1 Making Changes

Program transformation is the process to transform a program to another program. A typical example is compilers. As shown in Figure 1, a Java compiler transforms Java source code into Java bytecode, an intermediate representation of the program. The bytecode can be either interpreted by a bytecode interpreter to directly execute on Java Virtual Machine (JVM), or further transformed by a Just-in-Time (JIT) compiler to machine code for faster execution.

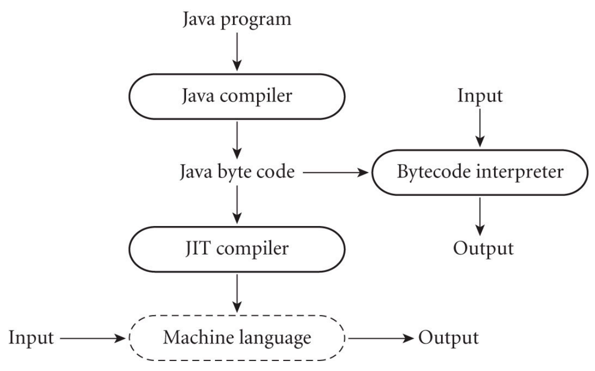


Fig. 1. Java compilation system

Manually transforming programs is tedious and error-prone, because people make mistakes when manipulating code to enforce the explicit or implicit transformation rules. Various automatic systems have been built to transform programs in a mechanic way. Such systems are built for different purposes. For instance, a source-to-source compiler translates programs from one programming language to another language. Source code generation produces source code based on an ontological model, which defines the types, properties, and interrelationships of entities. Many systems guarantee that each transformed program is semantically equivalent to the original program, while other systems do not preserve such semantic equivalence while transforming code. In this section, we will first overview various methods, techniques, and tools to automate source-to-source program transformation (Section 1.1), and then especially discuss refactoring—one kind of semantic-preserving transformation (Section 1.2), and systematic editing—one kind of transformation that does not preserve semantics (Section 1.3).

1.1 Automated Source-to-Source Program Transformation

Various methods, techniques, and tools have been designed and implemented to automatically transform programs from one format to another format. Since we will further discuss refactoring and systematic editing in the following sections, in this section, we mainly introduce the following two types of automatic program transformation: cross-language transformation, and programming by demonstration.

Cross-Language Program Transformation. Developers translate code from one language to another language for various reasons. For instance, when maintaining a legacy system that was written in Fortran decades ago, programmers may migrate the system to a mainstream general purpose language, such as Java, to facilitate the maintenance of existing codebase and to extend the system by leveraging new features of the popular language. When building phone apps, Mobile developers may port a mobile application from one platform (e.g., Android) to another (e.g. iOS) by translating code from Java to Swift.

Most researchers and engineers built cross-language program transformation tools by hard coding the translation rules and implementing any missing equivalent functionality between languages [52, 36, 46, 3, 1]. For instance, SPiCE translates Smalltalk programs to C [52]. The two languages are different in two aspects. First, the execution model of Smalltalk, which creates activation records as objects, is very different from that of C. Second, Smalltalk and C have very different approaches to storage management. To overcome the challenges, Yasumatsu et al. created runtime replacement classes implementing the same functionality of Smalltalk classes that are inherently part of the Smalltalk execution model. They also provided semi-conservative real-time compacting garbage collection that works without language support. Mossienko [36] and Sneed [46] automated COBOL-to-Java code migration by defining and implementing rules to generate Java classes, methods, and packages from COBOL programs. Although some of these tools can perform very complicated code translations tasks, it always costs a lot of time and effort to manually build such tools from scratch between any two languages. HTML5 is a markup language used for structuring and presenting content on the World Wide Web [2]. To simplify cross-platform mobile software development, PhoneGap [4] was built to automatically translate HTML5 implementation to Android or iOS native code.

