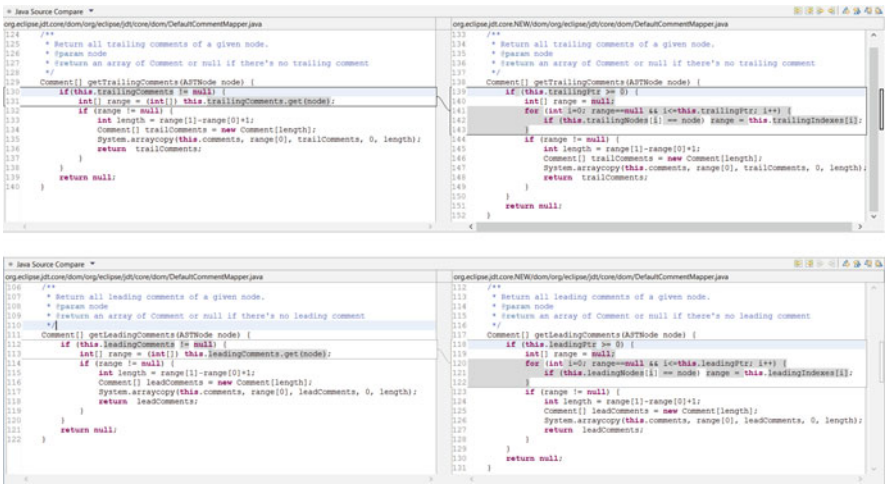

Fig. 16.6 A programmer makes similar but not identical edits to `getTrailingComments` and `getLeadingComments`. While `getTrailingComments` involves edits to `trailingComments` and `trailingPtr`, `getLeadingComments` involves edits to `leadingComments` and `leadingPtr`. The two examples are provided as input to Lase to generate a partially abstract, context-aware edit script [38]

16.5.3 LASE as an Eclipse Plugin

The Lase approach described above is implemented as an Eclipse IDE plugin [18]. Suppose that Bob modifies the code comment processing logic in `org.eclipse.jdt` by updating two methods `getTrailingComments` and `getLeadingComments` in `org.eclipse.jdt.core.dom.DefaultCommentMapper`, shown in Fig. 16.6. In the `getTrailingComments` method, he modifies the `if` condition, modifies an assignment to `range`, and inserts a `for` loop to scan for a given AST node. In the `getLeadingComments` method, he makes a similar edit by modifying its `if` condition, an assignment to `range`, and by inserting a `for` loop. After making these

repetitive edits to the two methods, Bob suspects a similar edit may be needed to all methods with a comment processing logic. He uses Lase to automatically search for candidate edit locations and view edit suggestions.

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

Edit Operation View. For each example, using an *edit operation view*, Bob examines the details of constituent edit operations (*insert*, *delete*, *move*, and *update*) with respect to underlying abstract syntax trees. In this view, Bob can also examine corresponding edit context—surrounding unchanged code that is control- or data dependent on the edited code. Figure 16.7a shows edit operations and corresponding context within the AST of the method `getTrailingComments`. The AST nodes include both unchanged nodes and changed nodes which are the source and/or target of individual insert, delete, move, or update operations. These nodes can be expanded to show more details.

Edit Script Hierarchy View. To create an edit script from multiple examples, Lase generalizes exemplar edits, pair-by-pair. Lase creates a base cluster for each method. It then compares them pair-by-pair. By merging the results of two cluster nodes, Lase generalizes common edit sequences in the edit hierarchy through a bottom-up construction.

For example, by opening the *edit script hierarchy view* shown in Fig. 16.7b, Bob can examine a group of inferred edit scripts at different abstraction levels. By default, Lase uses the top node, i.e., an edit script inferred from *all* examples. By clicking a node in the edit script hierarchy, Bob may select a different subset of provided examples to adjust the abstraction level of an edit script. The selected script is used to search for edit locations and generate customized edits.

