Transforming programs between APIs with Many-to-Many Mappings

Abstract

Transforming programs between two APIs or different versions of the same API is a common software engineering task. However, existing languages supporting for such transformation cannot satisfactorily handle the cases when the relations between elements in the old API and the new API are many-to-many mappings: multiple invocations to the old API are supposed to be replaced by multiple invocations to the new API. Since the multiple invocations of the old APIs may not appear consecutively and the variables in these calls may have different names, writing a tool to correctly cover all such invocation cases is not an easy task.

In this paper we propose a novel *guided-normalization* approach to address this problem. Our core insight is that programs in different forms can be semantics-equivalently normalized into a basic form guided by transformation goals, and developers only need to write rules for the basic form to address the transformation. Based on this approach, we design a declarative program transformation language, PATL, for adapting Java programs between different APIs. PATL has simple syntax and basic semantics to handle transformations only considering consecutive statements inside basic blocks, while with guided-normalization, it can be extended to handle complex forms of invocations. Furthermore, PATL ensures that the user-written rules would not accidentally break def-use relations in the program.

We formalize the semantics of PATL on Middleweight Java and prove the semantics-preserving property of guided-normalization. We also evaluated our language with three non-trivial case studies: i.e. updating Google Calendar API, switching from JDom to Dom4j, and switching from Swing to SWT. The result is encouraging; it shows that our language allows successful transformations of real world programs with a small number of rules and a small number of manual resolutions.

1 Introduction

Modern programs depend on library APIs, and when those APIs change, usually many places of the client programs need to be changed. As a result, it is desirable for tool vendors to provide program transformation tools that automate the task of transforming client programs. Practically, there are two types of such program transformation tools. One is for API upgrade [11, 19]: when a new version of the API is released, the tool is developed to upgrade the client programs to work with the new API. For example, a tool named 2to3 script [30] is provided to help users migrate programs from Python 2 to Python 3. Another is for API switching [34], where the tool migrates programs from one platform to another platform. For example, RIM provides a migration tool for transforming Android applications to Blackberry applications in order to attract more developers to the Blackberry platform.

Given the importance of developing program transformation tools, dedicated program transformation languages have been proposed, such as SmPL [27], Stratego [6], TXL [9], Syntax Macros [38], Twinning [24] and SWIN [18]. These languages typically allow the developers to describe a set of syntactically rewriting rules. A rule usually consists of a pattern that matches a piece of code and an action to be performed on the matched code, where a typical action is to replace the original code with a new piece of code.

For example, the rule `rSetAlign` in the left of Figure 1 is a simple rule to transform a method invocation in Swing to its counterpart in SWT. The rule is written in PATL (Patch-like Transformation Language), the transformation language to be introduced in the

```
//rule rSetAlign
(jb: JButton->Button, align: int->int){
 - jb.setAlignmentX(align);
 + jb.setAlignment(align);
}
```

```
//rule rButton
(jb: JButton->Button,
 parent: JPanel->Composite) {
 - jb = new JButton();
 - parent.add(jb);
 + jb = new Button(parent, SWT.PUSH);
}
```

Figure 1 Two transformation rules used in transformation of programs between Swing to SWT.

```
1 //Case 1:
2 jb = new JButton();
3 if(parent != null) {
4   parent.add(jb);
5 }
```

```
1 //Case 2:
2 jb = new JButton();
3 s = jb.getUIClassID();
4 parent = new JPanel();
5 parent.add(jb);
```

```
1 //Case 3:
2 jb = new JButton();
3 defaultButton = jb;
4 parent.set(defaultButton);
```

Figure 2 Three examples of non-consecutive API invocations

paper, which adopts a patch language style syntax used previously in SmPL [27]. This rule matches a call to method `JButton.setAlignmentX` and replaces it by a call to method `Button.setAlignment`. The line marked with '-' describes a pattern to match a statement, while the line marked with '+' describes a pattern for generating statements. Parameter `jb` is a metavariable used to match a variable of type `JButton`, and the notation `JButton->Button` indicates that class `JButton` in the old API corresponds to class `Button` in the new API.

Though program transformation languages have greatly reduced the difficulty of writing program transformations, it is still difficult to deal with many-to-many mappings between APIs. *Many-to-many mapping* means that a sequence of invocations to the old API is mapped to a sequence of invocations to the new API, and this mapping is minimal: there exists no mapping from a subsequence of invocations to the old API to a subsequence of the new API. As shown by existing studies [34, 4], *many-to-many mappings* are very common in transformation between APIs. Concretely, there are two key challenges in dealing with many-to-many mappings.

Challenge 1. The first challenge is that the same sequence of API invocations at runtime can be specified in many different forms in the code, and a transformation should consider all such possible forms. Let us consider transformation rule `rButton` in the right of Figure 1. In this case, the class `JButton` is mapped to the class `Button`, and the two calls in the Swing API are merged into one call to the constructor of `Button` in SWT. While this rule captures the basic forms of consecutively invoking the respective methods, many other different forms may produce the sequential invocation to these two statements, such as the those in Figure 2.

In this first case, every time the program enters the if-branch, the calls to `jb = new JButton()` and `parent.add(jb)` actually form consecutive calls defined in the pattern, while in the else-branch, such consecutive calls will not be produced. In the second case, though the two invocations are not consecutive, their behavior is equivalent to consecutive invocations (In this case, the third statement `parent = new JPanel()` has no dependency on the first two statements, nor does the second statement depend on the last two statements. So the whole execution is equivalent to an execution of statements 4, 2, 5, 3, where 2 and 5 are consecutive.) and thus should be transformed. In the third case, though the argument `defaultButton` in the third statement is different from `jb` in the first statement, they match the patterns in the rules as these two variables are aliases and they both refer to the same object. None of the three cases are directly captured by rule `rButton`.

