Supporting Selective Undo for Refactoring

Xiao Cheng*, Yuting Chen*, Zhenjiang Hu†, Tao Zan‡, Mengyu Liu*, Hao Zhong* and Jianjun Zhao*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†National Institute of Informatics, Japan
‡The Graduate University for Advanced Studies, Japan
{x.cheng, chenyt, liumengyu, zhonghao, zhao-jj}@sjtu.edu.cn, {hu, zantao}@nii.ac.jp

***Abstract*—Due to various considerations, programmers often need to backtrack their code. Furthermore, as the most recent edit may not be the wrong edit, programmers sometimes have to backtrack their code for arbitrary edits, which is referred as a *selective undo* in this paper. To meet the needs, researchers have proposed various approaches to support selective undo. However, to the best of our knowledge, these approaches can support only simple edits, and cannot handle refactoring, although most code editors already provide various refactoring actions. Indeed, it is challenging to support selective undoes for refactoring, since multiple code elements and complicated actions can be involved. In this paper, we present a novel approach that leverages Bidirectional Transformation (BX) to support selective undoes for refactoring. We evaluate our approach on a recent refactoring tool that transfers enhanced `for` loops to lambda expressions. Our results show that our approach achieves success ratio of up to 89%.**

I. Introduction

As human beings, programmers can make mistakes when coding, so as shown in an empirical study [30], programmers often need to backtrack their code. To meet the need, most code editors support undoing from the most recent edit, which is called a *linear undo*. Furthermore, as a mistake may not be the most recent edit, programmers sometimes have to conduct a *selective undo* for arbitrary edits, while preserving the follow-up edits after the mistake [2]. To support selective undo, Yoon and Myers [31] proposed AZURIE that records edits (*i.e.*, *insert*, *delete*, and *replace*), and allows programmers to fix conflicts after conducting a selective undo. However, despite their positive evaluation results, their tool does not support selective undo for refactoring.

Program refactoring has attracted great attention from both industry and academia [8], [18], [21], and most advanced code editors (*e.g.*, Eclipse [**?**] and Visual Studio [**?**]) support various refactoring actions. Compared with simple edits, refactorings involve more code elements and more actions. As a result, it becomes more challenging to support a selective undo for refactorings. For example, Fig. 1a shows a piece of sample code. It includes an enhanced `for` loop that computes the sum of primes, and it uses EclipseLink [**?**] to persist the class. Gyori *et al.* [9] propose a refactoring tool that refactors enhanced `for` loops into lambda expressions. Suppose that a programmer named Mary is maintaining the sample code, and knows the above refactoring tool. She may try the tool, and it refactors the sample code into a lambda expression as shown in Fig. 1b. After that, she may further insert a `filter` and update a parameter name as shown in Fig. 1c. Later, she reads a discussion on Stackoverflow [**?**], and realizes that EclipseLink does not support lambda expressions. As a result, she has to backtrack her code. If she relies on the linear undo provided by existing code editors, she has to give up all the edits after the refactoring action. If she tries the existing selective-undo tool [31], she will soon find that the tool does not support selective undo for refactoring actions. Our approach allows Mary to undo the refactored code while preserving the follow-up edits as shown in Fig. 1d. To support such an undo, we have to overcome the following challenge:

Challenge 1. Compared to simple edits such as *insert*, *delete*, and *replace*, refactoring can involve more code elements and more actions. As a result, the refactored code can be quite different from the original code. In such cases, it becomes difficult to align code elements and to conduct undo correctly.

As a complete concept, our approach has to support both undo and redo actions for refactoring, which needs to overcome two additional challenges:

Challenge 2. Typically, undo and redo actions alone shall not result in different code, which is called the *round-trip property*. It is difficult to keep the round-trip property, especially when refactoring involves multiple code elements and actions.

Challenge 3. Typically, undo and redo actions shall not introduce compilation errors. As refactoring involves multiple code elements and actions, it can introduce complication errors, especially when a piece of code is modified after refactoring.

To address the three challenges above, we borrow the idea of the putback-based updating technique [1], [6], and propose an approach that supports selective undo and redo for refactoring. In particular, our approach includes a bidirectional transformation (*BX*) engine to ensure the round-trip property, and it carefully defines putback-based update declarations to reduce ambiguity in the traditional bidirectional transformation. This paper makes the following contributions:

- We make the first attempt of *formalizing* selective undo and redo for refactoring as a structure-preserved putback-based updating problem, and solve this nontrivial problem with the emerging bidirectional transformation [33]. Compared to the traditional bidirectional approaches [28], [32], our approach is more predictable, especially for complicated mapping relations.
- We propose an approach that supports selective undo for refactoring. With predefined mapping relations between original code and refactored code, our *core putback-based update algorithm* automatically propagates edits

```java
@Entity
public class Prime {
  @Id
  private Integer num;
  public int primeSum(String[] numbers){
    List<String> l = Arrays.asList(numbers);
    int sum = 0;
    for(String e : l){
      Integer n = Integer.valueOf(e);
      if(Primes.isPrime(n))
      sum += n;
    }
    return sum;
  }
}
```

(a) cf_1

```java
@Entity
public class Prime {
  @Id
  private Integer num;
  public int primeSum(String[] numbers){
    List<String> l = Arrays.asList(numbers);
    int sum = 0;
    sum = l.stream()
      .map(e -> Integer.valueOf(e))
      .filter(n -> Primes.isPrime(n))
      .reduce(sum, (x,y) -> x+y);
    return sum;
  }
}
```

(b) cf_2

```java
@Entity
public class Prime {
  @Id
  private Integer num;
  public int primeSum(String[] numbers){
    List<String> l = Arrays.asList(numbers);
    int sum = 0;
    sum = l.stream()
      .filter(e -> e != null)
      .map(e -> Integer.valueOf(e))
      .filter(newParameter -> Primes.isPrime(
          newParameter))
      .reduce(sum, (x,y) -> x+y);
    return sum;
  }
}
```

(c) cf_2'

```java
@Entity
public class Prime {
  @Id
  private Integer num;
  public int primeSum(String[] numbers){
    List<String> l = Arrays.asList(numbers);
    int sum = 0;
    for(String e : l){
      if(e != null){
        Integer newParameter = Integer.valueOf(e);
        if(Primes.isPrime(newParameter))
          sum += newParameter;
      }
    }
    return sum;
  }
}
```

(d) cf_1'

Fig. 1. A refactoring example

on refactored code back to original code. Moreover, with more mapping relations, our *core putback-update algorithm* can be easily extended to support more types of refactoring actions.