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

Bob then examines the edit suggestions for the first candidate method `getExtendedEnd` using the *comparison view* (see Fig. 16.8). He sees that `getExtendedEnd` contains the same structure as his example methods. For example, the `if` statement checking whether `trailingComments` is set to `null` and the assignment to `range`. When viewing the Lase's edit suggestions, Bob notices that the suggested change involves inserting new variables. Lase cannot infer the names of the new variables because there are no matching variable names in the target context. Bob thus chooses

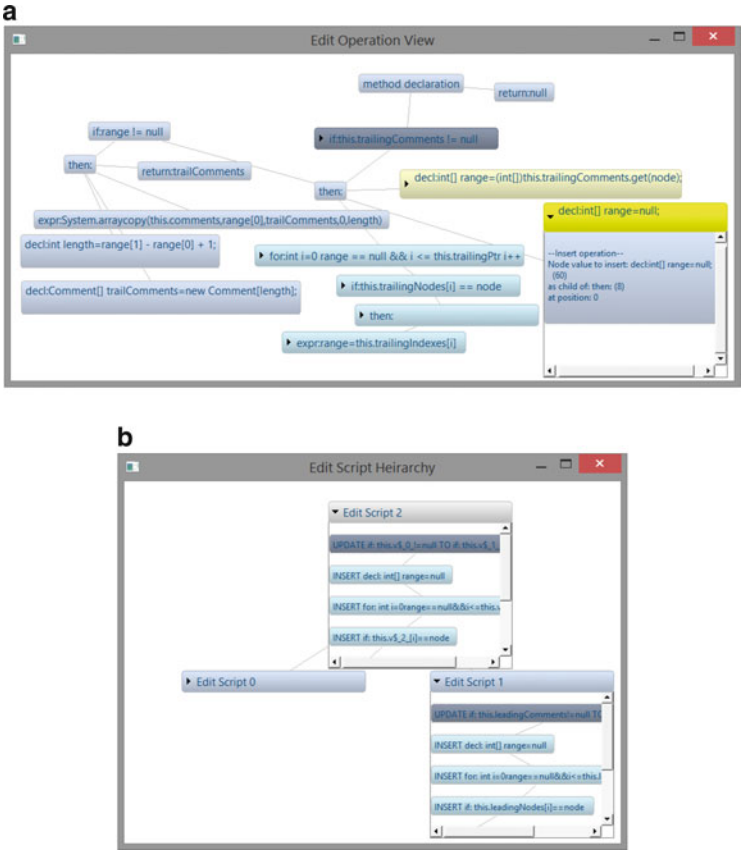

Fig. 16.7 (**a**) Lase visualizes edit operations and corresponding context with respect to the AST. (**b**) Lase learns an edit from two or more examples. Each node in the edit script hierarchy corresponds to an edit script from a different subset of the input examples [38]

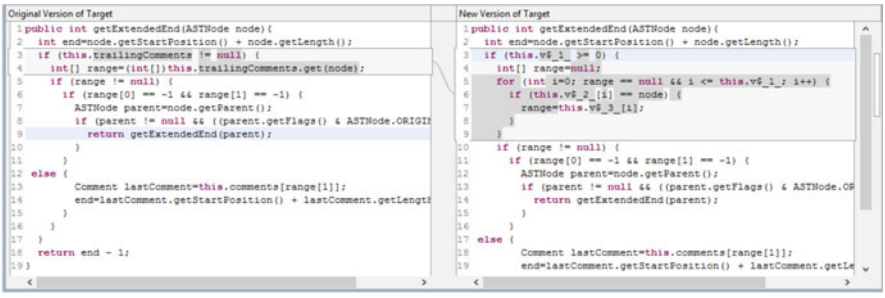

Fig. 16.8 A user can review and correct edit suggestions generated by Lase before approving the tool-suggested edit [38]

the names of those variables by replacing `$v_1_`, `$v_2_`, and `$v_3_` with concrete names. Choosing variables and any other changes Bob wishes to make could be easily done by making direct modifications on the edit suggestion in this comparison view. He applies the modified edits and repeats the process with the other methods.