Researchers have noticed this problem and have proposed different mechanisms to *match* the above cases, such as flow-based matching [7] and context-sensitive matching [9]. Given a rule that is similar to the rule `rButton`, these approaches can identify statements that may produce the same sequence of invocations at runtime, and return these statements as output. For example, in the first program in Figure 2, these approaches can identify the match between the rule `rButton` and the two statements in lines 2,4.

However, safely transforming these matched statements is challenging: contexts for matched statements should be considered and thus it is difficult to transform them uniformly using a simple strategy. For example, a common strategy used in existing approaches is to specify which statements in the '+' block replace which statements in the '-' block[a]. In this example, we can specify either the generated statement to replace the first statement marked with '-' or replace the second one. However, neither strategy can correctly handle even a simple case like Case-2 in Figure 2. Replacing the first statement (left program below) will lead to the problem that the `parent` used in the first statement (line 2) is either uninitialized or captures some undesirable value defined by the previous context. Similarly, replacing the second statement will lead to the fact that the statement `jb.getUIClassID()` refers to a variable `jb` undefined or defined by some undesirable object in the previous context.

```
1 //jb = new JButton();
2 jb = new Button(parent, SWT.PUSH);
3 /* other rules will transform the
      following two statements */
4 s = jb.getUIClassID(();
5 parent = new JPanel();
6 //parent.add(jb);
```

```
1 //jb = new JButton();
2 /* other rules will transform the
      following two statements */
3 s = jb.getUIClassID(();
4 parent = new JPanel();
5 //parent.add(jb);
6 jb = new Button(parent, SWT.PUSH);
```

To correctly handle such cases in existing languages [27, 6, 9], we have to write rules separately for each different case, capturing all possible contexts for all transformations that need to be performed. This adds a lot of burdens to the developers, and also there is no guarantee on whether these more complex patterns preserve the same meaning expressed in the basic rule.

In this paper, we present a novel transformation technique, *guided normalization*, which extends the notion of syntactical transformation expressed with a simple patch rule into a more expressive transformational semantics that can *match* and *transform* statements in control flow graphs. Using our approach, the developer only has to specify the transformation rules for consecutive blocks. The system first finds matched statements in a way similar to existing approaches, and for code pieces that cannot be directly transformed, a guided normalization process is performed: the system performs a set of semantics-preserving transformations to the program so that the matched statements become a consecutive block and can be directly transformed by simply syntactic substitution. If the normalization cannot be performed, a warning is generated to the user.

In this paper, we consider several semantics-preserving transformation primitives that are verified and widely used in compiler optimization [37, 1, 8, 14] and refactoring [33, 26, 31], including (1) *shifting* unconditional statements into all conditional branches when data dependency[b] does not break, (2) *swapping* two adjacent statements that have no data dependency with each other, and (3) *renaming* variables to their must-aliases or fresh names. Applying the three primitives on the three cases above, we can get the following programs, all of which can be directly matched and transformed by `rButton`.

```
//Case 1':
if(parent != null) {
  jb = new JButton();
  parent.add(jb);
}
```

```
//Case 2':
parent = new JPanel();
jb = new JButton();
parent.add(jb);
s=jb.getUIClassID(();
```

```
//Case 3':
jb = new JButton();
parent.set(jb);
defaultButton = jb;
```

[a] This strategy can be implemented as context-sensitive rules in TXL [9], dynamic rules in Stratego [6], and standard placement of '+' statements in SmPL [27].

[b] Note there are three types of data dependencies: read-after-write, write-after-read, and write-after-write. In this paper, when we use the term "data dependency", we refer to all the three.

A particular difficulty in designing guided-normalization algorithm is that several match instances can intertwine with each other, thus, a global solution is necessary to ensure that the normalization guided by one rule would not prevent the normalization guided by another rule. In our approach, we globally encode constraints brought by matches and dependencies, and then normalize the program based on the solution to the constraints.

Challenge 2. With guided normalization, we can ensure that code pieces in different forms are transformed as a consecutive block. However, the basic transformation specified by the user rules may also lead to undesirable conflicts with other code pieces. Here we focus on one type of such conflicts: breaking def-use relation. An example of breaking def-use relation is shown below.

```
1 //BadRule:
2 (x: ClassX->ClassX2,
3  y: ClassY->ClassY2) {
4   - x = new ClassX();
5   - y.add(x);
6   + y = new ClassY2(x);
7 }
```

```
1 //Program
2 y = System.defaultY();
3 x = new ClassX();
4 y.add(x);
5 SomeUse(y);
6 SomeOtherUse(x);
```

$\xrightarrow{\text{BadRule}}$

```
1 //Transformed Program
2 y = System.defaultY();
3 y = new ClassY2(x);
4 SomeUse(y);
5 SomeOtherUse(x);
```

As we can see, BadRule breaks the def-use relation in the program. The value `y` used in `SomeUse(y)` was defined by line 2 in the original program (left) and now is defined by line 3 in the transformed program (right). Similarly, the value of `x` in `SomeOtherUse(x)` was previously defined, and becomes undefined after the transformation. This problem is caused by the definition of BadRule, where a definition of `y` is undesirably added and a definition to `x` is undesirably removed.

We solve this problem by providing a static checker to check the def-use safety by looking into the set of transformation rules. Though this restricts the forms of the patch patterns to write, we will show in the evaluation section that such restriction will not damage our expressiveness in expressing API transformation problem.

Contributions. Concretely, this paper makes the following contributions.