TXL is a source transformation language designed to translate or manipulate programming languages [10]. Given a context-free grammar to describe program syntax, and a set of transformation rules to manipulate the syntax, TXL automatically transforms programs of the syntactic structure by applying those rules. With TXL, developers to not need to build programming language translators by coding every line of implementation. Instead, the TXL program transformation engine can automatically translate code once developers specify all needed grammars and rules. Researchers built tools to perform various code translation tasks using TXL [9, 22, 13, 47]. For instance, Hassan et al. migrated web applications between different web development frameworks like ASP and NSP [22], while El-Ramly et al. converted Java programs to C# [13].

mppSMT is the most recent code translation tool, which automatically infers and applies Java-to-C# migration rules using a phrase-based statistical machine translation approach [39]. It encodes both Java and C# source files into sequences of syntactic symbols, called syntaxemes, and then relies on the syntaxemes to align code and to train a sequence-to-sequence translation model. Unlike TXL which requires users to specify mapping rules, mppSMT automatically infers those rules from the corresponding implementations in both languages. However, this approach depends on the highly similar program syntaxes between Java and C#. It also requires tool builders to manually define rules to encode program syntactic components to syntaxemes.

Programming by Demonstration (PbD). It is also called programming by example (PbE) [30]. This 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.

Various approaches were built to generate programs based on the text-editing actions demonstrated or text change examples provided by users [42, 50, 27, 28]. For instance, TELS records editing actions, such as search-and-replace, and generalizes them into a program that transforms input to output [50]. It leverages heuristics to match actions against each other to detect any loop in the user-demonstrated program. Similarly, SMARTedit [28] automates repetitive text-editing tasks by learning programs to perform them using techniques drawn from machine learning. SMARTedit represents a text-editing program as a series of functions that alter the state of the text editor (i.e., the contents of the file, or the cursor position). Like macro recording systems, SMARTedit learns the program by observing a user performing her 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.

Program synthesis is the task of generating a program in a domain-specific language from a given specification using some search techniques [18]. For instance, Gulwani defined a string manipulation language that supports restricted forms of regular expressions, conditionals, and loops, and also defined an algorithm to synthesize a program in this language from input-output string examples [19]. The algorithm was implemented as an interactive add-in for Microsoft Excel spreadsheet system. Each time when users manipulate a string and provide both the before- and after- versions of the manipulation, the algorithm treats the input-output example as a constraint that a desired program should satisfy, and searches for all candidate programs that can produce the output given the input. As users provide more input-output examples, the algorithm filters out the candidates that partially satisfy some of the constraints, until finding the program which satisfies all input-output constraints. Such techniques were also used in other problem domains, such as table transformation in Excel spreadsheets [21], geometry construction [20], data structure transformations [14], and hierarchical structured data manipulation [51].

Simultaneous editing repetitively applies source code changes that are interactively demonstrated by users [35]. When users apply their edits in one program context, the tool replicates the *exact lexical* edits to other code fragments, or transforms code accordingly. For instance, Linked Editing requires users to first specify the similar code snippets which they want to modify in the same way [48].

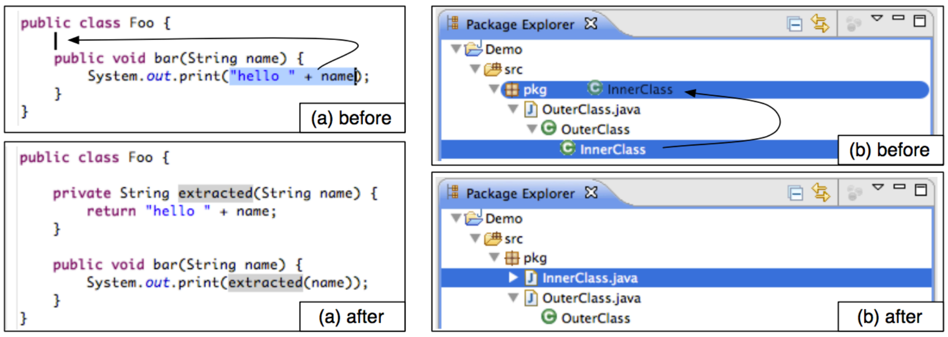
As users interactively edit one of these snippets, Linked Editing simultaneously applies the identical edits to other snippets. CloneTracker takes the output of a clone detector as input and creates a descriptor for each clone [12]. With such descriptors, CloneTracker tracks clones across program versions and identifies any modification to those clones. Similar to Linked Editing, CloneTracker also echoes edits in one clone to other counterparts upon a developer's request. Clever is another clone management system that tracks code clone groups and detects any inconsistent change applied to clones within the same group [41]. If a clone misses the updates applied to the other clones in the same group, Clever automatically suggests the missing update to that clone.

