Refactoring with Synthesis

Veselin Raychev
ETH Zürich
veselin.raychev@inf.ethz.ch

Max Schäfer
Nanyang Technological University
schaefer@ntu.edu.sg

Manu Sridharan
IBM T.J. Watson Research Center
msridhar@us.ibm.com

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Abstract

Refactoring has become an integral part of modern software development, with wide support in popular integrated development environments (IDEs). Modern IDEs provide a fixed set of supported refactorings, listed in a refactoring menu. But with IDEs supporting more and more refactorings, it is becoming increasingly difficult for programmers to discover and memorize all their names and meanings. Also, since the set of refactorings is hard-coded, if a programmer wants to achieve a slightly different code transformation, she has to either apply a (possibly non-obvious) sequence of several built-in refactorings, or just perform the transformation by hand.

We propose a novel approach to refactoring, based on synthesis from examples, which addresses these limitations. With our system, the programmer need not worry how to invoke individual refactorings or the order in which to apply them. Instead, a transformation is achieved via three simple steps: the programmer first indicates the start of a code refactoring phase; then she performs some of the desired code changes manually; and finally, she asks the tool to complete the refactoring.

Our system completes the refactoring by first extracting the difference between the starting program and the modified version, and then synthesizing a *sequence* of refactorings that achieves (at least) the desired changes. To enable scalable synthesis, we introduce *local refactorings*, which allow for first discovering a refactoring sequence on small program fragments and then extrapolating it to a full refactoring sequence.

We implemented our approach as an Eclipse plug-in, with an architecture that is easily extendable with new refactorings. The experimental results are encouraging: with only minimal user input, the synthesizer was able to quickly discover complex refactoring sequences for several challenging realistic examples.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Integrated environments; Eclipse

General Terms Algorithms

Keywords Refactoring; Synthesis

1. Introduction

Software refactoring improves the structure of existing code through behavior-preserving transformations, themselves called refactorings. Since it was first formally described twenty years ago [8, 22], refactoring has become an important part of modern software development, and is a staple of agile software development techniques such as Extreme Programming [2].

While originally conceived as a manual activity, it did not take long for the first refactoring tools to appear [26] that offered support for automating simple refactoring transformations. Since then refactoring support has become *de rigueur* in interactive development environments (IDEs) for object-oriented languages. In particular, all major Java IDEs such as Eclipse [4], IntelliJ IDEA [12] and NetBeans [21] come with built-in support for many refactorings.

Recent studies, however, have shown that these refactoring tools are severely underused [19, 36]: while programmers do refactor frequently, they perform up to 90% of refactorings by hand, even where tool support is available. One major issue identified by these studies is poor discoverability: in order to initiate an automated refactoring, the pro-

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509544

grammer has to select it by name from a menu[1] or use a corresponding keyboard shortcut, requiring the programmer to not only know which refactoring transformations the IDE supports but also to memorize their names. A second obstacle is the complexity of the user interface for some refactorings, which are controlled by complex configuration dialogs that disrupt the programmer's workflow.

To alleviate these problems, several researchers have proposed novel user interface paradigms for refactoring tools: Lee et al. [16] present a system where refactorings are initiated by drag-and-drop gestures, whereas BeneFactor [7] and WitchDoctor [5] are tools for refactoring "autocompletion" that observe a programmer's editing operations and try to discover editing patterns suggestive of refactorings. When such a pattern is discovered, the programmer is offered the choice of completing the refactoring task using the built-in refactoring tool.

These tools make it easier to apply individual refactorings already supported by the IDE. However, many very natural refactoring transformations do not straightforwardly map to these built-in refactorings. For example, the EXTRACT METHOD refactoring implementations of present-day IDEs provide no control over the set of parameters to be provided by the extracted method. As we shall discuss further in Section 2, this means that programmers sometimes have to perform a non-trivial sequence of auxiliary refactorings in order for the method extraction to yield the required result. This is tedious, particularly if some refactoring late in the sequence fails unexpectedly (e.g., due to a violated pre-condition) and the programmer has to undo the previous steps one by one. Even discovering the right sequence of refactorings to perform is often non-trivial, and the programmer may ultimately fall back to performing the refactoring by hand.

We propose a novel interface for invoking automated refactorings based on synthesis from examples (for a recent survey, see [9]). In our approach, a developer performs an automated refactoring using the following process:

1. Click a "Start Refactoring" button, thereby indicating the program P_i whose semantics should be preserved by the refactoring.
2. Manually perform a few of the edits for the desired refactoring, resulting in a program P_m.
3. Click a "Complete" button, at which point the tool takes as input the two examples P_i and P_m and attempts to discover a sequence of automated refactorings that, when applied to P_i, yields a program that includes the edits introduced in P_m over P_i.

This refactoring interface, which we have implemented in an Eclipse plugin called RESYNTH, provides a number of advantages over traditional approaches where the user needs to figure out which (sequence of) refactorings to invoke:

- The user is freed from remembering a different menu item or keyboard shortcut for each supported refactoring; instead, all refactorings are exposed through a simple, unified interface.[2]
- Applying a transformation requiring a sequence of refactorings becomes much easier, as the tool discovers the (possibly non-obvious) sequence for the user. Also, the tool ensures that all refactorings in the sequence will succeed before applying any of them, removing the burden of having to undo earlier refactorings if a later one fails.
- Similarly, performing a set of related refactorings where the order is unimportant (e.g., renaming both a field and its accessor methods) is simplified, since the refactorings are applied as a unit.
- For a refactoring developer, adding a new refactoring no longer requires cluttering the user interface with another menu item.

The heart of RESYNTH is a search strategy for discovering appropriate refactoring sequences based on a small number of user edits. A naïve brute-force search is ineffective for even the smallest programs. Instead, RESYNTH takes the following approach:

1. Program entities that were not edited by the user are discarded to narrow the search space.
2. Over this pruned program, an A^* search [11] is performed to discover refactoring sequences, guided by a heuristic function that minimizes edit distance and expression distance from the user edits.
3. After a solution is discovered that works for the pruned program, RESYNTH attempts to execute the discovered sequences of Eclipse refactorings on the full program.

While BeneFactor and WitchDoctor also infer refactorings from user edits, they differ from our system in that (1) they do not require the user to indicate a refactoring is occurring and (2) they cannot perform transformations requiring a sequence of refactorings. While (1) can be an advantage for novice users, requiring the user to indicate the start of the refactoring enables the tool to discover more complex sequences, and allows for performing several independent refactorings "atomically."

RESYNTH is architected in a manner that eases the process of adding new refactorings. Beyond standard refactoring functionality, RESYNTH only requires that each refactoring provide a `successors` function to enumerate the possible ways to apply the refactoring to a given pruned program, thereby defining the search space. Adding a `successors`

[1] In Eclipse 4.2, this menu has 23 entries.

[2] Of course, the alternative interface could also be used in conjunction with standard menu items.

function is straightforward in most cases, and otherwise refactoring implementors need not be concerned about details of the refactoring search.

To assess the effectiveness of our search strategy, we evaluated RESYNTH on a set of synthetic benchmarks and on real example refactorings collected from Stack Overflow[3] and other sources. We show that our search techniques are significantly more effective than alternate approaches, and can handle many challenging real-world examples.

Main Contributions The contributions of this paper are:

- A new interface for refactoring tools, where the user indicates the desired transformation with example edits and the tool synthesizes a *sequence* of refactorings that include the edits.
- A novel technique for synthesizing refactoring sequences via heuristic search over pruned programs.
- An implementation RESYNTH, whose architecture minimizes the effort required to add new refactorings.
- An initial evaluation of RESYNTH, showing it can synthesize complex refactoring sequences for real examples.

Our paper is organized as follows. Section 2 gives a detailed example to motivate our techniques. Then, Section 3 presents our core search techniques in detail. Section 4 details the design and implementation of our tool RESYNTH, and Section 5 presents our experimental evaluation. Finally, Section 6 discusses related work, and Section 7 concludes.

2. Overview

Modern Java IDEs such as Eclipse, NetBeans or IntelliJ IDEA offer a large number of built-in refactorings that can be activated through a menu or a keyboard shortcut. The precise set of supported refactorings differs between IDEs, but usually includes a set of core refactorings such as EXTRACT METHOD or ENCAPSULATE FIELD, many of them originally implemented in the Smalltalk Refactoring Browser [26].

This set of refactorings, however, is fixed, and does not directly accommodate some fairly simple transformations. As an example, consider the method extraction problem shown in Fig. 1, which is taken from Fowler's well-known book on refactoring [6]. The original program, a fragment of a class representing an account, is given on the left. As shown in the refactored program on the right, we want to extract lines 6 and 7 into a new method `printDetails()`. Crucially, however, the expression `getOutstanding()` on line 8 should *not* be extracted into the new method, but passed as a parameter instead.

This refactoring cannot be performed in one step using the refactoring tools of Eclipse, NetBeans or IntelliJ, since their implementations of EXTRACT METHOD do not permit the programmer to control the set of parameters of the extracted method, which is instead determined automatically by examining the program's data flow.