16.5.4 Evaluation

To measure Lase's precision, recall, and edit correctness, a test suite of supplementary bug fixes [49,54] was used. (See Walker and Holmes [61] in Chap. 12 about an evaluation method using a change history-based simulation). Precision and recall are regarding identified method-level edit locations and edit correctness measures the accuracy of applied edits to the found method locations. Supplementary bug fixes are fixes that span multiple commit, where initial commits tend to be incomplete or incorrect, and thus developers apply supplementary changes to resolve the issue or the bug. If a bug is fixed more than once and there are clones of at least two lines in bug patches checked in at different times, they are manually examined for systematic changes. Using this method, 2 systematic edits in Eclipse JDT and 22 systematic edits in Eclipse SWT are found.

Meng et al. then use these patches as an oracle test suite for correct systematic edits and test if Lase can produce the same results as the developers given the first two fixes in each set of systematic fixes. If Lase, however, produces the same results as developers do in later patches, it indicates that Lase can help programmers detect edit locations earlier, reduce errors of omissions, and make systematic edits. Lase locates edit positions with respect to the oracle data set with 99 % precision, 89 % recall, and performs edits with 91 % edit correctness. Furthermore, given the test suite, Lase identifies and correctly edits nine locations that developers confirmed they missed.

The number of exemplar edits from which Lase learns a systematic edit affects its effectiveness. To determine how sensitive Lase is to different numbers of exemplar edits, Meng et al. randomly pick seven cases in the oracle data set and enumerate subsets of exemplar edits, e.g., all pairs of two exemplar methods. They evaluate the precision, recall, and edit correctness for each set separately and calculate an average for exemplar edit sets for each cardinality to determine how sensitive Lase is to different numbers of exemplar edits. Table 16.5 shows the results.

Our hypothesis is as the number of exemplar edits increases, precision and edit correctness should decrease while recall should increase, because the more exemplar edits provided, the less common context is likely to be shared among them, and the more methods may be found to match the context. However, as shown in Table 16.5, precision P does not change as a function of the number of exemplar edits except for case 12, where two exemplar edits cause the highest precision because exemplar edits are very different from each other. Recall R is more sensitive to the number of exemplar edits, increasing as a function of exemplars.

Table 16.5 Lase's effectiveness when learning from multiple examples [38]

	Exemplars (#)	P (%)	R (%)	EC (%)
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 13	2	100	100	95
	3	100	100	94
	4	100	100	93
	5	100	100	91
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

In theory, edit correctness EC 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 edit correctness. 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 edit correctness.

16.6 Open Issues and Challenges

This section discusses open issues and challenges of recommending program transformations from examples.

Finding Input Examples. While the techniques discussed in Sects. 16.4 and 16.5 learn a generalized program transformation script from examples, it is still left to developers to provide multiple examples for edit script generation and refinement.

Where do these examples come from? Developers can construct the examples on purpose or carefully pick them out of the codebase they are working on. However, a more efficient way is to automatically detect repetitive code changes. One possibility is to mine software version history for similar code changes [23,26] by comparing subsequent snapshots of codebase. Another possibility is to observe developers' edit actions to recognize recurring code changes by monitoring the commands or keystrokes a developer inputs.

Granularity. Most approaches in Sect. 16.3 target replication of intra-method edits. For higher level edits, such as modifying a class hierarchy or delegating an existing task to a newly created methods, we need more complicated edit types to define and more sophisticated context modeling approaches to explore. The edit types should handle the coordination of heterogeneous edits, i.e., various edits to different program entities, in addition to replication of homogeneous edits. For instance, an edit type "Rename" includes renaming an entity (i.e., class, method, field, variable) and modifying all references to the entity. The context modeling approaches should correlate a changed code entity with other entities in the same class hierarchy or performing the same task. For instance, if a method is inserted to an interface, all classes directly implementing the class should be included as edit context as they need to add implementations for the newly declared method.