- We propose a novel transformation technique, *guided normalization*, for program transformation with many-to-many mappings. In our approach, the developers specify the transformations only for consecutive method invocations, and the system automatically applies the transformation to many different forms via guided normalization.
- We design a patch-rule like programming language, PATL, to transform programs between APIs, and provide a static checker to ensure block-level transformation safety in terms of def-use relation. We formalize the semantics of core PATL atop of Middleweight Java [5], and prove (1) the guided normalization is semantics-preserving, and (2) the transformation never breaks def-use relations.
- We have evaluated our language with three non-trivial case studies for Google Calendar API updates, switching from JDom to Dom4j, and switching from Swing to SWT. The evaluation shows that our language allows successful transformations of real world programs with a small number of rules, requiring only a small amount of manual resolutions.

In the rest of the paper, we first present the syntax and basic semantics of PATL (Sections 2, 3). Then we describe how we address the two challenges (Sections 4, 5). Next, we present the implementation (Section 6) and evaluation (Section 7). Finally, we discuss related work (Section 8) and conclude the paper (Section 9).

2 PATL Syntax

2.1 Background: Middleweight Java

Our discussion of PATL is based on a formal imperative core of the Java language, Middleweight Java (MJ) [5]. To simplify the formal presentation, we only consider MJ in three-address form, and in Section 6 we shall discuss how to transform a program into and out of three-address form.

In three-address MJ, arguments of method invocations or object constructions, field access targets, `if` condition expressions and `while` condition expressions are limited to variables. Besides all local variables in a method body are declared before the statements. The syntax of three-address Middleweight Java is presented in Figure 3.

In the formal notations, we use the bar notation similar to that adopted by Pierce [29] for repetitive elements: $\bar{a}$ indicate a sequence $a_1, a_2, ..., a_n$, and all operations defined on single values expand component-wisely along with the sequence, e.g. $\bar{C}\ \bar{f}$, is equal to $C_1\, f_1, \cdots, C_n\, f_n$, where n is the length of $\bar{C}$ and $\bar{f}$.

$$
\begin{array}{rcll}
p & ::= & \overline{cd} & \text{(Program)} \\
cd & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{fd}\ cnd\ \overline{md}\} & \text{(Class Definition)} \\
fd & ::= & C\ f & \text{(Field Definition)} \\
cnd & ::= & C(\bar{C}\ \bar{x})\{\texttt{super}(\bar{e});\ vd;\ \bar{s}\} & \text{(Constructor)} \\
md & ::= & \tau\ m(\bar{C}\ \bar{x})\{vd;\ \bar{s}\ \texttt{return}\ x;\} & \text{(Method Definition)} \\
\tau & ::= & \texttt{void} \mid C & \text{(Return Types)} \\
vd & ::= & \bar{C}\ \bar{x} & \text{(Variable Declaration)} \\
s & ::= & ps & \text{(Statement)} \\
 & \mid & \texttt{if}(x)\ \{\bar{s}\}\ \texttt{else}\ \{\bar{s}\} & \\
 & \mid & \texttt{while}(x)\{\bar{s}\} & \\
ps & ::= & x = e;\ \mid x.f = e;\ \mid pe; & \text{(Primitive Statement)} \\
e & ::= & \texttt{null} \mid x \mid x.f \mid (C)\ x \mid pe & \text{(Expression)} \\
pe & ::= & x.m(\bar{x}) \mid \texttt{new}\ C(\bar{x}) & \text{(Promotable Expression)}
\end{array}
$$

Figure 3 Syntax of three-address MJ, where x ranges over MJ variables, f ranges over field names, C ranges over class names, and m ranges over method names.

2.2 Syntax

The syntax of PATL is formally presented in Figure 4. Similar to SmPL [27] and SWIN [18], a PATL program is a set of patch-like transformation rules. Each rule π in a PATL program consists of two parts: 1) metavariable declarations including type information of metavariables: a declaration $v : C_1 \hookrightarrow C_2$ means that the source type (type in old API) of the metavariable v is C_1 and the target type (type in new API) is C_2, and 2) a rule body consists of a series of program patterns. The I^- pattern block describes the statements to be deleted and the I^+ pattern block describes the statements to be generated. In a PATL rule π, all metavaribales used in I^- and I^+ are required to be declared in the metavariable declaration part of that rule, and particularly, when a metavariable u is newly-introduced in I^+ (u does not appear in I^-, meaning that u has no source type), u should be defined with a dummy old type $\perp$ in its declaration.

$$
\begin{array}{rcll}
\Pi & ::= & \pi_1, ..., \pi_n & \text{(Rule Sequence)} \\
\pi & ::= & (\bar{d})\ \{I^-; I^+\} & \text{(Transformation Rule)} \\
d & ::= & u : C \hookrightarrow C & \text{(Metavariable Declarations)} \\
I^- & ::= & -\ p_1, ..., -\ p_n & \text{(Source Pattern)} \\
I^+ & ::= & +\ p_1, ..., +\ p_n & \text{(Target Pattern)} \\
p & ::= & u = r;\ \mid r; & \text{(Statement Pattern)} \\
r & ::= & u.m(\bar{u}) \mid \texttt{new}\ C(\bar{u}) \mid u.f & \text{(Expression Pattern)}
\end{array}
$$

Figure 4 PATL syntax, where u ranges over PATL metavariables and C ranges over MJ types.

3 Basic Semantics

Basic semantics of PATL performs only strict match on consecutive blocks and syntactical transformation. In Section 5 we shall describe how to extend the basic semantics to deal with code blocks in different forms.

3.1 Match

Intuitively, statements $\bar{s}^-$ in method M are matched by a rule π defined in a rule set Π, if 1) $\bar{s}^-$ is a sequence of consecutive statements in M that can be matched by the source pattern $\pi.I^-$, and 2) all variable occurrences matched by a same metavariable should have a same name. The formal definition is presented below. We assume each location in a program is uniquely identified, and use the notation x^l in the rest of the paper to denote the occurrence of variable x at location l in the program.