The transformation can be achieved by composing two refactorings. First, the two statements *including* the expression to be passed as a parameter are extracted into a `printDetails()` method:

```
40  private void printDetails() {
41    System.out.println("name: " + name);
42    System.out.println("outstanding: " +
43                       getOutstanding());
44  }
```

Then, the INTRODUCE PARAMETER refactoring is applied to the `getOutstanding()` call, yielding the desired result. However, INTRODUCE PARAMETER is not widely known [36], and, at least in Eclipse, its implementation is buggier than more popular refactorings.

Another alternative is to decompose the refactoring into three steps. First, we extract the expression to be passed as a parameter into a local variable `outstanding` inside `printOwing()`, as follows:

```
45  void printOwing() {
46    printBanner();
47    double outstanding = getOutstanding();
48    System.out.println("name: " + name);
49    System.out.println("outstanding: " +
50                       outstanding);
51  }
```

Then, we extract lines 48–50 into the `printDetails()` method. Since `outstanding` is used by, but not defined in, the code to be extracted, it will be turned into a parameter to `printDetails()`, as desired. Finally, we can apply INLINE LOCAL to `outstanding`, achieving the desired program from Figure 1(b).[4]

Vakilian et al. [35] found that this latter sequence of refactorings is quite frequently performed in practice, suggesting that programmers often need to perform complex refactorings that exceed the capabilities of current refactoring engines, forcing them to manually compose several refactorings in order to achieve a single transformation.

While EXTRACT LOCAL is more widely known than INTRODUCE PARAMETER [36], its use in this situation is highly non-obvious and indeed somewhat counter-intuitive, since its effect is later partially undone by an inlining. This requires the programmer to plan ahead to compensate for limitations of the refactoring tool.

Abadi et al. [1] conducted a case study using Eclipse's built-in refactorings to conduct a complex refactoring, and found that only three out of 13 uses of EXTRACT METHOD

[3] `http://stackoverflow.com`

[4] In fact, this approach does not quite work in either Eclipse, NetBeans or IntelliJ, since their implementations of EXTRACT LOCAL always put the declaration of the extracted local variable on the line immediately preceding the extraction site, so `outstanding` ends up after line 48 and has to be moved manually before the refactoring can continue.

```
1 public class Account {
2   private String name;
3
4   void printOwing() {
5     printBanner();
6     System.out.println("name: " + name);
7     System.out.println("outstanding: " +
8                        getOutstanding());
9   }
10
11  private double getOutstanding() {
12    //...
13  }
14
15  private void printBanner() {
16    //...
17  }
18 }
```

(a)

```
19 public class Account {
20   private String name;
21
22   void printOwing() {
23     printBanner();
24     printDetails(getOutstanding());
25   }
26
27   private void printDetails(double outstanding) {
28     System.out.println("name: " + name);
29     System.out.println("amount: " + outstanding);
30   }
31
32   private double getOutstanding() {
33     //...
34   }
35
36   private void printBanner() {
37     //...
38   }
39 }
```

(b)

Figure 1. Example of a complicated refactoring that cannot be achieved in a single step in current IDEs; changes highlighted.

could be automatically performed by Eclipse. They suggest improving Eclipse's implementation of EXTRACT METHOD to provide more fine-grained control over the code to extract and the parameters of the extracted method, but recent research [35] suggests that such an improved (and invariably much more complex) refactoring tool would not necessarily be very popular, since programmers seem to favor composition (of several simple refactorings) over configuration (of one complex refactoring).

Our approach We propose a radically different approach to solving this problem: instead of having to memorize what individual refactorings do and figure out how to compose them to achieve complex transformations, the programmer can perform some of the desired changes by hand, and then use a tool to automatically search for a suitable sequence of refactorings that performs (at least) those changes.

With our approach, a programmer could achieve the refactoring of Fig. 1 quite easily. After indicating to the tool that a refactoring is beginning, the user would just manually replace lines 6 and 7 with the desired method call `printDetails(getOutstanding())`. At this point, of course, the program does not compile any more, since the method `printDetails` has not been defined yet. But, based on this simple edit, our tool RESYNTH can determine the sequence of refactorings needed to refactor the original program into the form in Fig. 1(b). Note that existing tools like BeneFactor [7] and WitchDoctor [5] cannot handle this example based on the edit above: they require the user to know a sequence of refactorings to apply.

The same interface can, of course, also be used for transformations that can be achieved by a single refactoring (discussed further in Section 5). In such cases, RESYNTH has the advantage of providing a unified interface that frees the programmer from having to memorize refactoring names and their associated keyboard shortcuts or menu items.

There are many real-world examples besides that of Figure 1 in which a desired transformation can be accomplished by applying a non-obvious sequence of basic refactorings. For instance, previous work [25, 32] has discussed swapping two names, e.g., transforming field declarations `String s1; String s2;` to `String s2; String s1;`, changing uses of `s1` and `s2` appropriately. One cannot apply the RENAME refactoring to first change `s1` to `s2` or vice-versa, since a name conflict arises. With our approach, one simply swaps the names in the field declarations manually, and the tool discovers a sequence of three RENAME refactorings to complete the transformation: `s1` to `tmp`, `s2` to `s1`, and finally `tmp` to `s2`. As with the previous example, this refactoring sequence is non-obvious, but with RESYNTH, the user is freed from worrying about the details. In Section 5, we show that RESYNTH was able to handle several challenging real-world examples, including these.

3. Approach

In this section, we discuss our approach to synthesizing refactoring sequences in detail. We formalize the problem, introduce the notion of local refactoring, describe the synthesis algorithm, and prove some of its key properties.

3.1 Setting

A refactoring $r \in R = Prog \times \overline{PS} \rightharpoonup Prog$ is a partial function which takes as input a program and a sequence of parameters and returns a transformed program. The function is partial because the input program may not satisfy preconditions required for the transformation to be behavior-preserving. The set of parameters varies among refactorings. For example, a RENAME refactoring takes two parameters: a program entity e to be renamed and the new name for e. The types of parameters are not important here; they will be discussed further in Section 4. An invocation of a refactoring function is denoted as $r(P, \overline{ps})$ where each argument in $\overline{ps}$ belongs to $\overline{PS}$.

Sequence of Refactorings We often use the term "sequence of refactorings" to stand for a sequence of invocations of refactoring functions. Formally, a finite sequence of refactorings $r^n(P)$ of length $n > 0$ invoked with particular arguments is defined as:

$$r^n(P,\ \overline{ps_1}, ...,\ \overline{ps_n}) = r_n(...(r_2(r_1(P,\ \overline{ps_1}),\ \overline{ps_2})..., \overline{ps_n})$$

The problem addressed by our synthesis procedure can now be stated informally as follows:

> Given an initial program P_i and a modified program P_m, the goal of the synthesizer is to discover a sequence of refactorings $P_f = r^n(P_i,\ \overline{ps_1}, ...,\ \overline{ps_n})$ (for some $n > 0$) that preserves all changes introduced in P_m over P_i.

To solve this problem, a synthesizer must find *both*: i) which refactorings to use and ii) which arguments to pass to the refactorings of step i).

Key Challenge A naïve approach to the above problem would be to apply refactorings in sequence directly to P_i, searching for a sequence containing the changes in P_m. Unfortunately, this approach performs poorly, due to the need to repeatedly transform and check preconditions on the *entire* input program. We observed that applying a single Eclipse refactoring to even a small program took roughly half a second (consistent with previous work [5]), making a search among thousands of refactoring sequences infeasible in practice.

While in principle the Eclipse refactorings could be further optimized, precondition checking for many refactorings such as renaming is an inherently global problem that needs to take the whole program (including external libraries it depends on) into account, thus limiting the potential speedup. Furthermore, given the current architecture of Eclipse's refactoring engine, applying two refactorings in a row invariably entails changing the program at a textual level and reparsing affected compilation units, which would further slow down search.

The key challenge, then, is how to speed the search process sufficiently to discover realistic refactoring sequences in reasonable amounts of time.

Solution Outline Next, we discuss our solution. We first discuss how to capture changes between two programs. Then, we discuss *local refactorings*, an important component of our approach. A local refactoring restricts a refactoring to a program fragment, enabling the search to consider only the changed parts of a program instead of an entire program, which is key to scalability. We then present our search algorithm, based on local refactorings, and finally we discuss the overall guarantees provided by the synthesizer.

3.2 Capturing Program Change

We define $T(P)$ to be the abstract syntax tree (AST) of program P. For programs with multiple files, we create one tree with a root node that joins the ASTs of all files. Given trees T_1 and T_2, we define the following tree operations:

- $T_1 \setminus T_2$ is a tree where each full path in T_1 which does not occur in T_2 is kept.
- $T_1 \subseteq T_2$ is true iff every rooted path in T_1 is a rooted path in T_2.

A rooted path is a path from the root of tree T to any node in T, and a full path is a rooted path which ends in a leaf.