Context Characterization. The effectiveness of example-based program transformation approaches is affected by the amount of dependence information encoded in the abstract change context C derived from an exemplar edit. For example, given a statement inserted in a `for` loop, the edit could be applicable to all `for` loops, resulting in higher recall but lower precision. However, if the approach requires a context with a control-dependence chain that includes an `if` controlling execution of the `for`, then this context will help find fewer candidates and waste less time on testing extraneous cases. Determining a setting for context extraction requires a rigorous empirical study: (1) varying the number of dependence hops k, (2) varying the degree of identifier abstraction for variable, method, and type names, (3) including upstream- and/or downstream-dependence relations, (4) using containment dependencies only, etc.

Edit Customization. While some approaches are able to customize edit content to fit the target context, it is generally difficult to customize edit content in the target context, when it involves inserted code only. For example, in Lase, edits are customized based on mapping between symbolic identifiers and concrete identifiers discovered from a target context. However, such mappings cannot always be found for inserted code that only exists in the new version. For instance, as shown in Fig. 16.9, since `actionBars` only exists in $\texttt{A}_{new}$ and `serviceLocator` only exists in $\texttt{B}_{new}$, it is difficult to infer `serviceLocator` to use in $\texttt{B}_{new}$ from `actionBars` used in $\texttt{A}_{new}$. In this case, existing approaches borrow verbatim code, `actionBars`, from the source edit, and add it to the target edit without recognizing naming conversion patterns. As a result, it may produce semantically equivalent code with poor readability: e.g., `IServiceLocator actionBars` instead of `IServiceLocator`

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

a

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

b

Fig. 16.9 A motivating example for synthesizing target-specific identifiers [37]. (**a**) $\mathtt{A}_{old}$ to $\mathtt{A}_{new}$. (**b**) $\mathtt{B}_{old}$ to $\mathtt{B}_{suggested}$

`serviceLocator`. A better strategy is to synthesize the target-specific identifiers by inferring naming patterns from the source edit. This requires a natural language analysis of programs [52], e.g., semantic analysis of identifier names used in the target context.

Integrated Compilation and Testing. Existing tools suggest edits without checking correctness, so developers need to decide whether the suggestion is correct on their own. In extreme cases, when tools provide suggestions with many false positive, developers may spend more time examining the tools' useless suggestions than manually making systematic changes without any tool support. Before suggesting the edits, a recommendation tool may proactively compile a suggested version and run regression tests relevant to the proposed edits by integrating existing regression test selection algorithms. If the suggested version does not fail more tests, a user may have higher confidence in it. Otherwise, the tool may locate failure-inducing edits by integrating existing change impact analysis algorithms. This step is similar to speculatively exploring the consequences of applying quick fix recommendations in an IDE [42] and can help prevent a user from approving failure-inducing edits.

Edit Validation. During the inspection process, a user may still want to reason about the deeper semantics of the suggested edits. While this is a program differencing problem, a naive application of existing differencing algorithms may not help developers much—by definition, syntactic edits to the source and the target are the same. A new validation approach is needed to allow developers to focus their attention to *differential deltas*—differences between the effect of a reference edit ($\mathtt{A}_{old}$ to $\mathtt{A}_{new}$) and the effect of a target edit ($\mathtt{B}_{old}$ to $\mathtt{B}_{suggested}$). The insight behind this approach is that developers may have good understanding of a reference edit already and what they want to know is subtle semantic discrepancies caused by porting a

reference edit to a new context. For example, one may compare the control and data flow contexts of a reference edit against those of a ported edit. Another possible approach is to compare the path conditions and effects involving a reference edit against those of a ported edit in the target context [51]. Such semantic comparison could help developers validate whether there exists behavioral differences.

Edit Script Correction. Before accepting recommendation transformations, a user may want to correct the derived script or suggested edits. After correction, the tool may rematch the modified script and recompute the suggested edits, providing feedback to the user. To detect errors inadvertently introduced by manual edits such as a name capture of a preexisting variable, the tool must check name binding and def-use relation preservation [56].