- We conduct a proof-of-concept experiment to evaluate the usefulness of our approach. In particular, we define the mapping relations for the refactoring from enhanced `for` loops to lambda expressions [9], and implement our approach in BiFluX [22]. Our results show that our tool achieves success ratio of up to 89% on 200 instances of selective undo for the refactoring.

The remainder of the paper is organized as follows: Section II uses a simple example to illustrate the main idea of selective undo for refactoring. Section III formalizes selective undo and redo for refactoring as a putback-based updating problem. Section IV presents the main steps of our approach. Section V evaluates our approach. Section VII presents related work, and Section VIII concludes this paper.

II. An Illustrative Example

We next use an example to illustrate the requirements of selective undo for refactoring, and explain the essential idea of our approach.

First, let us revisit the refactoring that is proposed by Gyori *et al.* [9]. Let a code fragment before refactoring be a *source* code fragment and that after refactoring a *target* one. The source code fragment (*i.e.*, cf_1 in Fig. 1a) contains an enhanced `for` loop that takes an array of strings as input, converts each to an integer, and then sums up all of the primes. The source code fragment is refactored to a highly abstract target code fragment (*i.e.*, cf_2 in Fig. 1b) which contains a `Collection` iterator with functional operations.

After a piece of code is refactored, programmers can make more edits on the refactored code. For example, let cf_2' in Fig. 1c be edited from cf_2 by taking two editing operations: (1) inserting an operation filtering the non-null elements from the stream in Line 9 and (2) changing the parameter name `n` to `newParameter` in Line 11. In such cases, if programmers try to undo the refactoring action, we have to revise cf_1 appropriately to preserve the follow-up edits. Meanwhile, after conducting the undo, if programmers modify cf_1 and try to redo the refactoring action, we have to revise cf_2 appropriately to preserve the edits after the undo.

In the above selective undo and redo for refactoring, our approach must meet two requirements:

- *Semantic equivalency*. As refactoring does not change

semantics, it requires that after conducting selective undo or redo, the semantics remain the same, while preserving the follow-up edits.
- *Syntactic traceability*. It requires that the code elements of the original code and the refactored code shall be traceable so that the follow-up edits can be transferred to corresponding edits, when an undo or redo is conducted.

To support selective undo for refactoring, our approach has two major steps:

1) For each refactoring, our approach records the original code (cf_1 in Fig. 1a) and the refactored code (cf_2 in Fig. 1b), and encodes them into an XML file.
2) When programmers undo a refactoring action, our approach transfers the code structure of cf_2 to that of cf_1 according to the encoded XML file, and transfers the follow-up edits according to our putback-based update algorithm. In particular, our approach puts the insertion of `filter` in Fig. 1c to an `if` statement in Line 9 of Fig. 1d, and puts the changed parameter in Fig. 1c to its corresponding construct, *i.e.*, replacing `n` in Fig. 1a with `newParameter` to produce cf_1' in Fig. 1d.

III. Formalization

We formalize the selective undo and redo for refactoring as a structure-preserved putback-based updating problem. We first abstract code fragments as one or more program constructs, on which the follow-up edit operations can be propagated from refactored code fragment to the original one (undo), and vice versa (redo).

Definition 1: A **program construct** is captured by its structure defined as follows:

Atomic structures	$\alpha ::= string \mid n[\tau]$
Sequence structures	$\tau ::= \alpha \mid () \mid \tau \diamond \tau' \mid \tau, \tau' \mid \tau^\star$

In Definition 1, *Atomic structures* $\alpha \in Atom$ are primitive strings or labeled sequences *n[τ]*. *Sequence structures* $\tau \in Seq$ are defined with regular expressions such as an empty sequence *()*, an alternative choice $\tau \diamond \tau'$, a sequential composition τ, τ', and an iteration $\tau^\star$. The choice and composition are right-nested. In addition, we define $\tau^+ = \tau, \tau'$ and $\tau^? = \tau \diamond ()$.

By the definition, a program construct can be a text, a composition of program constructs, or a mix of all above.

If a program construct contains no program constructs, we call it atomic program construct, otherwise compound program construct. An atomic program construct can be a simple statement, an expression, or a variable. A compound program construct can be a block, a method, or a class. For example, cf_1 in Fig. 1a is composed of a method, and the method is a compound program construct that is composed of several statements. Among the statements, the `for` loop is also a compound program construct, while the expressions and some of the statements in the enhanced `for` loop are treated as atomic program constructs.

We take each code fragment as a compound program construct, therefore, the structures of the program constructs represent those of the code fragments. We use S to denote the abstract syntax structure of a code fragment.

Let CF be a set of code fragments, and CF_1 be a set of programs whose structures are S_1:

$$CF_1 := \{cf \in CF \mid cf : S_1\}$$

Let CF_2 be a set of programs whose structures are S_2:

$$CF_2 := \{cf \in CF \mid cf : S_2\}$$

Let $R : CF_1 \times CF_2$ be the refactoring relation between CF_1 and CF_2.

If there exists $(cf_1, cf_2) \in R$, the **selective undo for refactoring** from cf_1 to cf_2 is to produce $cf_1' \in CF_1$ when cf_2 is edited to $cf_2' \in CF_2$, so that $(cf_1', cf_2') \in R$; the **selective redo for refactoring** from cf_1 to cf_2 is to produce $cf_2' \in CF_2$ when cf_1 is edited to $cf_1' \in CF_1$, so that $(cf_1', cf_2') \in R$.

We formalize the two general requirements in Section II by defining the next three properties. When the selective undo or redo for refactoring needs to be conducted, these properties should be kept.