A *program change* is a pair (c_i, c_m) where c_i is a subtree of $T(P_i)$ and c_m is a subtree of $T(P_m)$. Intuitively, (c_i, c_m) means that the tree c_i from $T(P_i)$ was changed to the tree c_m of $T(P_m)$. In terms of the previously-defined tree operations, $(c_i, c_m) = (T(P_i) \setminus T(P_m), T(P_m) \setminus T(P_i))$.

A program P_f is said to preserve a change (c_i, c_m) if $c_m \subseteq T(P_f)$. Note that c_i and c_m need not be syntactically-valid program ASTs since they may be missing subtrees that did not change between P_i and P_m.

Example Consider the two expressions $P_i = x * y + 7$ and $P_m = f() + 7$, whose ASTs are shown in Fig. 3. As defined above, c_i is the tree $x*y+$ and c_m is the tree $f()+$, illustrated as dashed circles in the figure. In this example, neither c_i nor c_m are well-formed ASTs, since they miss the right-hand operand of $+$, which is identical in both programs.

In what follows, we write P for $T(P)$ to avoid notational clutter. So for instance, a change (c_i, c_m) is written as $(P_i \setminus P_m, P_m \setminus P_i)$ and preservation as $c_m \subseteq P_f$.

Now that we have defined the notion of a program change, we can re-state our goal formally:

> Given an initial program P_i and a modified program P_m, the goal of the synthesizer is to discover a sequence of refactoring invocations (for some $n > 0$) $P_f = r^n(P_i,\ \overline{ps_1}, ...,\ \overline{ps_n})$ such that $c_m \subseteq P_f$.

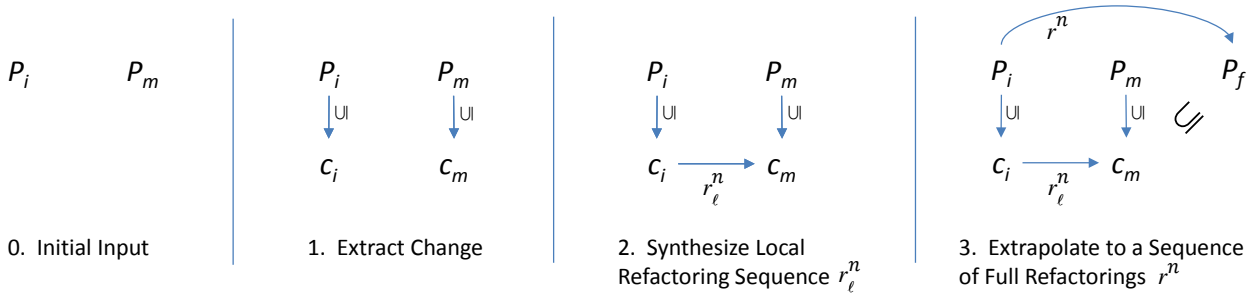

Figure 2. Synthesis Steps

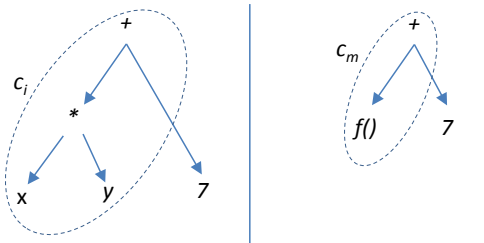

Figure 3. Two ASTs and the change (c_i, c_m) between them. The change is captured with dotted lines.

3.3 Local Refactoring

The concept of a *local refactoring* is key to the practicality of our synthesis approach. A local refactoring enjoys the following benefits: i) it can operate on a small portion of the program instead of the entire program; ii) it can perform fewer pre-condition checks that those performed by a full refactoring; iii) it works directly on trees and hence does not need to parse and generate code.

For a given refactoring r, we denote a corresponding local refactoring as r_l. A local refactoring r_l is defined as follows:

$$r_l : Tree \times \overline{PS} \rightharpoonup Tree$$

That is, r_l is a partial function which takes as input a tree, a sequence of parameters and returns a tree. Similarly to the full refactoring r, the function is partial because the input may not always satisfy certain pre-conditions required for the function to fire. The difference from the definition of a full refactoring is the use of trees instead of programs. Sequences of local refactorings are defined similarly to sequences of full refactorings defined earlier.

Extraction Function Given an invocation of a local refactoring, we often need to obtain a corresponding invocation of a full refactoring. Therefore, for each local refactoring function r_l, we associate a corresponding extraction function:

$$\mu_{r_l} : \overline{PS} \rightarrow \overline{PS}$$

Then, given a local refactoring invocation $r_l(t, \overline{ps})$, the function $\mu_{r_l}(\overline{ps})$ computes a new sequence of arguments $\overline{ps'}$ to be passed to the full refactoring r. Typically μ_{r_l} is the identity function, but we allow each implementation of a local refactoring to decide what function it needs.

For a local refactoring to be useful in synthesis, it needs to satisfy the following condition w.r.t a full refactoring.

Definition 3.1 (Correct Local Refactoring)**.** *A local refactoring r_l is correct w.r.t its corresponding full refactoring r iff* $\forall P \in Prog, f \in Tree, \overline{ps} \in \overline{PS}$*:*

$$f \subseteq P \wedge f' = r_l(f, \overline{ps}) \wedge P' = r(P, \mu_{r_l}(\overline{ps})) \Rightarrow f' \subseteq P'$$

Fig. 4 gives the intuition for the definition: a local refactoring is correct w.r.t. a full refactoring unless both are enabled and the result of the local refactoring is *not included* in the result of the full one. Note that depending on the definition of the extraction function μ there could be many different local refactorings r_l that are correct with respect to its corresponding full refactoring r.

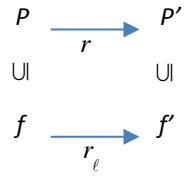

Figure 4. Local refactoring correctness

3.4 Modular Synthesis

We next describe the three steps of the synthesis process. These steps are also illustrated in Fig. 2.

Step 1: Extract Change The result of this step is a program change $(c_i, c_m) = (P_i \setminus P_m, P_m \setminus P_i)$ (as defined in Section 3.2).

Since we need only the change, we do not need to explicitly construct the entire ASTs P_i and P_m. Indeed, if we compare the trees of P_i and P_m and eliminate the common parts between them, this would eliminates classes, methods, type definitions, statements and expressions that remain unchanged between the two programs. This means that for large programs, we need not compare unmodified compilation units. Then, we construct the ASTs P_i' and P_m' which include only the modified compilation units and compute the program change by using the formula $(P_i' \setminus P_m', P_m' \setminus P_i')$.

Step 2: Synthesize Local Refactoring Sequence We next describe how we compute a local refactoring sequence. The goal of this step can be stated as follows:

> Given a program change (c_i, c_m), the goal of the synthesizer is to discover a sequence of local refactoring invocations $t = r_l^n(c_i, \overline{ps_1}, ..., \overline{ps_n})$ (where $n > 0$) such that $c_m = t$.

We assume that at each step in the search, there is a *finite* set of (refactoring, input) pairs that apply to the current tree. Some work may be required to compute this finite set, as refactorings can have unbounded inputs (like the new name input for RENAME); we describe how our implementation handles this issue in Section 4.

A naïve solution to our search problem would be to use breadth-first search, which will find the shortest possible refactoring sequence (assuming the aformentioned finitization). However, our experiments indicate that such a search rarely scales beyond sequences of more than four refactorings, as the search space grows exponentially in the length of the desired sequence.

Instead, we employ an A^*-based search [11]. A^* iteratively computes a distance function d from the initial tree c_i to every other generated tree. Our distance function is simply the length of the current (partial) refactoring sequence. To speed up the search, A^* also uses a heuristic function h that estimates the distance from each tree to the target tree c_m. At every step, the tree t with minimal $d(c_i, t) + h(t)$ is processed. The "successors" of t, i.e., the results of applying all possible local refactorings to t (a finite set, as discussed above), are added to the set of candidates, and the search continues.

Heuristic functions First, we define a correct heuristic function.

Definition 3.2 (Correct Heuristic Function)**.** *A heuristic function is correct if it satisfies the following properties:*

- $\forall t \in Tree$, $h(t) \geq 0$, *and*
- *If* $t = c_m$ *then* $h(t) = 0$

We propose the following correct heuristic functions:

1. $h_1(t)$ is the edit distance between t and c_m: this is the minimum number of node renames, leaf node inserts or leaf node deletes required to get to the tree c_m from the tree t.
2. $h_2(t)$ is the expression distance between t and c_m: this is the number of program expressions present in one of the trees which are *not* present in the other one. This function ignores parts of programs which are not valid expressions. For example the statement `x=y+1` consists of the expressions `x=y+1`, `x`, `y+1`, `y` and `1`. If t consists of the statement `x=y+1` and c_m of `x=z+1`, then $h_2(t) = 6$, because t contains `x=y+1`, `y+1` and `y` which are not present in c_m, and c_m contains `x=z+1`, `z+1` and `z` which are not present in t.