16.7 Conclusion

Systematic changes—making similar but not identical changes to multiple code locations—are common and often unavoidable, when evolving large software systems. This chapter has described the existing body of knowledge and approaches to address this problem. First, it described PBD techniques designed to automate repetitive tasks and discussed how EBE approaches are inadequate for automating program transformations due to their inability to model program-specific syntax and semantics. Second, it overviewed recommendation techniques that suggest candidate edit locations but do not manipulate code by applying code transformations to these locations. Third, it described *example-based program transformation approaches* that take code change examples as input, infer a generalized program transformation script, locate matching candidate locations, and apply the script to these locations. Existing approaches were compared using unified comparison criteria in terms of required inputs, user involvement, the degree of automation, edit capability, evaluation method, and scale to date. In particular, this chapter summarized two approaches, Sydit and Lase. These approaches were selected for an in-depth discussion because they are the most advanced in terms of their capability to position edits correctly by capturing the control- and data-flow contexts of the edits, and to apply non-contiguous, abstract program edits. These strengths make it possible to apply the inferred script to new contexts in a robust manner. The chapter concluded with a set of open problems and challenges that remain to be tackled to fully solve the problem of automating systematic software updates.

Acknowledgments We thank Kathryn McKinley who contributed to designing and evaluating key algorithms of Sydit and Lase. We also thank her for our fruitful collaboration and numerous discussions on example-based program transformation approaches. We thank John Jacobellis who is a main contributor of developing the Lase Eclipse plugin. This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, and CCF-0811524 and by a Microsoft SEIF award.

References

1. Andersen, J.: Semantic patch inference. Ph.D. thesis, University of Copenhagen (2009)
2. Andersen, J., Lawall, J.L.: Generic patch inference. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 337–346 (2008). doi:10.1109/ASE.2008.44
3. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS: program transformations for practical scalable software evolution. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 625–634 (2004). doi:10.1109/ICSE.2004.1317484
4. Boshernitsan, M., Graham, S.L., Hearst, M.A.: Aligning development tools with the way programmers think about code changes. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 567–576 (2007). doi:10.1145/1240624.1240715
5. Breu, S., Zimmermann, T.: Mining aspects from version history. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 221–230 (2006). doi:10.1109/ASE.2006.50
6. Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M.: Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 143–152 (2011). doi:10.1145/1985441.1985463
7. Castro, S., Brichau, J., Mens, K.: Diagnosis and semi-automatic correction of detected design inconsistencies in source code. In: Proceedings of the International Workshop on Smalltalk Technologies, pp. 8–17 (2009). doi:10.1145/1735935.1735938
8. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. **61**(3), 190–210 (2006). doi:10.1016/j.scico.2006.04.002
9. De Volder, K.: JQuery: a generic code browser with a declarative configuration language. In: Hentenryck, P. (ed.) Practical Aspects of Declarative Languages. Lecture Notes in Computer Science, vol. 3819, pp. 88–102. Springer, Heidelberg (2006). doi:10.1007/11603023_7
10. Duala-Ekoko, E., Robillard, M.P.: Clone region descriptors: representing and tracking duplication in source code. ACM T. Software Eng. Meth. **20**(1), 3:1–3:31 (2010). doi:10.1145/1767751.1767754
11. Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do crosscutting concerns cause defects? IEEE T. Software Eng. **34**(4), 497–515 (2008). doi:10.1109/TSE.2008.36
12. Fluri, B., Würsch, M., Pinzger, M., Gall, H.C.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE T. Software Eng. **33**(11), 725–743 (2007). doi:10.1109/TSE.2007.70731
13. Gulwani, S.: Dimensions in program synthesis. In: Proceedings of the ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming, pp. 13–24 (2010). doi:10.1145/1836089.1836091
14. Gulwani, S.: Automating string processing in spreadsheets using input–output examples. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages, pp. 317–330 (2011). http://doi.acm.org/10.1145/1926385.1926423
15. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 50–61 (2011). http://doi.acm.org/10.1145/1993498.1993505
16. Harris, W.R., Gulwani, S.: Spreadsheet table transformations from examples. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 317–328 (2011). doi:10.1145/1993498.1993536
17. Henkel, J., Diwan, A.: CatchUp!: capturing and replaying refactorings to support API evolution. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 274–283 (2005). doi:10.1145/1062455.1062512