▶ **Definition 1** (Match). Given a method M, statements $\bar{s}^-$ in M are said to form a *match instance* with a transformation rule $\pi = (\bar{d})\{I^- \ I^+\}$ if the following conditions are satisfied.

- (Block) A basic block $\bar{s}$ exists in M, s.t. $\bar{s}^-$ is a consecutive statement sequence in $\bar{s}$.
- (Source pattern match) $\bar{s}^-$ can be matched by the pattern block I^- syntactically. There exists a map ϕ from variable occurrences to metavariable names, s.t. by substituting each variable occurrence in $\bar{s}^-$ with its image in ϕ, $\bar{s}^-$ is exactly the same as I^-.
- (Variable mapping) Suppose x^l and $y^{l'}$ are two variable occurrences matched by the same metavariable u, then $x \equiv y$ (i.e. x and y are the same variable), and these two occurrences x^l, $y^{l'}$ are must aliases.
- (Variable typing) Suppose a variable occurrence x^l is matched by a metavariable u, then $\mathsf{type}(x) <: \mathsf{type}(u)$ if x^l is a right-value in M, and $\mathsf{type}(u) <: \mathsf{type}(x)$ if x^l is a left-value in M (Function $\mathsf{type}(x)$ refers to the type of x in M, and $\mathsf{type}(u)$ refers to the source type of metavariable u in π). ◀

We denote a *match instance* as a triple $b = (\bar{s}^-, \pi, \sigma)$, where $\sigma = \{u_1 \mapsto [x^{l_1}, ..., x^{l_n}], ...\}$ is the set of mappings from metavariables to variable occurrences formed in the match. An element $u_1 \mapsto [x^{l_1}, ..., x^{l_n}] \in \sigma$ indicates that the metavariable u matches to different occurrences $x^{l_1}, ..., x^{l_n}$ of the variable x in the program.

▶ **Definition 2.** A matching instance set $\mathcal{B}$ is a set of match instances $\{b_1, ..., b_n\}$ formed by matching a method M against a set of rules Π, such that for any $b_1 \neq b_2 \in \mathcal{B}$, where $b_1 = (\bar{s}_1^-, \pi_1, \sigma_1)$ and $b_2 = (\bar{s}_2^-, \pi_2, \sigma_2)$, the statements $\bar{s}_1^-$ and $\bar{s}_2^-$ should have no overlap.

This constraint is to ensure that one statement will not be deleted twice in transformation. In our system, a warning will be generated if two mapping instances overlap. In the following sections, we always refer $\mathcal{B}$ to a valid match instance set.

Note that in our pattern definition there is no syntax for capturing a field access expression. Here we simply treat field access expressions as variables and allow metavariables to match them.

Example. An example is presented in Figure 5: given the source program on left of Figure 5 and the transformation rules in Figure 1, the two statements in lines 2,3 will be matched by the rule `rButton`, and the statement in line 5 can be matched by the rule `rSetAlign`, and the match instance set is presented below: (Here, s^2 refers to the statement `btn=new Button();` in line 2, and `btn`^{2} refers to the occurrence of the variable `btn` in line 2, other notations are similar.)[c]

$$\mathcal{B} = \big\{(s^2s^3, \texttt{rButton}, \big\{\texttt{jb} \mapsto \{\texttt{btn}^2, \texttt{btn}^3\}, \texttt{parent} \mapsto \{\texttt{panel}^2\}\big\}), \\ (s^5, \texttt{rSetAlign}, \big\{\texttt{jb} \mapsto \{\texttt{btn}^5\}, \texttt{align} \mapsto \{\texttt{alX}^5\}\big\})\big\} \tag{1}$$

3.2 Transformation

▶ **Definition 3** (Target Pattern Instantiation)**.** Given a match instance set $\mathcal{B}$, suppose $b = (\bar{s}^-, \pi, \sigma) \in \mathcal{B}$, then $\pi.I^+$ will be instantiated into $\bar{s}^+$ by substituting each metavariable u in $\pi.I^+$ with an MJ variable:

- If u is a metavariable in $\pi.I^+$ and there exists $u \mapsto [x^{l_1}, ..., x^{l_n}] \in \sigma$, then u will be substituted with x.
- If u is a metavariable in $\pi.I^+$ and u is a metavariable not appearing in $\pi.I^-$, but defined in $\pi.I^+$, then u will be instantiated as a new variable name.

▶ **Definition 4** (Statement-level Transformation)**.** Given a method M (with its statement body $\bar{s}_M$) and a rule Π we denote the transformation of $\bar{s}_M$ by Π as $\Pi \triangleright \bar{s}_M$. Suppose $\mathcal{B} = \{b_1, ..., b_n\}$ is the set of match instances between M and Π where $b_i = (\bar{s}_i^-, \pi_i, \sigma_i)$, then $\Pi \triangleright M$ is the result of substituting all $\bar{s}_i^-$ with $\bar{s}_i^+$ (statements instantiated from $\pi_i.I^+$ with b_i). Formally, $\Pi \triangleright \bar{s}_M = [\bar{s}_1^- \mapsto \bar{s}_1^+, ..., \bar{s}_n^- \mapsto \bar{s}_n^+]\bar{s}_M$. ◀

Besides statement-level transformation, we will also 1) add the definitions of newly introduced variables in the variable declaration field of M (the type of a newly introduced variable is the target type of its corresponding metavariable defined in the rule) and 2) transform the types based on the type mapping information provided in metavariable declarations in each rules in Π. But as they are not our focus in this paper, we do not formalize these two transformations.