A *consistent* heuristic function is a correct function which also satisfies the property $\forall t_1, t_2$, $h(t_1) \leq d(t_1, t_2) + h(t_2)$. If a consistent heuristic function is provided, the A^* algorithm will always find the shortest possible sequences [11]. However, building such a heuristic function is dependent on the available local refactorings and hence would require change each time a new local refactoring is added.

The heuristic function that we choose is not consistent, but is correct, which is sufficient for our optimality guarantees (discussed later) and does not affect the correctness of the produced local sequence. The heuristic function is used only to improve the speed of the search and decrease the number of explored trees. In our implementation, we constructed several heuristic functions by building different linear combinations $h(t) = a_1 h_1(t) + a_2 h_2(t)$ and tuning the corresponding constants a_1 and a_2. Section 5 presents results with different choices for a_1 and a_2.

Step 3: Extrapolate to a Sequence of Full Refactorings Once a local refactoring sequence $r_l^n(c_i, \overline{ps_1}, ..., \overline{ps_n})$ is computed, the final step is to extrapolate from that sequence and obtain a sequence of full refactorings. The sequence of full refactorings is obtained by applying the extraction function to each local refactoring in the local refactoring sequence. That is, for the full refactoring sequence we obtain $r^n(P_i, \mu_{r_{l_1}}(\overline{ps_1}), ..., \mu_{r_{l_n}}(\overline{ps_n}))$. However, r^n may not actually be feasible, since local refactorings may ignore some of the pre-conditions which are checked in the sequence of full refactorings. If the sequence of full refactorings is infeasible, our algorithm searches for a different local refactoring sequence and repeats the process. Otherwise, we obtain the desired program P_f (from the sequence of full refactorings).

3.5 Guarantees

We next discuss the two main guarantees provided by our approach. First we show that the synthesizer produces a refactoring sequence which satisfies our objective (that is, correctness). Further, we also prove a property of optimality.

The fact that our synthesizer achieves the correctness objective is also illustrated in Step 3 of Fig. 2, that is, with $c_m \subseteq P_f$. We prove this below.

Lemma 3.3 (Correctness of Synthesis)**.** *For any sequence of full refactorings* $P_f = r^n(P_i, \overline{ps_1}, ..., \overline{ps_n})$ *produced by the synthesizer,* $c_m \subseteq P_f$.

Proof. Let $r_l^n(c_i, \ldots)$ be the local refactoring sequence produced in Step 2 of the algorithm, from which the given full refactoring sequence $r^n(P_i, \ldots)$ was obtained. Here we do not list the arguments to avoid clutter and use ... instead.

The proof proceeds by induction. For the induction hypothesis assume that for some $k < n$, $r_l^k(c_i, \ldots) \subseteq r^k(P_i, \ldots)$ where the sequence $r^k(P_i, \ldots)$ is obtained from the sequence $r_l^k(c_i, \ldots)$. We need to prove that the tree

$r_l^{k+1}(c_i, \ldots) \subseteq r^{k+1}(P_i, \ldots)$. From the requirement that each local refactoring is correct (Definition 3.1) it follows that $r_{l_{k+1}}(r_l^k(c_i, \ldots), \ldots) \subseteq r_{k+1}(r^k(P_i, \ldots), \ldots)$. Then, using induction we have proven that $r_l^n(c_i, \ldots) \subseteq r^n(P_i, \ldots)$. Step 2 of the local refactoring synthesis terminates only if $r_l^n(c_i, \ldots) = c_m$. Hence, by substitution it follows that $c_m \subseteq r^n(P_i, \ldots)$ proving our objective. □

Next, we define a notion of optimality:

Definition 3.4 (Optimality). *$P_f = r^n(P, \overline{ps_1}, ..., \overline{ps_n})$ if:*

- *$n = 1$ and $P \neq P_f$, or*
- *$n > 1$ and for all $i, 0 \leq i < n-1, r_o \in R, \overline{ps_o} \in \overline{PS}$, it is the case that $r_o(r^i(P, \overline{ps_1}, ..., \overline{ps_i}), \overline{ps_o}) \neq P_f$.*

Intuitively, the above definition of optimality says that we cannot replace a suffix of the refactoring sequence with another refactoring and obtain the same result. For example, a sequence of two rename refactorings where the first one renames method A to B and the second one renames B to C is not optimal, because that sequence can be replaced by a single refactoring that renames A directly to C. A consequence of the above definition is that if a sequence is optimal and correct, it means that no prefix of the sequence is correct. For example, if the sequence of refactorings $a \cdot b \cdot c$ is optimal and correct, it means that neither a nor $a \cdot b$ can solve the problem, that is, no shorter correct prefix exists.

Lemma 3.5 (Optimality of Synthesis). *A refactoring sequence $r^n(P, \overline{ps_1}, ..., \overline{ps_n})$ produced by the synthesizer is optimal if for all $i, 1 \leq i < n, \mu_{r_{l_i}}$ is a bijection.*

Proof. First, we will show that the local refactoring sequence produced by A^* is optimal. If the sequence $c_m = r_l^n(c_i, \overline{ps_1}, ..., \overline{ps_n})$ was not optimal, then $\exists j, 1 \leq j < n-1$ and a local refactoring r_{l_o} with parameters $\overline{ps_o} \in \overline{PS}$, such that $r_{l_o}(r_l^j(c_i, \overline{ps_1}, ..., \overline{ps_j}), \overline{ps_o}) = c_m$. Let $c_P = r_l^j(c_i, \overline{ps_1}, ..., \overline{ps_j})$. Because $c_m = r_{l_o}(c_P, \overline{ps_o})$, c_m is a successor of c_P. c_P was processed in A^* before c_m and its computed distance function is $d(c_i, c_P) = j$. Because all successors of c_P are added when c_P is processed and c_m is a successor of c_P, then $n = d(c_i, c_m) \leq j + 1$, which means $j \geq n - 1$, which contradicts the non-optimality of $r_l^n(c_i, ...)$.

Next, assume that $r^n(P, \overline{ps_1}, ..., \overline{ps_n})$ is produced by using $\mu_{r_{l_i}}$ $(i \in [1, n])$ from $r_l^n(c_i, \overline{ps_1}, ..., \overline{ps_n})$ and is not optimal. This means that $\exists j, 1 \leq j < n-1$ and a refactoring $r_o \in R$ with parameters $\overline{ps_o} \in \overline{PS}$, such that $r_o(r^j(P, \overline{ps_1}, ..., \overline{ps_j}), \overline{ps_o}) = P_f$. Because for all $i, 1 \leq i < n, \mu_{r_{l_i}}$ is a bijection, we must be able to obtain $r_o(r^j(P, \overline{ps_1}, ..., \overline{ps_j}), \overline{ps_o})$ from a local sequence $r_{l_o}(r_l^j(c_i, \overline{ps_1}, ..., \overline{ps_j}), \overline{ps_o}) = c_m$, but this contradicts to the optimality of $r_l^n(c_i, \overline{ps_1}, ..., \overline{ps_n})$.

□

Our optimality guarantee shows that the sequence of refactorings which we produce cannot be trivially shortened. This is important as it ensures that we do not produce redundant or unnecessary steps and that our diverse solutions are not trivially reducible to the same solution.

Example Let us illustrate the steps described in Fig. 2 on the simple example in Fig. 5. For readability, the example contains only a single refactoring. Here, the user provides an initial program P_i that computes the area of a triangle using Heron's formula. The variable `T` stores the perimeters divided by two and then the square of the area is computed in `s`. Assume that the user wants to rename the variable `T` to `p` by changing some places where `T` is mentioned to `p`. The changes are in the program P_m (P_m does not compile).

Given P_i and P_m, in step 1 the algorithm extracts the change and produces the two trees c_i and c_m. In step 2, a local refactoring sequence is synthesized. In this example, the sequence consists of a single local rename. That is, only the occurrences `T` in the tree c_i have their names changed to `p` resulting in the tree c_m. In step 3, the discovered local refactoring sequence is applied on the full program. As a result, P_f is a program where all occurrences of `T` are renamed to `p`. Note that only in this step, we can check if the rename is actually feasible. For instance, had the name `p` already been used as a name of another variable, then this step would fail.

When the rename refactoring is implemented as a local refactoring, it does not perform the check that the new name (e.g. `p` in our example) does not conflict with names outside of the local tree, but this check is performed in the full refactoring. Indeed, if we find a local refactoring sequence and its corresponding full refactoring sequence succeeds, then as shown earlier, the edits performed by the user are preserved (Lemma 3.3) and that the produced sequence cannot be trivially shortened (Lemma 3.5).

4. Implementation

We have implemented the approach from Section 3 in a tool called RESYNTH. The tool is built as a plug-in for Eclipse 4.2 [4]. Here we discuss how we designed RESYNTH to minimize the work needed to add new refactorings to the tool, and we present other salient implementation details.