18. Jacobellis, J., Meng, N., Kim, M.: LASE: an example-based program transformation tool for locating and applying systematic edits. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 1319–1322 (2013). doi:10.1109/ICSE.2013.6606707
19. Kapur, P., Cossette, B., Walker, R.J.: Refactoring references for library migration. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 726–738 (2010). doi:10.1145/1869459.1869518
20. Kellens, A., Mens, K., Tonella, P.: A survey of automated code-level aspect mining techniques. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development IV. Lecture Notes in Computer Science, vol. 4640, pp. 143–162. Springer, Heidelberg (2007). doi:10.1007/978-3-540-77042-8_6
21. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 1241, pp. 220–242 (1997). doi:10.1007/BFb0053381
22. Kim, M., Cai, D., Kim, S.: An empirical investigation into the role of API-level refactorings during software evolution. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 151–160 (2011). doi:10.1145/1985793.1985815
23. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 309–319 (2009). doi:10.1109/ICSE.2009.5070531
24. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 187–196 (2005). doi:10.1145/1081706.1081737
25. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 50:1–50:11 (2012). doi:10.1145/2393596.2393655
26. Kim, S., Pan, K., Whitehead Jr., E.E.J.: Memories of bug fixes. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 35–45 (2006). doi:10.1145/1181775.1181781
27. Landauer, J., Hirakawa, M.: Visual AWK: a model for text processing by demonstration. In: Proceedings of the IEEE International Symposium on Visual Languages, pp. 267–274 (1995). doi:10.1109/VL.1995.520818
28. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Learning repetitive text-editing procedures with SMARTedit. In: Lieberman, H. (ed.) Your Wish is My Command: Programming by Example, pp. 209–226. Morgan Kaufmann, Los Altos, CA (2001)
29. Li, Z., Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 306–315 (2005). doi:10.1145/1081706.1081755
30. Lieberman, H. (ed.): Your Wish Is My Command: Programming by Example. Morgan Kaufmann, Los Altos, CA (2001)
31. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 365–383 (2005). doi:10.1145/1094811.1094840
32. Masui, T., Nakayama, K.: Repeat and predict: two keys to efficient text editing. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 118–130 (1994). doi:10.1145/191666.191722
33. Maulsby, D., Witten, I.H.: Cima: an interactive concept learning system for end-user applications. Appl. Artif. Intell.: Int. J. **11**(7–8), 653–671 (1997). doi:10.1080/088395197117975
34. McIntyre, M., Walker, R.J.: Assisting potentially-repetitive small-scale changes via semi-automated heuristic search. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 497–500 (2007). doi:10.1145/1321631.1321718