1) *Structure-preserved property.* During the process of the $undo$ or $redo$, cf_1 and its edited version (*i.e.*, cf_1') need to keep the structure (say S_1). Similarly, cf_2 and its edited version (*i.e.*, cf_2') need to keep the structure (say S_2). That is,
$$\frac{(cf_1, cf_2) \in R \quad (cf_1', cf_2') \in R}{cf_1 : S_1 \implies cf_1' : S_1 \quad cf_2 : S_2 \implies cf_2' : S_2} \tag{1}$$
2) *Round-trip property.* It denotes that (1) if no edit is made to refactored code fragment (*i.e.* cf_2), no edit should be made to original code fragment (*i.e.*, cf_1) when the $undo$ is conducted, or (2) if refactored code fragment (*i.e.* cf_2) is edited to cf_2', after the $undo$ is conducted, these edits need to be reflected in original code fragment (*i.e.*, cf_1) to get cf_1', and once the $redo$ in conducted, cf_2' should be reproduced again. That is,
$$\frac{(cf_1, cf_2) \in R}{Undo(cf_1, cf_2) = cf_1} \quad \frac{Undo(cf_1, cf_2') = cf_1'}{Redo(cf_1') = cf_2'} \tag{2}$$
3) *Determinacy.* It denotes that a unique result needs to be given in order to eliminate any ambiguity when either $undo$ or $redo$ is conducted, *i.e.*,
$$\frac{\frac{(cf_{1a}', cf_2') \in R \quad (cf_{1b}', cf_2') \in R}{cf_{1a}' = cf_{1b}'} \quad \frac{(cf_1', cf_{2a}') \in R \quad (cf_1', cf_{2b}') \in R}{cf_{2a}' = cf_{2b}'}}{} \tag{3}$$

IV. Approach

We borrowed the *putback-based* updating technique from bidirectional transformation (*BX*) to implement selective undo and redo for refactoring through building the mapping relationship between program constructs. In this section, we first give a brief introduction to *BX* and its putback-based updating technique and then present the technical details of the selective undo and redo for refactoring.

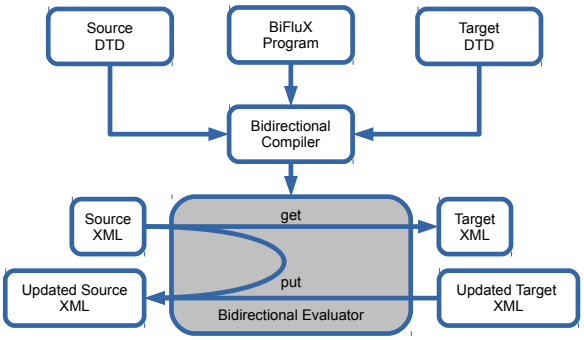

Fig. 2. Framework of BiFluX

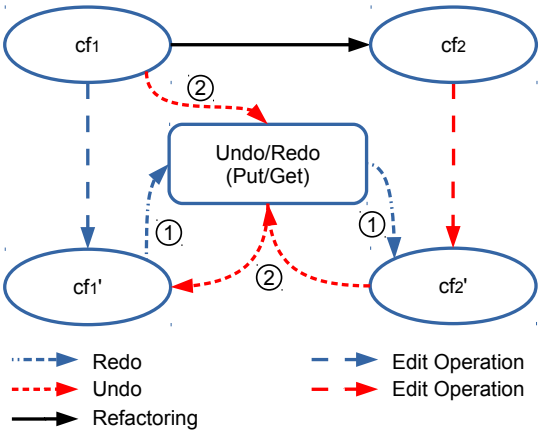

Fig. 3. Overview of putback-based selective undo and redo

A. Bidirectional Transformation

The bidirectional transformation consists of a pair of functions ("*get*" and "*put*") which plays an important role in maintaining consistency of two related models. Function "*get*" maps from a concrete *source* model to an abstract *target* model. Function "*put*" reflects the modifications on the abstract *target* model back to the concrete *source* model.

$$get : S \rightarrow T$$

$$put : S \times T \rightarrow S$$

As it is error-prone to write and maintain the two functions (*i.e.*, *get* and *put*) manually, many researchers have proposed various approaches that aid the programming of bidirectional transformations. For example, Foster *et. al* [6] propose lenses for synchronizing tree-structured data. Lenses consist of a list of primitive composable lenses, where each one is elaborately designed with *get*, *put* semantics, and lens composition operator. Those primitives can be composed using lens composition to accomplish large programs to synchronize complex tree-structured data. Hidaka *et. al* [10] propose GroundTram which gives a bidirectional semantics (*put* semantics) of an existing graph query language UnQL to make it workable on graphs, which is useful in model-driven development.

While normally abstract model only contains part of the source information, and the correct "*put*" is not unique. The proposed approaches above give a fixed "*put*" semantics which may not be applicable in real usage, *e.g.*, the *get* function does not know how the refactored code fragment will be edited while the edit operation on the refactored code fragment needs to be reflected to that before refactoring through the *put* function. therefore Hu *et. al* [33], [23] propose a putback-based bidirectional transformation language called BiFluX for XML data, which lets the programmer write a *put* function and derives a unique *get* function automatically.

Figure 2 shows the framework of BiFluX. Both source and target data are represented in XML and conform to the corresponding Document Type Definitions (DTDs); the DTDs are used for validating the corresponding XML before BX program is executed and also guiding the construction of the update program; a BiFluX update program needs to be written by programmers for describing how to update a source XML according to a target XML by inferring possible updates on the target and reflect them back to the source with the specified update strategy.

Based on the putback-based bidirectional transformation techniques [23], [22], we know that if the *put* function satisfies the two properties: "*put s*" is injective and putting t' twice on s produces the same s', the following propositions are also satisfied:

1) There exists a get function such that $get(s) = t$ and $put(s, t) = s$ are well-behaved [10].
2) The get function in 1) is the only one such that get and put are well-behaved.

B. Overview of Putback-based Selective Undo and Redo

We solve selective undo and redo for refactoring under the framework of bidirectional transformation. The original code fragments before and after refactoring (*e.g.*, cf_1 and cf_2) are treated as source model and target model, respectively. We conduct undo and redo by the two functions *put* and *get*, respectively. We employ BiFluX to support the bidirectional transformation mechanism for the consideration that it provides a simple way to define the relations between source and target models and provides the mechanism that checks whether the structures of code fragments satisfy a certain structure template to ensure the syntactical correctness.

Figure 3 shows an overview of putback-based selective undo and redo for refactorings. cf_1 and cf_2 are two code fragments, where cf_2 is refactored from cf_1. The code fragment cf_2' is edited from cf_2. Our approach implements undo by defining the *put* function which embeds program constructs in cf_2' into cf_1 to generate cf_1', and redo is conducted through the derived *get* function from *put* function automatically. Once cf_1 is edited to cf_1', the *get* function will extract cf_2' from cf_1'.