Example. Given the match instances in Equation 1 between the program in left Figure 5 and rules in Figure 1, target statements will first be generated before substitution. In the first match instance, the target statement is generated by substitution of metavariable `jb` with `button`, and substitution `parent` with `panel`. The resulting target statement is `btn=new Button(panel, SWT.PUSH);`. Similarly, by substituting `jb` with `btn` and substituting `align` with `alX`, we obtain the target statement `btn.setAlignment(alX);` from the second match instance. By substituting source statements in each match instance with its corresponding target statements, the desirable result can be obtained (right of Figure 5).

```
1 //Program
2 btn = new JButton();
3 panel.add(btn);
4 alX = 10;
5 btn.setAlignmentX(alX);
```

$$\xrightarrow[\texttt{rSetAlign}]{\texttt{rButton}}$$

```
1 //Transformed Program
2 btn = new Button(panel, SWT.PUSH);
3 alX = 10;
4 btn.setAlignment(alX);
```

Figure 5 A transformation example with rules defined in Figure 1, the statements in lines 2,3 (left) are transformed into the statement in line 2 in the result program (right) by rule `rButton`. Similarly, the statement in line 5 is transformed to the statement in line 4 (right) by rule `rSetAlign`. In our examples, variables are always declared in the previous context, but we omit them for concision consideration.

[c] If there exist two variables at the same line, the encoding of l will not just be the line number but both its line number and its character offset number, but as there is no such case in this example, we only use line number for convenience.

4 Preserving Def-Use Relations

As mentioned in Challenge 2 in the introduction, we would like to disallow rules that may change def-use relations. Thus, besides checking the syntax and type correctness of the rules as SWIN [18] did, an addition set of well-formedness conditions are checked against a transformaion program Π to ensure the it will perserve def-use relations in transformation.

▶ **Definition 5** (Well-Formedness Conditions)**.** A PATL program $\Pi = \pi_1, ..., \pi_n$ is well-formed if the following four conditions are satisfied.

- *(Definition deletion)* For all π, if there exists a pattern $-\ u = r$ in I^-, then there should also be a pattern $+\ u = r'$ in I^+.
- *(Definition introduction)* For all π, if there exists a pattern $+\ u = r$ in I^+, then either there exists $-\ u = r$ in I^- or u does not appear in I^-.
- *(New metavariable introduction)* Given a rule π, if there exists a pattern $+p$ in $\pi.I^+$ such that a metavariable u in p does not appear in $\pi.I^-$, then there exists a pattern $+u = r$ in $\pi.I^+$ that introduce the definition of u. ◀
- *(SSA pattern form)* For all π, the code block formed by I^- (and I^+) must be in static single assignment (SSA) form, i.e. each variable is assigned at most once.

The first and second conditions ensure the def-use relations of existing variables. The third condition ensures that the new variables are used after definition. The fourth condition makes the other three conditions simpler, and also contributes to guided normalization to be introduced in the next section. As an example, the BadRule in the introduction violates the first two conditions, and thus can potentially violate def-use relations.

Now we can formally define and prove the def-use preservation property of a checked PATL program Π. Suppose $\mathcal{B}$ is the match instance set between a method M and Π, $\Pi(s)$ is used to refer to the statements corresponding to s after transformation, defined as: (1) the statements generated from a binding b, if exists $b = (\bar{s}^-, \pi, \sigma) \in \mathcal{B}$ s.t. $s \in \bar{s}^-$, or (2) s itself, if s is not matched by any rule. We have the following theorem.

▶ **Theorem 6** (Def-use Preservation)**.** *Given a method M and a checked PATL program Π, suppose the statement-level transformation is $\Pi \triangleright \bar{s}_M = \bar{s}'_M$, then:*

1. *Suppose s_1 and s_n are two statements in a basic block in $\bar{s}_M$, and there exists a variable x that is used in s_n and defined in s_1, then after transformation, either $\Pi(s_n)$ does not use the variable x, or the occurrences of x used in $\Pi(s_n)$ is defined in $\Pi(s_1)$.*
2. *Suppose x is a variable newly introduced in $\bar{s}'_M$, then x is used after definition.*

Proof. (Proof sketch) For the first part of theorem, firstly, we need to prove that transforming a statement sequence $\bar{s}_1$ in M will not delete a variable definition statement. Suppose a statement "`x=e;`" is removed by a match instance b, then a pattern "`-u=e';`" must exists in b. According to the condition *Definition Pattern*, a target pattern "`+u=e''`" also exists in b and it will be instantiated into a statement which defines the variable `x`. Secondly, we need to show that the transformation will not introduce a variable definition `y=e;`, where no definition statement of `y` appears in $\bar{s}_1$ and `y` is defined and used in a context before $\bar{s}_1$. This can be proved by the following facts: 1) all newly introduced variables will have fresh names, according to the Definition 3 and 2) these exists no statement pattern that will write to a read-only variable in $\bar{s}_1$, according to the checking condition *Definition Introduction*. With the above two properties, the first part of the property can be proved inductively.

For the second part, if `x` is a newly introduced variable, then `x` is matched by a metavariable `u` newly introduced in I^+. As required in the checking condition *New Variable Introduction*, a metavariable newly introduced in I^+ should also be defined in I^+. This definition statement for `u` in I^+ will then be instantiated as the definition statement for the variable `x` in the transformed program, according to Definition 3. ◀

5 Extending Basic Semantics via Guided Normalization

As mentioned in Challenge 1 in the introduction, we use guided normalization to extend the basic semantics to different cases. We first start with the introduction of program analysis techniques used in our approach, then introduce the extended match that identifies different cases, and show how to guided-normalize the matched statements into the basic form.

5.1 Program Analysis

In order to semantics-preservingly transform a program p, the following two program analysis results are required.