4.1 Architecture

RESYNTH consists of a core refactoring search engine that invokes individual automated refactorings through a simple, uniform API. Beyond the standard functionality required of an automated refactoring (i.e., the ability to apply the refactoring to a program and detect pre-condition violations), RESYNTH also requires that each full refactoring implements its corresponding local refactoring as described earlier. Recall that each local refactoring takes a number of arguments as input. Conceptually, having an external search process choose those arguments violates modularity, as the search engine would need to know the meaning of each individual refactoring (in order to chose suitable arguments). Therefore, we require the developer to define a `successors`

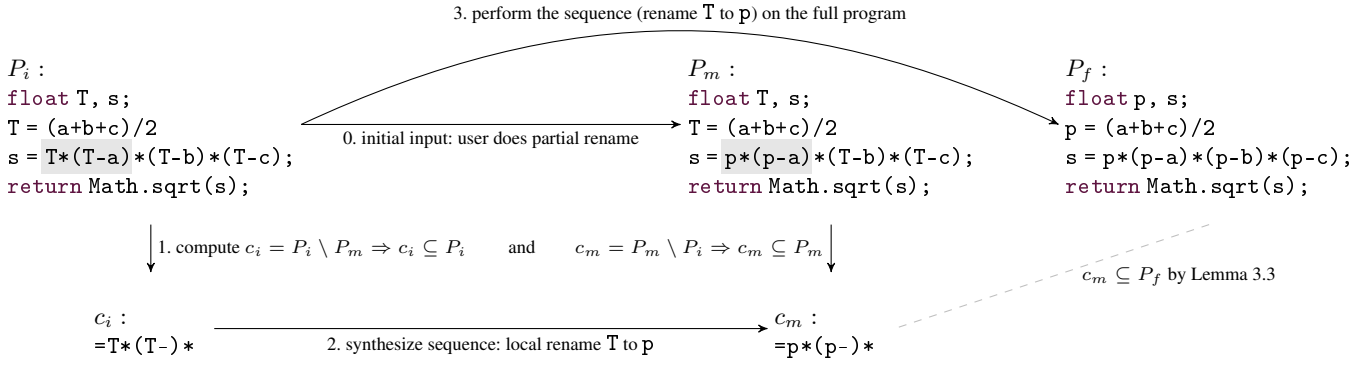

Figure 5. Example of synthesizing a refactoring sequence. Initially (stage 0), the user performs part of the rename (the user change is highlighted in both programs). Then c_i and c_m are computed (stage 1). Then, a sequence of one local rename is discovered (stage 2). Finally, the rename is applied to the full program (stage 3).

function associated with that local refactoring. The search engine simply invokes the function `successors` on all available refactorings to enumerate the search space.

Similar to a local refactoring, the `successors` function takes as input a tree. However, it produces a set of trees as output, by invoking the corresponding local refactoring with a set of possible arguments on the input tree. More formally, the `successors` function takes two parameters as input: i) the tree T_i representing the current local search state to be explored from, and ii) the target tree c_m (see Section 3.2). Given these parameters, `successors` must return a *finite* set of pairs $(T_o, \overline{args})$, where T_o is a successor tree obtained by applying a local refactoring to T_i with arguments $\overline{args}$ (e.g., the original and new names for a RENAME refactoring). We keep the set of arguments $\overline{args}$ in the returned pair in order to compute the full local refactoring sequence when the target tree c_m is reached.

The minimal additional functionality that RESYNTH requires of local refactorings makes adding new local refactorings to RESYNTH relatively easy. In particular, refactoring implementors need not concern themselves with details of *how* the search space is explored; this aspect is handled entirely within the core engine, using the techniques described in Section 3.4. Instead, they only need to specify *what* is the space to be searched, via the `successors` function.

Certain decisions made within the `successors` function of each refactoring may require some tuning. For refactorings with unbounded inputs (e.g., RENAME, which can take an arbitrary string as a new name), there can be an infinite number of successors for a given tree. In such cases, the refactoring implementor must decide what finite subset of the possible successors should be returned by `successors` (discussion of how our implemented refactorings handle this issue is described in Section 4.2). The amount of pre-condition checking to be performed within `successors` is also left to the refactoring implementor. In our experience, we found that performing aggressive pre-condition checking during successor computation was worthwhile, to reduce the size of the search space and to discover violations cheaply during local refactoring whenever possible.

To implement local refactorings efficiently, the input trees to `successors` include symbol information for all names, e.g., whether a name references a local variable, a field, a method, or some other entity. This information is useful during local pre-condition checking inside refactorings, e.g., to detect name conflicts. The initial information is computed from program P_i, and each refactoring must preserve the information appropriately for each successor tree it produces.

To further simplify the task of implementing `successor` functions, we provide utility functions that operate on trees and perform the following basic modifications:

1. Replace all occurences of a given symbol by a new tree.
2. Replace a node of the tree by a new subtree.
3. Insert a new statement in a tree.
4. Delete a statement from a tree.

Note that our trees are immutable, so these modifications actually produce new trees. These utilities were used across the refactorings we have implemented thus far, and we believe they will be useful for implementing other refactorings as well.

We now briefly compare our architecture to that of WitchDoctor [5]. Like RESYNTH, WitchDoctor also aims to make it easy to integrate new Eclipse refactorings into its search. To support a new refactoring in WitchDoctor, its effects have to be described using declarative rules that match changes in the AST. While this allows more high-level specifications than our `successors` function, it requires the AST resulting from the refactoring to be known already. This is true in WitchDoctor, where only a single refactoring application is detected at a time, but not in RESYNTH, where a refactoring may create an intermediate AST that is further changed by other refactorings before resulting in the final AST. Our ap-

proach thus is more powerful, but requires a bit more work to integrate a new refactoring.

4.2 Implementation Details

RESYNTH currently includes implementations of `successors` functions for RENAME, INLINE METHOD, EXTRACT METHOD, INLINE LOCAL, and EXTRACT LOCAL.[5]

Together, the refactorings above covered many of the interesting refactoring sequences we observed in real-world examples. Similar to the WitchDoctor system [5], our refactorings currently rely on underlying Eclipse refactoring implementations to perform full refactorings. As discussed in previous work [5], the Eclipse refactorings have not been engineered to run quickly in scenarios like these and our experiments confirm that each Eclipse refactoring needs around half a second to run. In a from-scratch implementation, more code could be shared between the lightweight local refactorings and the full refactoring, and global pre-condition checking could be optimized in a manner that could speed up the overall search.

As discussed in Section 4.1, our refactoring implementations are required to finitize a potentially-infinite set of successors in their `successors` functions. For RENAME, we limited the set of new names for an identifier to the name present at the same tree node in the final tree c_m or one additional fresh temporary name. The temporary name is useful for cases where the refactoring sequence requires names to be swapped. Because multiple name swaps can be composed sequentially, one temporary name is sufficient. The EXTRACT LOCAL and EXTRACT METHOD refactorings always use a deterministically-generated name for the new local variable or method: if needed, a subsequent RENAME refactoring can used to match the user's desired name.

RESYNTH contains around 3100 lines of Java code, the majority of which is the engine, the plugin and the evaluation code. Only 750 of the code lines are the local refactorings.

Example We briefly illustrate the work needed to add the INLINE LOCAL refactoring. First we implement the `successors` function. Recall that the `successors` function takes a tree T_i and returns a *finite* set of pairs $(T_o, \overline{args})$. For the INLINE LOCAL refactoring, each local variable declaration v with an initializer expression e in T_i produces one successor tree. Since T_i is a finite tree, the number of successors is finite. Each successor tree is computed as follows:

1. Let s be the resolved symbol of the variable v. First, we replace all nodes in T_i that resolve to the symbol s with the expression e. This replacement is in fact done via a utility method provided by RESYNTH.
2. Second, we delete the statement that declared the variable.[6]

Then, the produced tree is one successor T_o of T_i and the set of corresponding arguments $\overline{args}$ to T_o is only the variable v.

When we need to execute the full refactoring, we are given a program P, such that $T_i \subseteq P$ and the arguments of the local refactoring $\overline{args} = v$. To execute the full refactoring, all we need to do is find the AST node of the variable declaration v in P, and to call the `InlineTempRefactoring` Eclipse refactoring with this AST node as a parameter.

4.3 Limitations

Our current prototype has several limitations. Currently, we do not formally prove that our local refactorings perform edits that are consistent with the full Eclipse refactorings. Rather than verifying each local refactoring separately, in the future, we envision that the full refactoring is implemented in a way which enables one to derive a correct-by-construction local refactoring from it.

Also, some of our local refactorings do not currently handle the whole functionality of the full refactorings (for the proof-of-concept, we focused on the most common cases). For example, our local INLINE METHOD refactoring does not inline statements that are outside of the local tree. Due to this limitation, in some test cases, RESYNTH does not find the desired refactoring sequence, but finds a similar alternative one. This limitation can be avoided if the user edits include the deletion of the inlined function.