1.2 Refactoring

Code refactoring is the process of restructuring source code without changing its external behaviors. It is usually applied to improve code readability or extensibility, or to reduce code complexity. Fowler et al. defined a catalog of refactorings that developers can apply to improve their codebase in different ways [5], although developers can also define and manually apply their own refactorings. Eclipse IDE also provides tool support to automate some of the refactorings mentioned in Fowler's catalog, such as *Extract Method*, *Pull up Method*, and *Move Field*. Researchers proposed various approaches to automate refactoring or to complete the refactoring tasks initiated by developers [17, 6, 11, 16, 8, 29, 26, 32, 23]. To ensure the semantic equivalence between programs that are before and after a predefined refactoring program transformation, automated tools always check some pre-conditions before applying the transformation, and may check some post-conditions after the transformation. In this section, we will explain two refactoring tools with more details: Drag-and-Drop Refactoring (DNDRefactoring) [29] and R3 [23].

Drag-and-Drop Refactoring. Prior research identified at least three dominant usability problems when using automated refactoring tools [43, 31, 44, 38, 37, 49]. First, programmers have trouble identifying opportunities for using the tools. Second, programmers have difficulty invoking the right refactoring from a lengthy menu of available refactorings. Third, programmers find it complicated to properly configure refactoring dialogs. DNDRefactoring was proposed to facilitate automated refactoring by allowing programmers to apply refactorings through direct manipulation on program elements, including variables, expressions, statements, and methods, in the IDE. In this way, developers do not need menus or dialogs to specify what refactoring to apply or how to apply those refactorings. Instead, DNDRefactoring can automatically infer such information by monitoring the selection and drag-and-drop operations of developers.

Figure 2 presents two scenarios where DNDRefactoring automates refactorings based on the gestures of developers in Eclipse IDE. As shown in Figure 2 (a), when developers select a code snippet in an existing method `bar(...)`, and then drag-and-drop the snippet to a location outside `bar(...)`, DNDRefactoring automates the *Extract Method* refactoring by creating a new method `extracted(...)`, which takes in one parameter: `name`. In this process, no menu or configuration dialog is needed, because the tool can infer developers' refactoring intent, and decide all detailed information required for the refactoring, such as the method name, the method body, and the code location where to place the new method. Figure 2 (b) shows another scenario. When developers manually select the `InnerClass`, a class declared inside another class `OuterClass`, and then drag-and-drop the class to a package `pkg`, DNDRefactoring infers that developers want to extract the type and create a new Java file. Therefore, it generates a Java file InnerClass.java to hold all implementation of `InnerClass`.



```
public class Foo {
    |
    public void bar(String name) {
        System.out.print("hello " + name);
    }
}
```

```
public class Foo {

    private String extracted(String name) {
        return "hello " + name;
    }

    public void bar(String name) {
        System.out.print(extracted(name));
    }
}
```

Fig. 2. DNDRefactoring: Drag-and-drop gestures in (a) Java editor for Extract Method refactoring, and (b) Package Explorer for Extract Type to New File refactoring [29].

Although DNDRefactoring demonstrated effectiveness in simplifying the application of automated refactorings, it still suffers from several limitations. First, not every refactoring can be conducted in a drag-and-drop manner. DNDRefactoring effectively supports move- and extract-based refactorings, but does not support *Rename* refactoring. If developers want to rename a variable or a class, they cannot express that intent via selecting some text and moving it around with the cursor. Second, even for move- or extract- based refactorings, DNDRefactoring mainly works when the drag source and drop target elements are shown in the same screen. It does not work when these elements are located too far away to be presented in the same visual editor simultaneously. Third, DNDRefactoring does not allow users to freely configure refactoring details. For example, when users want to specify a meaningful name of an extracted method, with DNDRefactoring, they have no way to customize the information.

R3. Compared with DNDRefactoring which improves the usability of Eclipse Refactoring with better UIs, R3 presents an alternative refactoring engine that works 10 times faster than Eclipse Refactoring. Specifically, R3 provides refactoring scripts as short Java methods, which enables users to easily define new refactorings. To speed up the application of refactorings, R3 first parses Java sources files, and builds a main-memory, non-persistent database to encode Java entity declarations (e.g., packages, class, methods), their containment relationships, and language features such inheritance and modifiers. By converting program Abstract Syntax Tree (AST) transformations to database queries and manipulations, R3 significantly reduces the runtime overhead of automating refactorings.