1. Alias relations between variable occurrences in p: given two variable occurrences x^{l_1} and y^{l_2}, identify whether x^{l_1} and y^{l_2} are *none-aliases*, *may-aliases* or *must-aliases*[d].
2. Dependency relations between statements in p: given two primitive statements s_1 and s_2, whether there are no dependencies between them or there may exist dependencies between them, considering both data dependency and control dependency.

As the transformation algorithm only requires the analysis results, the analysis algorithm is orthogonal to the transformation phase and any *conservative* program analysis tool providing these results can be used. And as we will show later, the precision of analysis result will not affect the semantics-preserving property of the transformation process due to our conservative treatment of the analysis result, but less precise analysis result may increase the number of warnings reported by the transformation algorithm that require manual resolutions.

5.2 Extending Match

Now we define the extended match, considering match instances formed by potentially scattered consecutive statements and rules.

▶ **Definition 7** (Match*). Given a method M, statements $\bar{s}^-$ in M are said to form a *match instance* with a transformation rule $\pi = (\bar{d})\{I^- \ I^+\}$ if the following conditions are satisfied.

- (Path*) There exists a statement sequence $\bar{s}$, which forms an execution path in the control flow graph of M, s.t. $\bar{s}^-$ is a (potentially scattered consecutive) sub-sequence in $\bar{s}$.
- (Source pattern match) $\bar{s}^-$ can be matched by pattern I^- syntactically.
- (Variable mapping*) Suppose x^l and $y^{l'}$ are two variable occurrences matched by the same metavariable u, then x^l and $y^{l'}$ are may-aliases in M.
- (Variable typing) Suppose a variable occurrence x^l is matched by a metavariable u, then $\mathsf{type}(x) <: \mathsf{type}(u)$ if x^l is a right value in M, and $\mathsf{type}(u) <: \mathsf{type}(x)$ if x^l is a left value in M. ◀

Different from Definition 1, in Definition 7: (1) statements are only required to appear in a path in the control flow graph, and (2) variable occurrences bound to the same metavariable are only required to be may-aliases. Here, we use $\mathcal{B}^* = \{b_1^*, ..., b_n^*\}$, where $b_i^* = (\bar{s}_i^-, \pi_i, \sigma_i^*)$,

[d] Typical analysis tools will only provide none-alias relations and may-aliases relations, but enhancing them with a conservative intra-procedural aliases analysis can further provides must-alias relations between variable occurrences.

to refer to the set of match instances formed between a method M and a PATL program Π with extended match definition Match^*. The difference between σ^* and σ is that given $u \mapsto [x_1^{l_1}, ..., x_n^{l_n}] \in \sigma^*$ (indicating metavariable u maps to the variable occurrences $x_1^{l_1}, ..., x_n^{l_n}$ in the match instance), occurrences $x_1^{l_1}, ..., x_n^{l_n}$ are only required to be aliases but not necessarily with the same name.

Match Finding and Checking. Given a method M and a set of rules Π, we use a dataflow analysis to obtain all match instances between statements in M and rules in Π. The behavior of the dataflow analysis is similar to the method presented by Brunel et al. [7]. Due to space limit, we omit the details here.

As mentioned before, not all match instances can be guided-normalized. If there exist match instances that cannot be handled by guided-normalization, we will report them to users as warnings. Some untransformable match instances can be easily identified by static condition check. More will be identified during the process of guided normalization. Basically, we identify any match instance $b^* = (\bar{s}_1, \pi, \sigma^*)$ satisfying one of the following three conditions and report it as a warning.

- Any two variable occurrences matched by the same metavariable in σ^* are not must-aliases.
- Statements $\bar{s}_1$ appear across methods.
- Statements $\bar{s}_1$ appear across boundary of a `while` statement, i.e., the matched statements are in different iterations, or, some are inside a loop while some outside.

The latter two conditions are checked after the matching process, while the first condition is checked after the guided-shift step in guided normalization (introduced later in this section). This is because the guided-shift step will eliminate some may-aliases, and after checking, all uncertain aliases between variables occurrences involved in match instances are resolved. On the other hand, other uncertain aliases relations will not need to be handled as they will not affect transformation correctness, as a result, only a small part of uncertainties is required to resolve to proceed transformation.

Example. An example program demonstrating extended match definition with rules `rButton` and `rSetAlign` is presented below on the left of Figure 6. In this program, firstly, the alignment field of the button is set via a call to "`btn.setAlignmentX(alX);`". Then, before adding the new `JButton` object to `panel`, it checks whether `panel` is `null`: if it is not null, then the `btn` is added to `panel`, otherwise the button is assigned to `defaultBtn`, added to a default panel `defaultPnl`.

From pointer analysis we can obtain that all occurrences of `btn` and `defaultBtn` are must-aliases (and thus are may-aliases). Thus, three match instances (in Match*) can be obtained between the program and the rules in Figure 1: i.e. 1) statements in lines 1,6 can form a match instance with the rule `rButton`, 2) statements in lines 1,9 can form another match instance with the rule `rButton` and 3) statement in line 2 can be matched by the rule `rSetAlign`. And these match instances can be represented as Equation 2:

$$\begin{aligned}\mathcal{B}^* = \big\{&(s^1 s^6,\ \texttt{rButton},\ \big\{\texttt{jb} \mapsto \{\texttt{btn}^1, \texttt{btn}^6\}, \texttt{parent} \mapsto \{\texttt{panel}^6\}\big\}),\\ &(s^1 s^9,\ \texttt{rButton},\ \big\{\texttt{jb} \mapsto \{\texttt{btn}^1, \texttt{defaultBtn}^9\}, \texttt{parent} \mapsto \{\texttt{defaultPnl}^9\}\big\}),\\ &(s^2,\ \texttt{rSetAlign},\ \big\{\texttt{jb} \mapsto \{\texttt{btn}^2\}, \texttt{align} \mapsto \{\texttt{alX}^2\}\big\})\big\}\end{aligned} \qquad (2)$$

5.3 Guided Normalization

Guided normalization transforms the program in a semantics-preserving manner such that the match instance in the extended semantics can be transformed by the basic semantics. We first demonstrate the result of guided normalization by example.