5. Evaluation

In this section, we present an initial evaluation of the RESYNTH tool. We first illustrate simple edits that we successfully used to perform individual refactorings with RESYNTH. We then show that RESYNTH was able to synthesize complex refactoring sequences required for several real-world examples. Furthermore, we show that the search strategy used by RESYNTH performed better than alternate strategies, both on real examples and on a synthetic benchmark suite.[7] Finally, we performed a small user study to see how programmers like the basic idea of synthesis-based refactoring, and to obtain feedback on RESYNTH and suggestions for improvements.

5.1 Individual Refactorings

As discussed in Section 4, RESYNTH currently includes implementations of five refactorings that are commonly implemented in modern IDEs. Since they are frequently used,

[5] We also extended Eclipse's implementation of this refactoring with an additional parameter that determines the position at which to declare the new local variable to avoid the problem mentioned in Section 2.

[6] Java allows several variable declarations per statement, but our local trees use slightly modified ASTs where each variable declaration is a separate statement.

[7] All results were obtained on a 64-bit Ubuntu 12.04 machine with a 4-core 3.5GHz Core i7 2700k processor and 16GB of RAM, running under Eclipse 4.2 with a 4GB maximum heap.

Example	steps	Source
ENCAPSULATE DOWNCAST	3	literature [6]
EXTRACT METHOD (advanced)	4	literature [6]
DECOMPOSE CONDITIONAL	6	literature [6]
INTRODUCE FOREIGN METHOD	2	literature [6]
REPLACE TEMP WITH QUERY	3	literature [6]
REPLACE PARAMETER WITH METHOD	3	literature [6]
SWAP FIELDS	3	literature [32]
SWAP FIELD AND PARAMETER	3	literature [25]
INTRODUCE PARAMETER	6	Stack Overflow[9]

Table 1. Realistic examples used to test RESYNTH.

IDEs often make these refactorings easy to invoke; e.g., Eclipse assigns each of them a direct keyboard shortcut. Nevertheless, RESYNTH provides an advantage when applying these refactorings individually, as they can all be invoked through a uniform interface. We confirmed that RESYNTH could successfully perform individual refactorings when given the following edits (performed between pressing the "Start" and "Complete" buttons):

Rename An entity `x` (method, field, or local variable) can be renamed by editing the declaration of `x` or any reference to use the new name.

Inline Local A local variable can be inlined simply by deleting its declaration.

Inline Method Similarly, a method `m` can be inlined at all sites by deleting `m`.

Extract Local To extract an expression e into a local `x`, one simply replaces e with `x`.

Extract Method With Holes To extract an expression e into a new method `m`, the user can replace e with an invocation of `m`, where the invocation includes the desired parameters for `m`. The final k statements in `m` can be extracted in a similar fashion.[8] Note that, as described in Section 2, this refactoring is more general than the standard EXTRACT METHOD, as it gives the user very fine-grained control over which expressions to pass as arguments to `m`. Hence, a refactoring sequence may be required to handle these edits.

Of course, a developer must learn that the edits outlined above will accomplish the desired refactorings before she can use them. However, we believe the edits are fairly intuitive, as they are a subset of the edits required to perform the refactorings manually. The simplicity of the edits, along with having a uniform interface instead of separate menu items / keyboard shortcuts for each refactoring, could ease the process of applying these refactorings in practice.

[8] We have not yet implemented support in our local refactoring for extracting multiple contiguous statements from the middle of a method.

[9] `http://stackoverflow.com/questions/10121374/referencing-callee-when-refactoring-in-eclipse`

5.2 Refactoring Sequences

Benchmarks To test RESYNTH's effectiveness for synthesizing refactoring sequences, we collected a set of examples that require a non-trivial refactoring sequence to achieve the user's desired transformation. The examples are listed in Table 1, along with information on where the example was obtained. The first five examples are transformations presented in Fowler [6] that are not implemented in Eclipse, but can be accomplished via a sequence of the refactorings we have implemented. We also include two examples of name swapping refactorings that have appeared in the literature [25, 32], each of which requires a non-obvious introduction of a temporary name in the refactoring sequence. Finally, the INTRODUCE PARAMETER example, found online, can be achieved via a complex sequence of six of our implemented refactorings. While Eclipse provides an implementation of INTRODUCE PARAMETER, it fails to handle this example. The EXTRACT METHOD (advanced) and SWAP FIELDS examples were previously discussed in Section 2.

Four other examples from Fowler's book could be composed from additional Eclipse refactorings for which we have not yet implemented local refactorings.

We also generated a suite of random benchmarks to perform further stress testing of RESYNTH, as follows. For each benchmark, we first generated a Java class C containing a sequence of static field declarations followed by a sequence of static methods. Each static method returns a random expression e, generated by using binary operators to combine constants, local variable and field references, and method invocations up to some depth. Given C, we then generated an edited version C' by performing a random sequence of edits like those described in Section 5.1 to induce individual refactorings. Each (C, C') pair is a benchmark that tests whether RESYNTH can discover a sequence of refactorings on C that preserves the edits in C'. For our experiments, we generated 100 such benchmark inputs, with five fields, five methods, and expression depth of three for C, and two random edits to produce C'. We found these parameters to create benchmarks that were somewhat more challenging than our real examples (see below), while still remaining realistic. All of our benchmarks (real and synthetic) are available online at `http://www.srl.inf.ethz.ch/resynth.php`.

Success Rate Table 2 shows results from applying RESYNTH to our real-world and synthetic benchmarks. RESYNTH successfully generated a refactoring sequence for all but two of the real-world examples and for 84% of the synthetic examples. We bounded the search to a maximum of 20,000 trees for these results; we explore different bounds shortly. From each explored tree, we run the `successors` function, which returns multiple successor trees and effectively explores a higher number of refactorings than the number of trees.

On average applying a full Eclipse refactoring takes about 0.5 seconds and as we explore thousands (1296 for our real world tests) of refactorings in order to discover the desired

Metric	Dataset	
	Real	**Synthetic**
Number of tests	9	100
Avg. number of trees searched	87	3752
Avg. number of successors in a search	1296	105310
Avg. search time	0.014s	1.629s
Avg. Eclipse refactoring time	2.953s	1.654s
Refactoring sequence length		
1 refactoring	0	2
2 refactorings	1	45
3 refactorings	5	7
4 refactorings	1	15
5 refactorings	0	3
6 refactorings	2	2
7 refactorings	0	9
8 refactorings	0	0
9 refactorings	0	1
Failure to find sequence after 20000 searched trees	0	16

Table 2. Results for our refactoring sequence search. The A^* heuristic function weights (see Section 3.4) were a_1 = 0.125 and a_2 = 0.25.

sequence, had we performed the search with full refactorings instead of local refactorings, the process would have taken at least 10 minutes, clearly an undesirable response time when performing interactive edits.

The two real-world examples for which RESYNTH did not find the sequence from Fowler's book were ENCAPSULATE DOWNCAST and REPLACE PARAMETER WITH METHOD. This is due to a limitation (see Section 4.3) in our `successors` function for INLINE METHOD. It did, however, find another valid refactoring sequence that matches the same edits.

As shown by the mean search time and mean number of trees searched, the synthetic examples were significantly more challenging than the real-world examples on average. Also note that the time required to apply the full Eclipse refactorings after the sequence was discovered was roughly equal to the entire search time for synthetic benchmarks, and much larger than the search time for real examples, indicating the need for local refactorings.

Out of the 16 examples that we fail to solve, nine require minimum 11 steps to accomplish, four tests require minimum 9 steps and three tests require minimum 7 steps. These are examples where the A^* search would need to explore more than 20000 trees until it can find a refactoring sequence. Overall, the scalability results for our technique are quite encouraging.

Table 3 shows how our success rate on synthetic benchmarks is affected by the search bound. Increasing the search space leads to a slightly lower failure rate, but it also significantly increases the average search time. Further testing with users is required to discover an appropriate bound in practice.

Metric	Search space limit (# trees)		
	20,000	100,000	500,000
Num. failed tests	16	15	9
Avg. number of searched trees	3752	15900	64392
Avg. search time	1.629s	7.802s	34.473s

Table 3. Success rate on synthetic benchmarks with different search bounds. Tested with $a_1 = 0.125$ and $a_2 = 0.25$.

Alternate Search Strategies We tested alternate strategies for searching for refactoring sequences by tuning the coefficients for the heuristic function $h(t) = a_1 h_1(t) + a_2 h_2(t)$, described previously in Section 3.4. Note that setting $a_1 = 0$ and $a_2 = 0$ disables the heuristic function, making the A^* search perform a straightforward breadth-first search.

Results for different values of a_1 and a_2 with a search budget of 20,000 trees appear in Table 4. The a_1 and a_2 values we chose for our other experiments are in bold, along with the best values in the other columns. The naïve breadth-first search ($a_1 = a_2 = 0$) fails for 2 of the real examples and 32 of the random tests, corresponding to the tests requiring four or more refactoring steps. A linear combination of h_1 and h_2 performs better than using each of the heuristic functions alone, for both real examples and the synthetic tests. In RESYNTH, we selected $a_1 = 0.125$ and $a_2 = 0.25$ as these values had best results on the real tests and close to the best results on the synthetic tests.