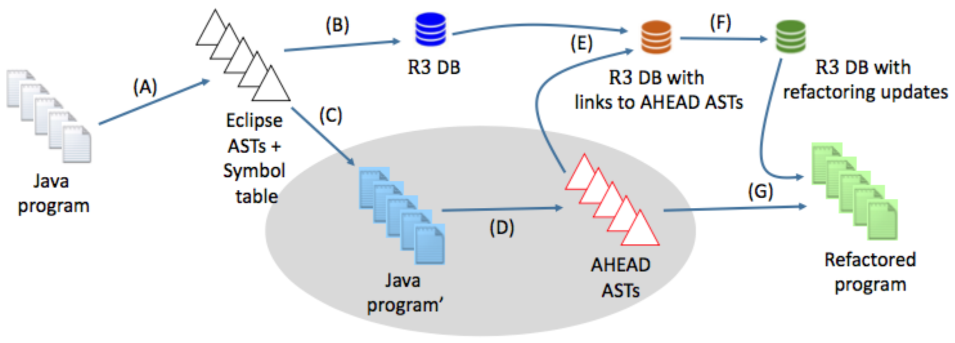

Java program
(A)
Eclipse ASTs + Symbol table
(B)
R3 DB
(C)
Java program'
(D)
AHEAD ASTs
(E)
R3 DB with links to AHEAD ASTs
(F)
R3 DB with refactoring updates
(G)
Refactored program

Fig. 3. R3 pipeline [23].

Figure 3 presents the R3 pipeline, which has a series of stages (A)-(G) that map a target Java program (JDT project) on the left to a refactored program on the right. Stage (A) parses a Java program into ASTs with an Eclipse ASTParser. Stage (B) visits each generated AST to gather all Java entity-relevant information, and to save the information in R3 DB. To prepare program refactorings based on R3 DB, stage (C) generalizes the original program by replacing concrete entity information with abstract symbolic names, with each name corresponding to one concrete entity identifier kept in R3 DB. Stage (D) uses AHEAD [7] to further parse the generalized program representation and create AHEAD ASTs. In Stage (E), tuples in R3 DB are doubly-linked to their AHEAD AST nodes, so that each pretty-printer of an AST node can reference the corresponding R3 tuple and vice versa.

Stage (F) execute R3 refactorings. Unlike classical refactoring engines that modify Abstract Syntax Trees (ASTs), R3 refactorings modify only the database. For instance, if developers want to move a method from one class to another class, R3 does not directly move the method AST. Instead, it simply updates the method's database entry to have the new receiver type. With such information updates to the database, R3 can avoid repetitive AST manipulations. Finally, according to the updated entity information in R3 DB, stage (G) pretty-prints the source code, producing the resulting refactored program.

In addition to converting expensive AST manipulations to cheap database updates, R3 also precomputes the value of many properties of Java entities and saves those values in its database. Since these values can be used int the precondition checks of many refactoring tasks, keeping them in the database can avoid repetitive value computations, which can further reduce runtime overhead. R3 can well handle move- and rename-based refactorings, but does not support extract-based refactorings, because it does not store or modify statement-level or expression-level information in the database.

1.3 Systematic Editing

Systematic editing is the process of applying similar, but not necessarily identical, program changes to multiple code locations. Prior work shows that programmers apply systematic edits to either add features, fix bugs, or refactor code [25, 24, 40]. Manually applying similar but different edits to multiple code locations is tedious and error-prone for two reasons. First, developers may forget to apply systematic edits to all program contexts where the edits are needed, committing errors of omission. Second, developers may apply edits inconsistently and thus introduce new bugs. To improve programmer productivity and software quality, several approaches [34, 33, 45] have been proposed to infer the general program transformation from one or more code change examples provided by developers, and then apply the transformation to other program contexts in need of similar changes. In this section, we will explain one approach with more details—SYDIT [34], and briefly discuss another approach: LASE [33].

SYDIT. When developers want to change multiple code locations in similar but not identical ways, SYDIT requires developers (1) to modify one of these code locations and present it as a code change example, and (2) to manually specify the other locations to change similarly. By inferring a program transformation from the given example, SYDIT can apply the transformation to those locations and automate systematic editing accordingly.