Example. A desirable guided-normalization for the program with the match instances in Equation 2 is presented below (6 right), as after normalization, statements matched to a same rule appear consecutively in basic block and variable occurrences matched by a same metavariable have same name. It is obvious that after guided-normalization, the transformation defined in Section 3 can be performed as all extended match instances become the basic in-block matches. Please note in this program different API methods do not write or read to the same field, so there is no dependency between these method calls. This information can be obtained by a dependency analysis.

```
 1 btn = new JButton();
 2 btn.setAlignmentX(alX);
 3 System.out.print(alX);
 4 b = panel != null;
 5 if (b) {
 6   panel.add(btn);
 7 } else {
 8   defaultBtn = btn;
 9   defaultPnl.add(defaultBtn);
10 }
```

$\xrightarrow{normalize}$

```
 1 System.out.print(alX);
 2 b = panel != null;
 3 if (b) {
 4   btn = new JButton();
 5   panel.add(btn);
 6   btn.setAlignmentX(alX);
 7 } else {
 8   x = new JButton();
 9   defaultPnl.add(x);
10   btn = x;
11   defaultBtn = btn;
12   btn.setAlignmentX(alX);
13 }
```

Figure 6 An Example Program to be Transformed by Rules `rButton` and `rSetAlign` and its guided normalized result. The program on the right is normalized with match instances in Equation 2, and after guided normalization, the match instances become: 1) statements in lines 4,5 form a match with rule `rButton`, 2) statement in line 6 form a match with rule `rSetAlign`, 3) statements in lines 8,9 form a match with rule `rButton` and 4) statement in line 12 form a match with rule `rSetAlign`.

Semantics Preserving Transformation. As mentioned in Section 1, the key point of guided-normalization is to ensure that the normalization process is semantics-preserving. Thus before moving to an algorithmic description of guided-normalization, we shall first define which transformations are semantics-preserving.

▶ **Definition 8** (Semantics-preserving transformation). A method M (the body of M is $\bar{s}_M$) is said to be semantics-equivalently transformed into M', if M can be transformed into M' with a series of the transformation primitives defined below. We denote such semantics-preserving transformation as $M \xrightarrow{\sim} M'$.

- *Alias Renaming Primitive:* In $\bar{s}_M$, if x^l is a must alias of variable y in the statement of l, renaming x to y at the statement of l is semantics-preserving.
- *Left-value Renaming Primitive:* In $\bar{s}_M$, suppose $x = e;$ is a statement defining the value of x, and y is a free name in M, then after declaring y in M, substituting $x = e;$ with $y = e; x = y;$ is semantics preserving.
- *Fresh-variable Introduction Primitive:* In $\bar{s}_M$, suppose x is a fresh variable name in M, then declaring x in M and inserting $x = y;$ at a location where y is defined is semantics-preserving.
- *Swapping Primitive:* In $\bar{s}_M$, suppose $s_1 s_2$ are two adjacent statements with no dependency, then transforming them into $s_2 s_1$ is semantics preserving.
- *Shifting Primitive:* In $\bar{s}_M$, (1) given $s_1\texttt{if}(x)\{\bar{s}_2\}\texttt{else}\{\bar{s}_3\}$ and suppose x does not depends on s_1, then substituting it with $\texttt{if}(x)\{s_1\bar{s}_2\}\texttt{else}\{s_1\bar{s}_3\}$ is semantics preserving. (2) given $\texttt{if}(x)\{\bar{s}_2\}\texttt{else}\{\bar{s}_3\}s_1$, substituting it with $\texttt{if}(x)\{\bar{s}_2 s_1\}\texttt{else}\{\bar{s}_3 s_1\}$ is semantics preserving. ◀

The transformation primitives defined above are verified and commonly used in compiler optimization [37, 1, 8, 14] and semantics-preserving refactoring [33, 26, 31], and as long as we can show that a transformation algorithm can be decomposed into such series of transformation primitives, the transformation process is guaranteed to be semantics-preserving.

Transformation Stages. Algorithmically, the guide-normalization process can be decomposed into the following three stages:

1. Stage-1: Transforming the program with GuidedShift Algorithm so that statements matched by a rule will appear in a basic block in the resulting program.
2. Stage-2: Transforming the program with GuidedRename Algorithm, and variable occurrences matched a same metavariable in a rule will have same name and definition.
3. Stage-3: Transforming statements in basic blocks with GuidedReorder Algorithm, so that statements matched by a same rule will appear consecutively in the block.

In the rest of the section, we will concretely describe each guided normalization stage and prove its semantics-preserving property. The guided normalization process consists of three transformation stages (three algorithms), i.e. GuidedShift, GuidedRename and GuidedReorder.

In our algorithms, we use Δ to denote a *must-alias* checker, which is obtained by program analysis, i.e. given two variable name x, y and a location l, $\Delta(x, y, l) = \texttt{true}$ indicates that x and y are must-aliases at the location l otherwise not. We also use Θ to represent a dependency checker, i.e. given two statements, $\Theta(s_1, s_2) = \texttt{true}$ indicates that s_1 and s_2 may have dependencies otherwise not [e]. Particularly, besides dependencies obtained from analysis, here we also consider *match dependency*: two statements s_1 and s_2 are said to have match dependency when they are in a same match instance in $\mathcal{B}^*$. We shall refer match dependency and data dependency uniformly as dependency.