For different refactorings, the structures of code fragments are quite different, and there exist different mapping relations

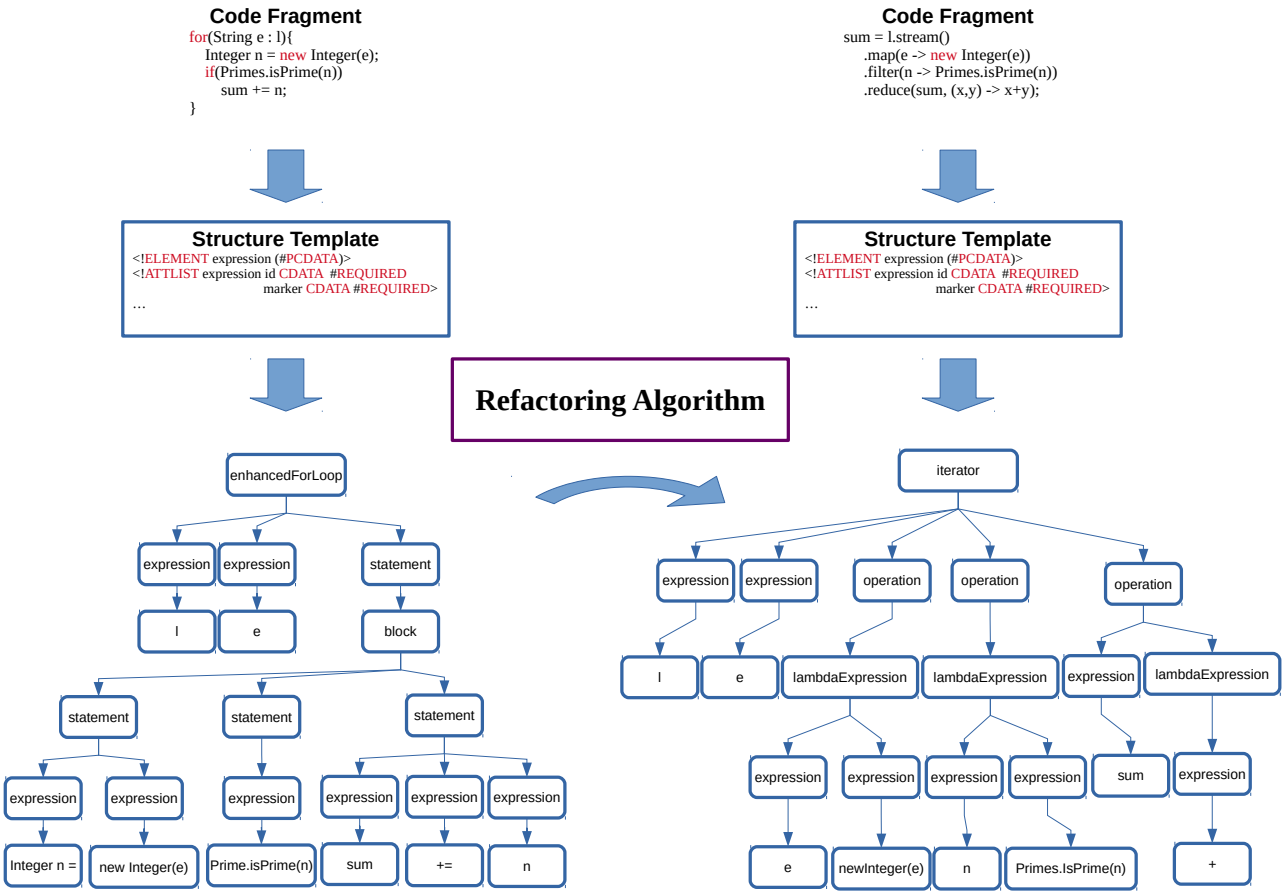

Fig. 4. Structure of code fragments

between the program constructs of code fragments before and after refactorings. The *put* and *get* functions are also varied, but the major steps are as follows:

1) Defining the code structure templates of code fragments before and after refactoring (Section IV-C).
2) Generating functions undo and redo (Section IV-D).
3) Performing undo or redo (Section IV-E).

Since the selective undo and redo are implemented through *put* and *get* respectively, we make no differences between undo and *put*, redo and *get* in the remainder of the paper.

C. Define the Structure Templates of Code Fragments

Refactoring algorithms tell us what kind of code fragments will be refactored and what kind of code fragments will be generated. In other words, we know the structures of the code fragments before and after refactoring and use a template, which is called structure template, to describe the structure.

The structure templates are defined through assigning the atomic program construct and compound program construct of the code fragments. For example, each leaf node in the AST of the code fragment can be assigned as an atomic program construct. However, it is not always necessary to assign such fine-grained program construct. For particular refactoring algorithm, some coarse-grained atomic program constructs (*i.e*, expression and statement) are preferred. The grain of the atomic program construct is determined by the grain of code edited. After the atomic program constructs are defined, the compound program constructs will be defined by composing atomic program constructs. From the definition of the structure template, we can see that the code fragments are represented as trees.

Each time the undo or redo is conducted, the structure of the code fragments will be checked. If it satisfies the structure template, the undo or redo will continue, otherwise, the undo or redo will abort and a notification will be sent to the user. As Fig. 4 shows, the code fragments are transformed to tree structure checked by the structure template.