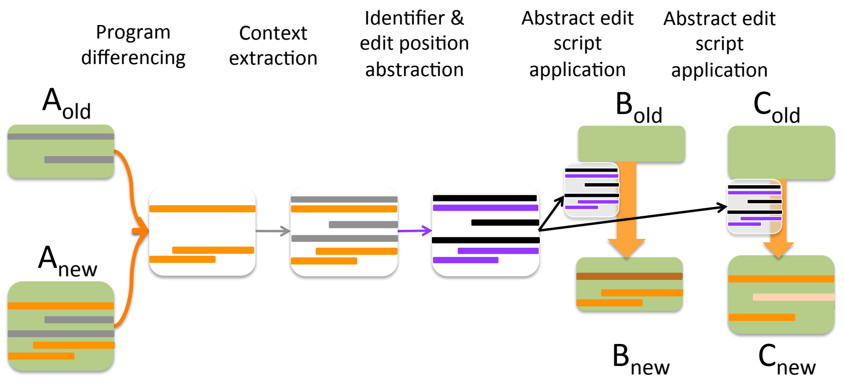

Program differencing
Context extraction
Identifier & edit position abstraction
Abstract edit script application
Abstract edit script application
A_{old}
A_{new}
B_{old}
B_{new}
C_{old}
C_{new}

Fig. 4. SYDIT overview.

As shown in Figure 4, suppose developers want to apply systematic edits to three methods: A, B, and C. Without any tool support, developers have to manually apply these similar but different edits to all three locations. With SYDIT, developers only need to (1) modify one location (e.g., A), and (2) manually specify the other locations to change (e.g., B and C). With the user input, SYDIT takes four steps to generalize and apply a program transformation. In the first step (program differencing), given the old and new version of method A, SYDIT first leverages ChangeDistiller [15] to compare the ASTs of both versions, and to extract changes as an AST edit script. The edit script may involve four types of statement-level edit operations: insert, delete, update, and move. To generalize the concrete edit script to a program transformation that is applicable to other program contexts, SYDIT needs to abstract the edit context and concrete identifiers. In step 2, SYDIT abstracts the edit context by identifying all unchanged code that is either control- or data-dependent on by the edited code, because such unchanged code manifests the semantic constraints those applied edits put on the program context. In step 3, SYDIT also abstracts the concrete identifiers used in the demonstrated edit so that the general program transformation is also applicable to code snippets that use different identifiers. Besides, SYDIT abstracts all edit operations' positions with respect to the extracted unchanged code, so that the inferred program transformation describes each edit operation within the edit-relevant context. In this way, SYDIT derives an abstract, context-aware edit script from a given code change example.

In step 4, SYDIT concretizes the inferred program transformation to given code locations B and C, and applies the customized edits to suggest new versions of code. This step is the reverse process of the steps 1-3 mentioned above. With more details, given a code location to change (e.g., B), SYDIT first looks for any matching for the abstract edit context. If no matching is found, it indicates that similar changes should not be applied to the selected location, because the semantic constraints embedded in the abstract context are not satisfied. If one matching is found, SYDIT further concretizes the identifier usage and edit location information in the abstract script, creating a customized edit script applicable to the specific location. Finally, by modifying the code's AST based on the script, SYDIT produces a revised program for developers to review.

Although SYDIT can automate systematic editing in specified code locations, it always relies on users to manually pick those locations. In reality, finding code locations may be more challenging than applying the edits, especially when the codebase is large, and developers lack expertise of the project. Additionally, one code change example may not precisely characterize all program contexts which should be changed similarly. If the single example contains some edit-relevant context that is unique to the code location, SYDIT has no clue about how to only generalize the edit-relevant context which is commonly shared among all code locations.

LASE. To facilitate systematic editing when developers cannot manually identify all code locations that should be changed similarly, LASE requires developers to provide at least two code change examples, from which it infers a general program transformation, and then leverages the transformation to both find other edit locations and suggest customized edits.

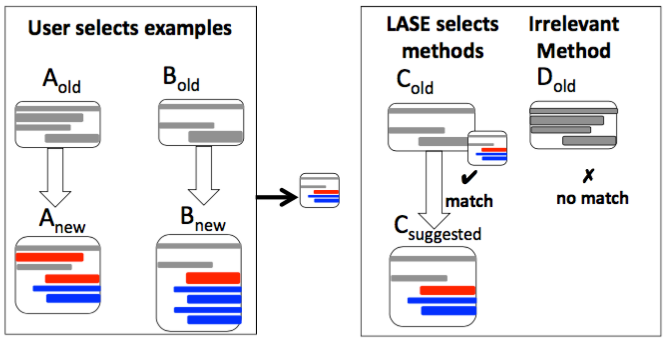


Fig. 5. LASE overview.