5.3 User Study

Although RESYNTH is currently a research prototype and somewhat unpolished, we conducted a small, informal user study to gauge how useful programmers find the concept of refactoring with synthesis, and to get some feedback on how to improve our tool.

We recruited six participants: two undergraduate students, three graduate students, and one professional. All of the participants had prior experience with Java programming (between one and five years), and all of them were familiar with Eclipse. One of them was a proficient user of Eclipse's built-in refactoring tools, while the other participants had little to no experience with tool-supported refactoring.

After a brief demonstration of RESYNTH, we gave each participant a set of three refactoring tasks based on examples from Fowler's textbook [6], and asked them to complete the tasks using either RESYNTH, Eclipse's built-in refactorings, or manual editing.[10] For one of the tasks (Task 3), RESYNTH cannot, in fact, find the desired solution, but comes up with a slightly different solution.

Four of our participants were able to complete all three tasks using RESYNTH, and two of them failed on one of

[10] A complete description of the tasks is available at `http://www.srl.inf.ethz.ch/resynth.php`.

Search Parameters		Real Examples		Synthetic Tests	
Edit distance weight a_1	Expr. distance weight a_2	Num. failed tests	Avg. num. searched trees	Num. failed tests	Avg. num. searched trees
0.000	0.000	2	5306	32	6943
0.000	0.125	1	3076	26	5373
0.000	0.250	**0**	184	26	5348
0.000	0.500	**0**	119	24	4537
0.000	1.000	1	2350	16	3316
0.125	0.000	**0**	1248	25	5065
0.125	0.125	**0**	115	19	4642
0.125	**0.250**	**0**	**87**	16	3752
0.125	0.500	**0**	122	15	3396
0.125	1.000	**0**	1154	**14**	**3243**
0.250	0.000	**0**	291	21	4456
0.250	0.125	**0**	281	18	3885
0.250	0.250	1	223	18	3694
0.250	0.500	1	153	17	3485
0.250	1.000	1	623	**14**	3516
0.500	0.000	2	358	26	4481
0.500	0.125	1	189	23	4401
0.500	0.250	1	158	22	4274
0.500	0.500	1	158	20	4114
0.500	1.000	1	465	18	4092
1.000	0.000	2	3033	24	4704
1.000	0.125	1	641	24	4796
1.000	0.250	1	637	25	4786
1.000	0.500	1	477	26	4704
1.000	1.000	1	768	22	4490

Table 4. Search space with different parameters of the heuristic function for the A* search (When $a_1 = a_2 = 0$, the search is a breadth-first search.)

the tasks (though not on the same one): one of them tried to manually compose the transformation out of individual small refactorings, but became confused and ultimately gave up; the other participant was unable to find the right manual edits to perform. In the future, we will investigate improvements to our search techniques and user interface to reduce the number of cases where intuitive edits do not lead to the desired solution.

Only two participants noticed that the solution found by RESYNTH for Task 3 was not quite the expected solution, but one of them thought the alternative solution was good enough.

After they had finished the tasks, we asked the participants whether they thought a tool like RESYNTH could be useful, and what improvements they would like to see. Four participants responded that they did find it useful, although one of them qualified his response by saying that he would not trust the tool on a complex code base. This was because during the experiment he discovered a bug in the Eclipse implementation of INLINE LOCAL VARIABLE, which led him to doubt the reliability of Eclipse's built-in refactoring tools on which RESYNTH is built. Of the two participants who were not convinced of the usefulness of RESYNTH, one was not comfortable with the idea of performing long sequences of refactorings in one go and instead preferred to compose refactorings by hand, while the other was already very familiar with the Eclipse refactoring tools and did not see the need for another tool.

Finally, the participants suggested three improvements: (1) handling uncompilable code; (2) eliminating the "Start Refactoring" button; (3) adding support for more refactorings. The third suggestion is, in principle, quite easy to implement by writing local equivalents for more built-in Eclipse refactorings. The first suggestion could be addressed by using a more robust error-correcting parser, though it remains to be seen whether this would adversely affect the quality of refactoring suggestions.

The second suggestion is more difficult to accommodate. The participant explained that it was easy to forget to click the "Start Refactoring" button before initiating a refactoring, which is similar to the "late awareness dilemma" described by Ge et al. [7]. The participant suggested that instead of setting an explicit checkpoint, the IDE should try to infer a likely checkpoint, such as the last version of the program that compiled. Given that another participant specifically asked for support for refactoring uncompilable programs, however, it seems unlikely that this heuristic would suit all users equally well.

While it is impossible to generalize from a study with such a limited number of participants, we think that our results show that the idea of synthesis-based refactoring has promise, and with further improvements a tool like RESYNTH could usefully complement the built-in Eclipse refactoring tools.

6. Related Work

In this section, we discuss some of the more closely related work in the areas of program refactoring and synthesis.

Refactoring The idea of *refactoring catalogs* that list commonly used refactoring operations and specify their behavior already appeared in Opdyke's thesis [22], one of the earliest works in the field. Another influential refactoring catalog was compiled by Fowler in his book on refactoring for Java [6]. Refactoring tools in current Java IDEs still tend to follow this catalog in the repertoire of refactorings they offer and the names assigned to them.

However, Murphy-Hill et al. [19] found in a landmark study on refactoring practices that while programmers refactor frequently, about 90% of refactorings are performed by hand, even where tool support is available. These numbers were confirmed in a recent, more detailed study by Negara et al. [20]. To make refactoring tools more attractive to programmers, some authors proposed user-interface improvements [18], but even such improved tools still suffer from discoverability issues: the programmer may not know that an automated implementation is available for a refactoring

they perform by hand, or they may start a manual refactoring before remembering that tool support is available.

To address these issues, Ge et al. [7] and Foster et al. [5] proposed systems that observe a programmer's editing operations and try to discover editing patterns suggestive of refactorings. If they discover such a pattern, the user is offered the choice of completing the refactoring task using a refactoring tool. Similarly, Lee et al. [16] advocate an approach in which refactoring operations can be initiated through drag-and-drop gestures.

In contrast to our work, all these systems can only detect and suggest applications of a single refactoring.

Negara et al. [20] provide evidence to suggest that programmers often perform several refactorings in sequence to achieve a single transformation. Vakilian et al. [35] further show that programmers do this even if there is tool support for a single, larger refactoring that would perform this transformation in one go, preferring the higher level of predictability and control afforded by step-by-step refactorings.

Our approach gives the programmer full flexibility: they can either only perform a small set of edits and use RESYNTH to perform the associated small-scale refactoring, or perform more edits and let RESYNTH infer a sequence of refactorings to achieve a large-scale transformation.

Even more flexibility is achieved by languages for scripting refactorings such as JunGL [39] or Jackpot [15], which allow programmers to implement their own custom refactorings. Given the effort involved in learning a new language and API, however, it seems unlikely that any but the most determined developers will use such systems on a regular basis.

Currently, RESYNTH is built on top of the refactorings provided by Eclipse JDT. This means that it will sometimes fail to infer a sequence of refactorings if some intermediate step cannot be performed using the available refactorings. This problem could be alleviated by instead basing RESYNTH on more fine-grained transformations as proposed in the literature [25, 27].

Steimann et al. [31, 32] take a more radical approach which abolishes the notion of individual atomic refactorings altogether. Instead, a given program's static semantics (name binding, overriding, accessibility, etc.) is encoded as a set of constraints such that any program that fulfills the same constraints must be semantically equivalent to the original program. Refactoring is then simply a search problem in the space of all programs satisfying the same constraints. While their approach is appealing for its simplicity and generality, it has only been shown to work for a very restricted set of refactorings (essentially, refactorings for renaming and moving program elements). In contrast to BeneFactor, WitchDoctor or RESYNTH, this approach cannot be implemented on top of an existing refactoring tool and instead requires a complete reimplementation of the entire refactoring engine.

In a slightly different context, several researchers have considered the problem of detecting refactorings or other forms of systematic code changes from program revision histories to better understand program evolution [13, 24, 41], and to adapt clients of evolving frameworks and libraries [3, 33, 42]. Vermolen et al. [40] consider the same problem in a modeling setting, where models need to be migrated when their metamodel changes. Since the goal of these approaches is to infer completed refactorings, they can assume that all edits arising from the refactorings have already been performed. RESYNTH, on the other hand, assumes that the user only performed some edits by hand, and that further edits may be needed to complete the intended refactorings, which leads to a much larger search space.

Synthesis There has long been significant interest in using program synthesis techniques aimed to simplify various software development tasks. For a recent survey see Gulwani [9]. Here we describe the techniques that are most closely related to our work, focusing on techniques for program completion.

Prospector [17] introduced a synthesis procedure where given an input type I and output type O, the tool statically discovers (by examining the API specifications) a sequence of API calls which, starting from an I object, produce an O object. PARSEWeb [34] addresses the same problem, but its search is guided by existing source code mined from the web, which helps eliminate many undesirable API sequences. More recent work [10, 23] searches for suitable expressions of a given type at a particular program, considering more of the program context around that point. These systems rely on good ranking algorithms to handle large numbers of potentially-suitable expressions. In constrast to the aformentioned static approaches, MatchMaker [43] synthesizes code based on observed API usage in dynamic executions of real-world programs.