Let us revisit the example in Section I. In this example we define the structure template of enhanced `for` loop and `Collection` iterator with functional operations as follows: The enhanced `for` loop consists of two expressions, which denote the `Collection` and its representative element respec-

TABLE I
STRUCTURE TEMPLATES OF FOR LOOP AND ITERATOR

Program Construct	Structure Template
expression	`<!ELEMENT expression (#PCDATA)>` `<!ATTLIST expression id CDATA #REQUIRED marker CDATA #REQUIRED>`

statement	`<!ELEMENT statement (expression+ | block)>`
	`<!ELEMENT block (statement*)>`
	`<!ATTLIST statement id CDATA #REQUIRED marker CDATA #REQUIRED>`
	`<!ATTLIST block id CDATA #REQUIRED marker CDATA #REQUIRED>`

enhanced for loop	`<!ELEMENT enhancedForLoop (expression, expression, statement)>` `<!ATTLIST enhancedForLoop id CDATA #REQUIRED marker CDATA #REQUIRED>`
λ expression	`<!ELEMENT lambdaExpression (expression, expression)>` `<!ATTLIST lambdaExpression id CDATA #REQUIRED marker CDATA #REQUIRED>`
operation	`<!ELEMENT operation (expression, expression?, lambdaExpression)>` `<!ATTLIST operation id CDATA #REQUIRED marker CDATA #REQUIRED>`
iterator	`<!ELEMENT iterator (expression, operation+)>` `<!ATTLIST iterator id CDATA #REQUIRED marker CDATA #REQUIRED>`

TABLE II
ALIGNED STATEMENTS AND OPERATIONS

Statement	Operation
`Integer n = new Integer(e);`	`map(e -> new Integer(e))`
`if(Primes.isPrime(n))`	`filter(n -> Primes.isPrime(n))`
`sum += n;`	`reduce(sum, (x,y) -> x+y);`

tively, and a statement. An expression is an atomic program construct represented by a string, while the statement is a compound program construct that consists of either expressions or a block. Similarly, the structure template of `Collection` iterator is defined based on expressions and statements. Table I shows the structure template of the enhanced `for` loops and `Collection` iterator that represented as DTD.

As BiFluX supports the transformation of files in XML, we define a language based on srcML [5] to represent code fragment. Here, srcML is an extended XML language for representing code. In practice, DTD is widely used to define structures, legal elements, and attributes of XML documents. As we define code in a srcML-based language, the code structures are specified by DTD. In particular, a program construct is treated as an element in srcML. The element name denotes the name of a construct; the content of an element includes more detailed elements or atomic constructs in the text format; the attributes (or wrapped ones) denote the attributes of a program construct.

After defining the structure template of code fragments, the program constructs in the source code fragment and the target code fragment should be aligned so that once one is edited, we know which one should be modified accordingly.

The mapping relation between two compound program constructs is composed by a set of mapping relations between their sub-constructs, which can be established when a code fragment is refactored to another. As Fig. 4 shows, the refactoring algorithm plays a role in aligning the program constructs. Let a construct $c_{2j} \in C_2$ be refactored from $c_{1i} \in C_1$. A mapping $c_{1i} \rightarrow c_{2j}$ is constructed between the two constructs, which means c_{1i} and c_{2j} are aligned. The relations between program constructs can be built iteratively until no relations can be found. We set each pair aligned program constructs a marker, through which a program construct can be easily referred and matched with another one.

The alignment between the two compound program constructs is represented as a set.

$$C_1 \times C_2 : \{(c_{1i}, c_{2j}) \mid c_{1i} \in C_1, c_{2j} \in C_2, c_{1i} \rightarrow c_{2j}\}$$

For example, the `Collection` and its representative element in the enhanced `for` loop can be aligned to those in the iterator, and a statement in `for` loop can be aligned with a functional operation in the iterator if their composed expressions are aligned. *LambdaFicator* [9] transforms all statements (but if statements) to `map` operations by default. For `if` statements which have no `else` branch and no statements after them, they are transformed to `filter` operations. Finally, the last statement is transformed to an eager operation. Therefore, the statements and operations in the example are aligned as Table II shows.

D. Generating Undo and Redo Functions

We reduce the problem of selective undo and redo for refactoring to traditional data transformation, through defining the structure template of both source and target code fragments and aligning the corresponding program constructs.

For a particular code refactoring, an undo function need to be designed to describe how to utilize the edited program constructs in the target code fragment to update its aligned ones in the source code fragment, such that the new source code fragment (undo result) preserves the follow-up edit operation after refactoring. Algorithm 1 presents a generic undo function which embeds constructs of edited target code fragment ($target'$) in the source code fragment $source$. The undo function is defined to handle the aligned constructs in $source$ with those in $target'$ followed by an activity of generating new $source$:

1) If c_{2j} in $target$ is edited, the aligned program construct in the code needs to be edited (lines 8-14);
2) If no program construct in $target$ is aligned with a construct (say c_{1i}) in the $source$, c_{1i} needs to be removed from the $source$(lines 3-4);

Algorithm 1 ***Undo***

Input:
A program construct for source C_1: $\{c_{11}, \ldots, c_{1n}\}$
A program construct for edited target C_2': $\{c_{21}, \ldots, c_{2m}\}$
Output:
A program construct for new source C_1': $\{c_{11}', \ldots, c_{1n}'\}$

```
1: for ⟨c_1i ∈ C_1, c_2j ∈ C_2⟩ in aligned construct pairs do
2:     if c_1i ≠ ϵ && c_2j = ϵ then
3:         delete c_1i;
4:     else if c_1i = ϵ && c_2j ≠ ϵ then
5:         create c_1t ∈ C_1 st. ⟨c_1t, c_2j⟩ in aligned pairs;
6:         goto Line 9;
7:     else if c_1i ≠ ϵ && c_2j ≠ ϵ then
8:         if c_1i is atomic && c_2j is atomic then
9:             c_1i ← c_2j
10:        else
11:            Undo(c_1i, c_2j)
12:        end if
13:    end if
14: end for
```

Algorithm 2 ***Redo***

Input: A program construct for source C_1.
Output: A program construct for target C_2.

```
1: if C_1 is atomic then
2:     C_2 ← C_1
3: else if C_1 is composed of { c_11, c_12, ··· , c_1n} then
4:     {c_21, c_22, ··· , c_1m} ← Redo({c_11, c_12, ··· , c_1n}),
   where c_2j = redo(c_1i);
5:     C_2 ← {c_21, c_22, ··· , c_1m}
6: end if
```

3) If a program construct c_{2j} is newly inserted into $target$, a corresponding program construct needs to be generated and inserted into $source$ (lines 5-7).