As shown in Figure 5, given two changed examples: A and B, LASE infers a general program transformation for each example, and then extracts the largest commonality between them in terms of edit operations, edit-relevant contexts, and identifier usage. In this way, LASE filters out any location-specific edit information, and only generalizes the program transformation that occurs multiple times. With such inferred program transformation, LASE attempts to establish matching between the abstract edit context and every method in the whole codebase. If a method contains a matching to the given context (e.g., C), LASE recommends the method as a candidate edit location, and suggests customized edits according to the matching information. If a method does not contain any matching to the given context (e.g., D), LASE considers the method as an irrelevant method to the systematic editing task.

References

1. COBOL Migration to Java or .NET. `http://www.semdesigns.com/Products/Services/COBOLMigration.html?Home=LegacyMigration`.
2. HTML5. `https://www.w3.org/TR/html5/`.
3. JOVIAL2C Translator. `http://www.semdesigns.com/Products/MigrationTools/JOVIAL2C.html`.
4. PhoneGap. `https://build.phonegap.com`.
5. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
6. M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, page 326, 1999.
7. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.
8. N. Chen and R. E. Johnson. Jflow: Practical refactorings for flow-based parallelism. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 202–212, Nov 2013.

9. J. Chu and T. Dean. Automated migration of list based jsp web pages to ajax. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 217–226, Sept 2008.
10. J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
11. D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: Refactoring for loop parallelism in java. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 793–794, New York, NY, USA, 2009. ACM.
12. E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.
13. M. El-Ramly, R. Eltayeb, and H. A. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, 2006.
14. J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.*, 50(6):229–239, June 2015.
15. B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.
16. X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press.
17. W. G. Griswold. *Program Restructuring As an Aid to Software Maintenance*. PhD thesis, Seattle, WA, USA, 1992. UMI Order No. GAX92-03258.
18. S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.
19. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.
20. S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 50–61, New York, NY, USA, 2011. ACM.
21. W. R. Harris and S. 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.
22. A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 2005.
23. J. Kim, D. Batory, D. Dig, and M. Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1145–1156, New York, NY, USA, 2016. ACM.
24. M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.
25. M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, Sept. 2005.

26. G. P. Krishnan and N. Tsantalis. Refactoring clones: An optimization problem. *ICSM*, 0:360–363, 2013.
27. 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.
28. T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. *Learning repetitive text-editing procedures with SMARTedit*, pages 209–226. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
29. Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 23–32, Piscataway, NJ, USA, 2013. IEEE Press.
30. H. Lieberman, editor. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann Publishers, 2001.
31. E. Mealy, D. Carrington, P. Strooper, and P. Wyeth. Improving usability of software refactoring tools. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 307–318, April 2007.
32. N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 392–402, Piscataway, NJ, USA, 2015. IEEE Press.
33. N. Meng, M. Kim, and K. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *ICSE '13: Proceedings of 35th IEEE/ACM International Conference on Software Engineering (Accepted)*, page 10 pages. IEEE Society, 2013.
34. N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 329–342, New York, NY, USA, 2011. ACM.
35. R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.
36. M. Mossienko. Automated Cobol to Java recycling. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, 2003.
37. E. Murphy-Hill, M. Ayazifar, and A. P. Black. Restructuring software with gestures. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 165–172, Sept 2011.
38. E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 421–430, New York, NY, USA, 2008. ACM.
39. A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.
40. T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324, New York, NY, USA, 2010. ACM.

41. T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2009. IEEE Computer Society.
42. R. 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.
43. A. O'Connor, M. Shonle, and W. Griswold. Star diagram with automated refactorings for eclipse. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '05, pages 16–20, New York, NY, USA, 2005. ACM.
44. C. Parnin, C. Görg, and O. Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 77–86, New York, NY, USA, 2008. ACM.
45. R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press.
46. H. M. Sneed. Migrating from COBOL to Java. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010.
47. P. Tonella and M. Ceccato. Migrating interface implementations to aspects. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 220–229, Sept 2004.
48. M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.
49. M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243, June 2012.
50. I. H. Witten and D. Mo. *TELS: learning text editing tasks from examples*, pages 183–203. MIT Press, Cambridge, MA, USA, 1993.
51. N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 508–521, New York, NY, USA, 2016. ACM.
52. K. Yasumatsu and N. Doi. SPiCE: a system for translating Smalltalk programs into a C environment. *IEEE Transactions on Software Engineering*, 1995.