The concept of starting with partial programs and completing them has recently been explored in various synthesis works. The sketching approach [28] takes as input a program with holes (a "sketch"), where the user specifies a space of possible expressions which can be used to fill the holes. The synthesizer then searches for correct program completions (completions which satisfy a given property). The idea has been applied to various application domains including bitciphers [30] and concurrency [29]. In the context of concurrency, recent work has also devised ways to infer various synchronization constructs in the context of concurrent programming. Examples include atomic sections [38], memory fences [14] and conditional critical regions [37]. These approaches complete an existing concurrent program that may be potentially incorrect.

Similarly to the above works, our work can also be seen as solving an instance of the program completion problem. Here, our objective is to complete an intermediate program P_m into another program P_f, such that P_f is a refactoring

of P_i. In some sense, P_i serves as the "specification" for our search in the sense that P_i and P_f should be semantically equivalent. However, unlike the previous work, we do not aim to complete P_m into P_f directly—such an approach would require partial refactoring transformations to complete the edits performed by the user, and it would be quite difficult to even enumerate the space of such transformations. Our synthesizer *does not* make any attempt to complete P_m, but instead begins its search from the "specification" P_i, using the intermediate program P_m to decide when the search has been successful.

We believe that there are many interesting future directions in combining ideas from the fields of refactoring and synthesis. For instance, suppose that our approach cannot find a refactoring sequence from P_i to P_m. An interesting direction here would be to allow P_m to represent not a single program but a partial program which symbolically represents a set of programs (e.g. a sketch). Then, the synthesizer would try to discover a refactoring sequence starting from P_i where the result would be correct if it contains *any* of the programs symbolically represented by P_m. If such a refactoring sequence is discovered, it would still be an acceptable solution (if it passes the global pre-conditions). This approach would give the user more freedom in expressing the partial changes to the program P_i. In addition, in terms of the synthesis algorithm, an interesting direction could be in formulating the problem as a logical formula and then using an SMT solver to perform the search.

7. Conclusion and Future Work

We have presented a new approach to automated refactoring, inspired by synthesis from examples. In our approach, the user describes a desired transformation by performing some of the required edits on an initial program P_i, and the synthesizer searches for a sequence of refactorings of P_i that includes the edits. We described a search strategy based on the concept of local refactoring combined with a tuned heuristic search. We implemented our techniques in a tool called RESYNTH and showed that it can already handle challenging real-world examples.

In future work, we plan to: i) implement more local refactorings in RESYNTH and further optimize its search techniques, ii) add a user interface for rejecting a proposed refactoring sequence and continuing the search (the tool already has the option to display the discovered sequence to the user), iii) handling code which does not compile, and iv) generalize our approach to transformations beyond refactoring, e.g., generation of boilerplate code: modern Java IDEs include many such transformations, and our techniques could further ease their usage.

References

[1] ABADI, A., ETTINGER, R., AND FELDMAN, Y. A. Reapproaching the Refactoring Rubicon. In *WRT* (2008).

[2] BECK, K., AND ANDRES, C. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[3] DIG, D., COMERTOGLU, C., MARINOV, D., AND JOHNSON, R. Automated Detection of Refactorings in Evolving Components. In *ECOOP* (2006).

[4] Eclipse 4.2. `http://www.eclipse.org/eclipse4`.

[5] FOSTER, S. R., GRISWOLD, W. G., AND LERNER, S. WitchDoctor: IDE Support for Real-time Auto-completion of Refactorings. In *ICSE* (2012).

[6] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2000.

[7] GE, X., DUBOSE, Q. L., AND MURPHY-HILL, E. R. Reconciling Manual and Automatic Refactoring. In *ICSE* (2012).

[8] GRISWOLD, W. G. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis, University of Washington, 1991.

[9] GULWANI, S. Dimensions in program synthesis. In *ACM PPDP* (2010).

[10] GVERO, T., KUNCAK, V., KURAJ, I., AND PISKAC, R. On Complete Completion using Types and Weights. Tech. Rep. 182807, EPFL, 2012.

[11] HART, P. E., NILSSON, N. J., AND RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics* (1968), IEEE, pp. 100–107.

[12] IntelliJ IDEA 12 Community Edition. `http://www.jetbrains.com/idea`.

[13] KIM, M., NOTKIN, D., AND GROSSMAN, D. Automatic Inference of Structural Changes for Matching Across Program Versions. In *ICSE* (2007).

[14] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. *ACM SIGACT News* (2012).

[15] LAHODA, J., BEČIČKA, J., AND RUIJS, R. B. Custom Declarative Refactoring in NetBeans. In *WRT* (2012).

[16] LEE, Y. Y., CHEN, N., AND JOHNSON, R. E. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *ICSE* (2013).

[17] MANDELIN, D., XU, L., BODÍK, R., AND KIMELMAN, D. Jungloid mining: helping to navigate the API jungle. In *ACM PLDI* (2005).

[18] MURPHY-HILL, E. R., AND BLACK, A. P. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *ICSE* (2008).

[19] MURPHY-HILL, E. R., PARNIN, C., AND BLACK, A. P. How We Refactor, and How We Know It. *TSE 38*, 1 (2012).

[20] NEGARA, S., CHEN, N., VAKILIAN, M., JOHNSON, R. E., AND DIG, D. A Comparative Study of Manual and Automated Refactorings. In *ECOOP* (2013).

[21] NetBeans 7.0.1. `http://netbeans.org`.

[22] OPDYKE, W. F. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[23] PERELMAN, D., GULWANI, S., BALL, T., AND GROSSMAN, D. Type-directed completion of partial expressions. In *ACM PLDI* (2012).

[24] PRETE, K., RACHATASUMRIT, N., SUDAN, N., AND KIM, M. Template-based Reconstruction of Complex Refactorings. In *ICSM* (2010).

[25] REICHENBACH, C., COUGHLIN, D., AND DIWAN, A. Program Metamorphosis. In *ECOOP* (2009).

[26] ROBERTS, D., BRANT, J., AND JOHNSON, R. E. A Refactoring Tool for Smalltalk. *TAPOS 3*, 4 (1997).

[27] SCHÄFER, M., VERBAERE, M., EKMAN, T., AND DE MOOR, O. Stepping Stones over the Refactoring Rubicon – Lightweight Language Extensions to Easily Realise Refactorings. In *ECOOP* (2009).

[28] SOLAR-LEZAMA, A. The sketching approach to program synthesis. In *APLAS* (2009).

[29] SOLAR-LEZAMA, A., JONES, C. G., AND BODIK, R. Sketching concurrent data structures. In *ACM PLDI* (2008).

[30] SOLAR-LEZAMA, A., TANCAU, L., BODIK, R., SESHIA, S., AND SARASWAT, V. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.* (2006).

[31] STEIMANN, F., KOLLEE, C., AND VON PILGRIM, J. A Refactoring Constraint Language and Its Application to Eiffel. In *ECOOP* (2011).

[32] STEIMANN, F., AND VON PILGRIM, J. Refactorings Without Names. In *ASE* (2012).

[33] TANEJA, K., DIG, D., AND XIE, T. Automated Detection of API Refactorings in Libraries. In *ASE* (2007).

[34] THUMMALAPENTA, S., AND XIE, T. Parseweb: a programmer assistant for reusing open source code on the web. In *ACM/IEEE ASE* (2007).

[35] VAKILIAN, M., CHEN, N., MOGHADDAM, R. Z., NEGARA, S., AND JOHNSON, R. E. A Compositional Paradigm of Automating Refactorings. In *ECOOP* (2013).

[36] VAKILIAN, M., CHEN, N., NEGARA, S., RAJKUMAR, B. A., BAILEY, B. P., AND JOHNSON, R. E. Use, Disuse, and Misuse of Automated Refactorings. In *ICSE* (2012).

[37] VECHEV, M., YAHAV, E., AND YORSH, G. Inferring synchronization under limited observability. In *TACAS* (2009).

[38] VECHEV, M., YAHAV, E., AND YORSH, G. Abstraction-guided synthesis of synchronization. In *ACM POPL* (2010).

[39] VERBAERE, M., ETTINGER, R., AND DE MOOR, O. JunGL: A Scripting Language for Refactoring. In *ICSE* (2006).

[40] VERMOLEN, S., WACHSMUTH, G., AND VISSER, E. Reconstructing Complex Metamodel Evolution. In *SLE* (2011).

[41] WEISSGERBER, P., AND DIEHL, S. Identifying Refactorings from Source-Code Changes. In *ASE* (2006).

[42] XING, Z., AND STROULIA, E. API-Evolution Support with Diff-CatchUp. *TSE 33*, 12 (2007).

[43] YESSENOV, K., XU, Z., AND SOLAR-LEZAMA, A. Data-driven synthesis for object-oriented frameworks. In *ACM OOPSLA* (2011).