We find that the undo function defined based on Algorithm 1 satisfies the two properties referred in Section IV-A. With BiFluX, the unique redo function will be derived automatically such that redo and undo are well behaved. Although the derived *get* function are binary code in BiFluX, for convenience, we recover it in the pseudo code as Algorithm 2 shows. While it is impossible to derive a redo function from arbitrary the redo function, the syntax of BiFluX is designed in order to make a given program derivable. A subset of XPath is used to retrieve information from a specific location in an XML, and the bidirectional semantics are carefully designed to remove ambiguity; XQuery expressions are employed to represent XML data. Evaluation and inverse computation of expression are designed. Pattern matching on source and target are supported to decompose them into small components; An alignment operator is designed to handle updating a source list of elements with a target list of elements. All of these are composed together to construct a BiFluX program and it is guaranteed to be correct respect to the bidirectional transformation properties.

TABLE III
PROJECT LIST.

Project	#For Loops	#SLOC of For Loops	#Operations
ANTLR	4	46	3.25
FitNesse	2	23	2
Hadoop	3	30	2.33
Tomcat	2	22	2.5
jEdit	2	41	3.5
FindBugs	2	34	3.5
jUnit	2	30	3
Soot	3	43	3.33
Total	20	269	2.95

E. Performing Undo and Redo

After the undo and redo functions are derived, the refactoring can be selectively redone or undone automatically. In particular, if cf_1 is edited to cf_1', the redo function is called to produce cf_2'; if cf_2 is edited to cf_2', the undo function is called to produce cf_1'. Since the redo and undo functions are well-behaved, the round-trip property referred in Section III is ensured.

V. EVALUATION

In order to evaluate our approach, among various refactoring techniques [18], we choose *LambdaFicator* to perform refactoring in our experiment, since it is complicated refactoring and easy to have edit operation conflict with the refactoring when editing the refactored code. It will be inspiring if our approach works well on undo and redo for such kind complicated refactoring. We conduct the experiment, and try to answer the following two research questions:

1) How effectively does our approach support selective undo and redo for refactoring?
2) How much effort is saved compared with using a linear undo and redo mechanism in the text editor?

In Section V-B, the experiment result shows that our approach achieves 72.5% and 89% success ratio of redo and undo, respectively. Through the failure cases, we summarize two findings. In Section V-C, the result shows that our approach saves a lot of effort compared with using a linear undo and redo mechanism in the text editor.

A. Experimental Setup

We selected eight open source projects, which contain a lot of enhanced `for` loops, as the subject projects, and randomly selected several `for` loops from each project.

Table III shows the statistic of the selected projects. Column "#For Loops" lists the number of selected enhanced `for` loops. Column "#SLOC of Fors" lists lines of `for` loop code. For each selected `for` loop, we use *LambdaFicator* [9] to produce the `Collection` iterator with functional operations. The produced iterator is in the same manner and is of the same functionality as the original code. Column "#Operation" lists the average number of functional operations in the refactored code. In total, the average #SLOC of the loops is 13.45, and the average number of functional operation is 2.95. By

the definition of *LambdaFicator*, 39 out of the 59 (66.1%) functional operations are eager (e.g., `filter`, `map`), and the remaining 20 operations are lazy (e.g., `reduce`, etc.).

Although our study is not large-scale, it still reflects the general nature, since the distribution of our data is largely consistent with the result of a large-scale study [9]. In other words, we did not select those samples that are biased in favor of our approach.

As the produced code is in the same manner as shown in Section IV-C, we used the same DTD files to define code structure templates in our evaluation. For each `for` loop and its refactored one, we made ten respective edit operations and checked whether the selective undo for refactoring is correct. Here, for each pair of produced code fragments cf_1' and cf_2', we use *LambdaFicator* [9] to refactor cf_1' again and compare the re-refactored result with cf_2'. Only when they are exactly the same, we decide that the produced code is correct, and the selective undo or redo is successful.

In this paper, we use FOR and ITERATOR to denote the set of `for` loops and the set of their refactored ones and use FOR′ and ITERATOR′ to denote the set of edited `for` loops and the set of edited refactored code.

Even if redo is not significantly different from refactoring again, we also conduct redo in the experiment, since undo and redo are a pair of inverse actions. When performing redo, we made edit operations on FOR and checked whether ITERATOR′ is correct; when performing undo, we made edit operations on ITERATOR and checked whether FOR′ is correct. In both cases, we checked the saved efforts by counting generated lines of code.

B. Effectiveness

Table IV shows the result of redo. Column "#Versions" lists the number of versions. In each version, we made, at least, an edit operation on a `for` loop, and each `for` loop has ten versions. Columns "#Insert", "#Delete" and "#Update" list number of *inserting*, *deleting* and *updating*, respectively. In total, 35.6% of edit operations are *inserting*, 16.9% are *deleting*, and 47.5% are *updating*.

Table V shows the result of undo. Its columns have the same definitions as Table IV. In total, 43.8% of edit operations are *inserting*, 18.3% are *deleting*, and 37.9% are *updating*.

Here, we made more *inserting* and *updating* edit operations than *deleting* modifications, due to the grammar restrictions. Every edit operation guaranteed the syntactical correctness of programs, which means each version after revision satisfies the specified structure.

In Tables IV and V, the columns named "#Error" list number of incorrect results of redo and undo or failures in producing the results of redo and undo, and the columns named "Success Ratio" list their success ratio. As listed in Column "Success Ratio", the average success ratio of redo in our approach is up to 72.5%, while the undo achieved 89% success ratio in our experiment.

Although our success ratios are relatively high, there is still space for improvement. In total, 55 errors in the process of redo and 22 errors in the process of undo are produced. We investigated these errors, and our findings are as follows:

Finding 1. In the process of redo, most of the error cases are related to *inserting* and *deleting* modification on statements. As these edits may change the data flow and variable availability, it can lead to the increment, decrement, or name change of multiple functional operations in ITERATOR. As our implementation does not contain such information, it increases or decreases the functional operations according to the edits, which is different from what *LambdaFicator* does. If we supplemented the information, these errors would be avoided. For example, consider the following `for` loop:

```
for(int type : getItemTypes()) {
  if(type != 0){
    width += getItemIcon(type).getIconWidth();
  }
}
```

We inserted a statement as follows:

```
for(int type : getItemTypes()) {
  if(type != 0){
    width += getItemIcon(type).getIconWidth();
    System.out.println(width);
  }
}
```

To refactor the edited code fragment again with *LambdaFicator*, the `reduce` operations should be modified. As our implementation does not analyse variable availability, it skips this edit. Schürr [26] proposes a triple graph grammar that specifies interdependencies between graph-like data structures. It could be feasible to leverage the graph grammar to define fine-grained *m-to-n* inter-graph relations between abstract syntax trees and a data flow graph, and the relations could be useful to reduce this type of errors.