35. McIntyre, M.: Supporting repetitive small-scale changes. MSc thesis, University of Calgary (2007)
36. Meng, N., Kim, M., McKinley, K.S.: Sydit: creating and applying a program transformation from an example. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 440–443 (2011a). doi:10.1145/2025113.2025185
37. Meng, N., Kim, M., McKinley, K.S.: Systematic editing: generating program transformations from an example. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 329–342 (2011b). doi:10.1145/1993498.1993537
38. Meng, N., Kim, M., McKinley, K.S.: LASE: locating and applying systematic edits by learning from examples. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 502–511 (2013). doi:10.1109/ICSE.2013.6606596
39. Mens, K., Lozano, A.: Source code based recommendation systems. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering. Springer, Heidelberg, Chap. 5. (2014)
40. Mens, K., Wuyts, R., D'Hondt, T.: Declaratively codifying software architectures using virtual software classifications. In: Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems, pp. 33–45 (1999). doi:10.1109/TOOLS.1999.778997
41. Miller, R.C., Myers, B.A.: Interactive simultaneous editing of multiple text regions. In: Proceedings of the USENIX Annual Technical Conference, pp. 161–174 (2001)
42. Muşlu, K., Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Speculative analysis of integrated development environment recommendations. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 669–682 (2012). doi:10.1145/2384616.2384665
43. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE T. Software Eng. **38**(1), 5–18 (2011). doi:10.1109/TSE.2011.41
44. Nguyen, H.A., Nguyen, T.T., Wilson Jr., G., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to API usage adaptation. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 302–321 (2010a). doi:10.1145/1869459.1869486
45. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Recurring bug fixes in object-oriented programs. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 315–324 (2010b). doi:10.1145/1806799.1806847
46. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Clone-aware configuration management. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 123–134 (2009). doi:10.1109/ASE.2009.90
47. Nix, R.: Editing by example. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages, pp. 186–195 (1984). doi:10.1145/800017.800530
48. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in Linux device drivers. In: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 247–260 (2008). doi:10.1145/1352592.1352618
49. Park, J., Kim, M., Ray, B., Bae, D.H.: An empirical study of supplementary bug fixes. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 40–49 (2012). doi:10.1109/MSR.2012.6224298
50. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.F., Zibin, Y., Ernst, M.D., Rinard, M.: Automatically patching errors in deployed software. In: Proceedings of the ACM Symposium on Operating Systems Principles, pp. 87–102 (2009). doi:10.1145/1629575.1629585
51. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 226–237 (2008). doi:10.1145/1453101.1453131

52. Pollock, L.: Leveraging natural language analysis of software: achievements, challenges, and opportunities. In: Proceedings of the IEEE International Conference on Software Maintenance, p. 4 (2012). doi:10.1109/ICSM.2012.6405245
53. Purushothaman, R., Perry, D.E.: Toward understanding the rhetoric of small source code changes. IEEE T. Software Eng. **31**(6), 511–526 (2005). doi:10.1109/TSE.2005.74
54. Ray, B., Kim, M.: A case study of cross-system porting in forked projects. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 53:1–53:11 (2012). doi:10.1145/2393596.2393659
55. Robbes, R., Lanza, M.: Example-based program transformation. In: Proceedings of the International Conference on Model-Driven Engineering of Languages and Systems. Lecture Notes in Computer Science, vol. 5301, pp. 174–188 (2008). doi:10.1007/978-3-540-87875-9_13
56. Schaefer, M., de Moor, O.: Specifying and implementing refactorings. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 286–301 (2010). doi:10.1145/1869459.1869485
57. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the International Conference on Aspect-Oriented Software Deveopment, pp. 212–224 (2007). doi:10.1145/1218563.1218587
58. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 167–178 (2007). doi:10.1145/1250734.1250754
59. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: *N* degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 107–119 (1999). doi:10.1145/302405.302457
60. Toomim, M., Begel, A., Graham, S.L.: Managing duplicated code with linked editing. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 173–180 (2004). http://dx.doi.org/10.1109/VLHCC.2004.35
61. Walker, R.J., Holmes, R.: Simulation: a methodology to evaluate recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering. Springer, Heidelberg, Chap. 12. (2014)
62. Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H., Yu, J.X.: Matching dependence-related queries in the system dependence graph. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 457–466 (2010). doi:10.1145/1858996.1859091
63. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 364–374 (2009). doi:10.1109/ICSE.2009.5070536
64. Weißgerber, P., Diehl, S.: Are refactorings less error-prone than other changes? In: Proceedings of the International Workshop on Mining Software Repositories, pp. 112–118 (2006). doi:10.1145/1137983.1138011
65. Witten, I.H., Mo, D.: TELS: Learning Text Editing Tasks from Examples, pp. 183–203. MIT, Cambridge, MA (1993)