In addition, if the edits violate the preconditions of the refactoring, *LambdaFicator* cannot produce any results, we also judge that the redo is unsuccessful.

Finding 2. In the process of undo, most of the errors are caused by functional operation chains, and especially, half of the all functional operation chains are in a form of `filter().operation`, and 11% are `map().operation`. When maintainers edit the internal iterators of functional operation chains, they inject new identifiers to represent parameters of lambda expressions. As a result, the functional operations of such a functional operation chain do not have the same parameters. When perform undo based on these functional operation chains, it produces undefined or redefined variables in `for` loops. For example, in the following functional operation chain, the parameter of the `filter` operation is `tp` or `type`:

```
width = getItemTypes().stream()
  .filter((tp) -> tp != 0)
      //Both 'tp' and 'type' are correct.
  .map((type) -> getItemIcon(type).getIconWidth())
  .reduce(width, Integer::sum);
```

The result of undo is as follows:

TABLE IV
RESULTS OF REDO.

Project	#Versions	#Insert	#Delete	#Update	Total	Length	#Error	Success Ratio
ANTLR	40	20	10	16	46	3.5	12	70%
FitNesse	20	7	1	13	21	2.3	3	85%
Hadoop	30	12	8	15	35	2.5	12	60%
Tomcat	20	8	4	15	27	2.6	5	75%
jEdit	20	12	4	7	23	3.75	6	70%
FindBugs	20	5	3	14	22	3.6	3	85%
jUnit	20	6	2	12	20	3.2	4	80%
Soot	30	10	6	15	31	3.47	10	66.7%
Total	200	80	38	107	225	3.14	55	72.5%

TABLE V
RESULTS OF UNDO.

Project	#Versions	#Insert	#Delete	#Update	Total	#SLOC	#Error	Success Ratio
ANTLR	40	23	7	17	47	483	5	87.5%
FitNesse	20	14	1	12	27	251	2	90%
Hadoop	30	13	8	13	34	309	2	93.3%
Tomcat	20	10	6	8	24	233	2	90%
jEdit	20	12	6	7	25	416	2	90%
FindBugs	20	10	3	8	21	349	3	85%
jUnit	20	8	2	10	20	308	3	85%
Soot	30	13	10	14	37	432	3	90%
Total	200	103	43	89	235	2772	22	89%

```
for(int tp : getItemTypes()) {
  if(tp != 0){
    width += getItemIcon(type).getIconWidth();
  }//Error, because 'type' is undeclared.
}
```

In the above code fragment, the `type` variable is undefined. Typically, a refactoring has some preconditions to avoid such errors. If two functional operations o_1 and o_2 are chained as $o_1.o_2$, we propose the constraints as follows:

- o_1 is a `filter` functional operation, and the parameter of the lambda expression in o_2 is the same as that in o_1;
- o_1 is a `map` functional operation, and the lambda expression of o_1 returns a variable that is the same as the parameter of the lambda expression of o_2;
- o_1 is a `map` functional operation with a lambda expression returning an expression, the parameter of the lambda expression of o_2 does not conflict with any parameters of lambda expression in existing functional operations.

In summary, our results show that the selective undo and redo for refactoring can achieve relatively high success ratio in our approach. We further analyse those failure cases, and we propose two findings that could potentially reduce failures during the selective undo and redo.

C. Saved Effort

We measure the saved effort compared with using redo and undo mechanism in text editors by counting the lines of code that is inserted, deleted and removed automatically in our approach, which should have been done manually by the programmers.

Table VI shows the saved effort in undo. Comparing with the lines of code in Table III, we can decide that our implementation saves a lot of manual edit operation. In redo, as a functional operation chain is counted as a line of code, only one line of code should have been edited. However, as the functional operation chain is complicated, it is still tricky and error-prone to edit a line of functional operation chain. With the support of our approach, if all of the refactoring is implemented with our undo and redo mechanism, software maintainers do not solely rely on manual effort to selective undo and redo for refactoring any more.

TABLE VI
SAVED #SLOC IN UNDO.

Project	Insert	Delete	Update	Total
ANTLR	34	11	17	62
FitNesse	23	2	12	37
Hadoop	19	10	13	42
Tomcat	14	7	8	29
jEdit	16	7	7	30
FindBugs	14	5	8	27
jUnit	12	4	10	26
Soot	18	13	14	45
Total	150	59	89	298

Since the refactoring is treated as an atomic edit, the refactoring in our experiment is actually treated as using a statement (iterator) to replace another statement (enhanced `for` loop) when the existing selective redo/undo editor is used. In this case, the edits after refactoring are actually manually preserved to handle the conflict. It has no significant with manual work. Therefore, in our experiment, we measure the saved effort only through comparing with using redo and undo mechanism in text editors instead of selective one.

D. Threats to Validity

In our evaluation, there exist two main threats:

External validity. Although we try to use a relatively complicated refactoring which is easily cause the operation conflict with later edit operation, hoping that working on these complicated cases can quarantine working well on those simpler cases. It still cannot represent all kinds of refactoring. Nevertheless, we would like to try it to some other refactoring and make it more convincing in the future work.

Internal validity. To reduce possible bias, we randomly apply *inserting*, *deleting*, and *updating* during the process of undo and redo, posing an internal validity. As inspired by our evaluation, we plan to eliminate the threat by introducing data flow analysis, variable availability analysis, and preconditions in future work.

VI. Discussion and Future Work

Currently, our approach has been implemented to a particular non-trivial refactoring from enhanced `for` to lambda expression, which could well represent such kind of refactoring, in which there exist certain mapping relation between the program constructs of the code fragments before and after refactoring in the refactoring algorithm. We would like to extend and implement our approach on more such kinds of refactorings in the future work to make them more convincing and benefit more developers.

There exist some potential for applying the putback-based approach to variants of code living on separate but related branches of a repository. Supposing there is a project with a development branch and a release branch. A patch comes in to fix the release branch and then must be applied to the development branch. But the development branch has been edited since the time it forked from the release branch. We can firstly *put* the original development branch to the unfixed release branch to build the bidirectional transformation between the two branches. Then when either branch is modified or fixed, the modification could be reflected to the other branch through the *put* or *get* function to avoid drawback the most recent edits before propagate the bug fixing action.

VII. Related Work

A. Selective Undo

Selective undo has been well studied in the field of graphical, interactive editors. Berlage [2] implemented the selective undo model in GINA though the reverse operation of the selected command to the current context. Myers *et al.* [20], [19] followed the paradigm and improved the mechanism to supporting select new object. Emacs and DistEdit [24] allowed user to select a region of text and undid the most recent edit operations to the region. Yoon *et al.* [31] proposed AZURIE that records edits (i.e., insert, delete, and replace) on code, and allowed programmers to fix conflicts after conducting selective undo. Maruyama [16] proposed an approach to undoing the refactoring which did not conflict with other edit operation or refactoring by keeping a chain of past refactorings for each source file and monitoring the most recent refactoring in each chain. However, all of the above techniques do not support selective undo for refactoring.

B. Refactoring

Program transformations can be conducted between two programs or two representations of a program. Most program transformation techniques are uni-directional. Program refactoring advocates an idea of transforming one program into another which holds the same external behaviors as the original program [8], [18], [21], meanwhile few refactorings support bi-directional transformations. Schürr has proposed triple graph grammars [26] which extend the original pair graph grammar approach [25] to specify the interdependencies between graph-like data structure, and applied it to define fine-grained *m-to-n* inter-graph relationships between program representations (e.g., abstract syntax tree and control flow graph) [14]. However, there still remains a difficulty about how to transform two programs bidirectionally.

C. Bidirectioanl Transformation

In this paper, we have presented a putback-based updating approach to supporting selective undo for refactoring. The notion of putback-based updating originates from the database community, for describing how the relation between two data sets is maintained, and when one is updated, how the other is updated [12], [13]. The putback-based updating strategy has been applied in the domain of software engineering, as a technique for model-to-model transformations. Lämmel [15] has identified the category of coupled transformations: an updating of one artefact requires an updating of another dependent artefact to be performed such that the consistency between them is guaranteed. Xiong *et al.* [29] have proposed Beanbag, an OLC-based language for defining and checking model consistencies, which allows updates to be defined and propagated between models. Yu *et al.* [32] have proposed a tool-support approach, *blinkit*, to maintaining invariant traceability between user-modified and template-generated code in model-driven development. It utilizes a *BX* engine called GRoundTram [10], which helps to guarantee the round-trip property in bidirectional model transformations. Song *et al.* [27] have utilized *BX* to link software architecture to programs and eliminate the conflicts between them. Foster *et al.* [6] have introduced a domain specific programming language with which a number of *lenses* can be defined. Since then, a variety of lenses [3], [4], [7], [11] have then been developed. Compared with the above work, we utilize the synchronization feature to solve the problem of selective undo for refactoring.

VIII. Conclusion

In this paper, we have formalized selective undo for refactoring as a putback-based updating problem and proposed an approach to solving the problem. We define a *BX* function between code fragment before and after refactoring, which can help propagate the edit operation when either is edited. We have also conducted a controlled experiment to evaluate the approach. The results show that our approach provides with support in selective undo for refactoring effectively.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank Hiroyuki Kato and Soichiro Hidaka for their invaluable suggestions on bidirectional transformations. Yuting Chen is the corresponding author. This research is sponsored by 973 Program in China (Grant No. 2015CB352203), and the National Nature Science Foundation of China (NSFC) (Grant Nos. 61572312, 61572313, and 61272102). This work is also partially supported by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 in Japan. Hao Zhong is partially supported by Science and Technology Commission of Shanghai Municipality's Innovation Action Plan (No.15DZ1100305).

REFERENCES

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transaction on Database System*, 6(4):557–575, 1981.

[2] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1(3):269–294, 1994.

[3] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419, 2008.

[4] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS*, pages 338–347, 2006.

[5] M. L. Collard, J. I. Maletic, and B. P. Robinson. A lightweight transformational approach to support large scale adaptive changes. In *Proc. ICSM*, pages 1–10, 2010.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proc. POPL*, pages 233–246, 2005.

[7] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *ICFP*, pages 383–396, 2008.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[9] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *ESEC/FSE*, pages 543–553, 2013.

[10] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *Proc. ASE*, pages 480–483, 2011.

[11] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *POPL*, pages 495–508, 2012.

[12] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM*, pages 178–189, 2004.

[13] S. Kawanaka and H. Hosoya. bixid: a bidirectional transformation language for xml. In *ICFP*, pages 201–214, 2006.

[14] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[15] R. Lämmel. Coupled software transformations. In *IWSET*, pages 31–35, 2004.

[16] K. Maruyama. An accurate and convenient undo mechanism for refactorings. In *13th Asia-Pacific Software Engineering Conference (APSEC 2006)*, pages 309–316, 2006.

[17] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[18] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[19] B. A. Myers. Scripting graphical applications by demonstration. In *Proceeding of the CHI '98 Conference on Human Factors in Computing Systems*, pages 534–541, 1998.

[20] B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *Conference on Human Factors in Computing Systems, CHI '96*, pages 260–267, 1996.

[21] W. F. Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[22] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *Proc. PEPM*, pages 39–50, 2014.

[23] H. Pacheco, T. Zan, and Z. Hu. Biflux: A bidirectional functional update language for xml. In *Proc. PPDP*, pages 147–158, 2014.

[24] A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, 1994.

[25] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971.

[26] A. Schürr. Specification of graph translators with triple graph grammars. In *WG*, pages 151–163, 1994.

[27] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[28] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *Proc. ESEC/FSE*, pages 315–324, 2009.

[29] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proc. 22nd ASE*, pages 164–173, 2007.

[30] Y. Yoon and B. A. Myers. An exploratory study of backtracking strategies used by developers. In *5th CHASE*, pages 138–144, 2012.

[31] Y. Yoon and B. A. Myers. Supporting selective undo in a code editor. In *Proc. ICSE*, pages 223–233, 2015.

[32] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In *Proc. ICSE*, pages 540–550, 2012.

[33] T. Zan, H. Pacheco, and Z. Hu. Writing bidirectional model transformations as intentional updates. In *ICSE Companion*, pages 488–491, 2014.