5.3.1 GuidedShift Algorithm

In the first stage, we make all statement sequences in a match instance appear in the same basic block, i.e. given the match instance set $\mathcal{B}^* = \{(\bar{s}_1^-, \sigma_1^*, \pi_1), ...\}$, if $\bar{s}'_M = \mathsf{GuidedShift}(\bar{s}_M, ST_{\mathcal{B}^*}, \Theta)$, then all $\bar{s}_i^-$ in match instances from $\mathcal{B}^*$ will appear in a basic block in $\bar{s}'_M$. Here $ST_{\mathcal{B}^*}$ is the shifting targets of $\mathcal{B}^*$, indicating which statements are supposed to appear in a basic block. $ST_{\mathcal{B}^*}$ is calculated by including all adjacent statement pairs in $\bar{s}_i^-$, formally, given $b^* = (\bar{s}_i^-, \sigma_i^*, \pi_i) \in \mathcal{B}^*$, if $s_1 s_2$ are two adjacent statements in $\bar{s}_i^-$, then a pair (s_1, s_2) is added into $ST_{\mathcal{B}^*}$.

The function locateBlock(s) is used to find the basic block where s is in, ShiftDownInto (ShiftUpInto) is used to move a statement into the beginning (end) of both branches of an `if` statement without moving any other statement, and UpdateLocation is used to update locations for all statements whenever a shift operation happens (as it changes locations of statements).

In the algorithm, given a target pair (s_a, s_b), we will first decide find two compound statements s_1, s_2 such that s_1, s_2 appear in the same basic block and s_1 contains or equals s_a, s_2 contains or equals s_b (line 2). Then we determine whether s_1 is a compound statement containing s_a. If so, we shift s_2 into an inner level of the block s_1 (lines 3-13). In the shifting process, we visit each statement after s_1 in the block s_1 is in, and when a statement s' having dependence with s_2 is found, it will be shifted first to avoid dependency breaking (lines 8-10). If no such statement exists, s_2 will moved into an inner level of the block s_1 using the

[e] If may-dependence relation is reported between two statements s_1 and s_2, the result will be treated as "s_1 and s_2 have dependencies" in our approach. This treatment ensures that any *possible* dependencies will not be broken in transformation.

Algorithm 1: GuidedShift algorithm

```
Input: statements $\bar{s}$, shifting targets $ST = [(s_a, s_b), ...]$,
       dependency checker $\Theta$.
Output: shifted statement sequence $\bar{s}'$
1  while exists tuple $(s_a, s_b) \in ST$, s.t. $s_a$ and $s_b$ are not in the same block do
2      find $s_1, s_2$, s.t. $s_1$ is the statement containing $s_a$, $s_2$ is the statement containing $s_b$ and $s_1, s_2$
       are in the same basic block.
3      if $s_1 \neq s_a$ and $s_1$ is a compound statement containing $s_a$ then
4          $\bar{s}_t \leftarrow$ locateBlock($s_1$);
5          $i_1 \leftarrow$ indexOf($s_1, \bar{s}_t$);
6          for $i \leftarrow i_1 + 1, ..., $ size$(\bar{s}_t) - 1$, do
7              $s_x \leftarrow \bar{s}_t[i]$
8              if $s_x == s_2$ then
9                  shiftUpInto($s_2, s_1$); updateLocations($\bar{s}$);
10                 break;
11             if $\Theta(s_2, s_x) ==$ true then
12                 GuidedShift($\bar{s}_t, [(s_x, s_1)]$);updateLocations($\bar{s}$)
13                 break;
14     else if $s_1 = s_a$ and $s_2 = $ if$(u)\{s_{21}\}$else$\{s_{22}\}$ then
15         $\bar{s}_t \leftarrow$ locateBlock($s_a$);
16         $i_2 \leftarrow$ indexOf($s_2, \bar{s}_t$);
17         for $i \leftarrow i_2 - 1, ..., 0$ do
18             $s_x \leftarrow \bar{s}_t[i]$
19             if $s_x == s_a$ then
20                 if $\Theta(u, s_a) ==$ false then
21                     shiftDownInto($s_a, s_2$); updateLocations($\bar{s}$);
22                     break;
23                 else
24                     report(); retract(); exit();
25             if $\Theta(s_x, s_1) ==$ true then
26                 $\bar{s}_t \leftarrow$ GuidedShift($\bar{s}_t, [(s_x, s_2)]$);
27                 updateLocations($\bar{s}$)
28                 break;
```

shiftUpInto function (line 8-10). When $s_1 = s_a$, meaning that s_a is in an outer block level compared to s_b, we need to shift s_a into s_2 unless it is s_a and s_b are already in the same basic block (lines 14-28). This process is similar to the former part except that dependence relation will be checked when we try to shift a statement into an if statement, if dependence between the condition variable and the statement exists, an warning will be generated for user to handle (line 24).

The algorithm will always terminate as the block-level (the number of nested blocks a statement is in) of some statement will increase in each loop. And the algorithm only terminates when the shifting goal is satisfied. The semantics-preserving property of the algorithm is presented below.

▶ **Property 9** (GuidedShift semantics-preserving). *Let M be a method ($\bar{s}_M$ be its body), Θ be a dependency checker that contains all statement dependencies in $\bar{s}_M$, and ST be a set of shifting targets. We have $M \xrightarrow{\sim} [\bar{s}_M \mapsto \bar{s}'_M]M$ if the invocation to the algorithm* GuidedShift$(\bar{s}_M, ST, \Theta) = \bar{s}'_M$ *finishes without warning.*

Proof. They key point is to show that whenever we call the function ShiftDownInto(s, s_1) in line 21 (or ShiftUpInto(s, s_1) in line 9), where s_1 is an if statement, all statements between s and s_1 have no dependency with s, so that we can decompose this operation into a series of primitive swaps and a primitive shift operation. And this is ensured as we will recursively shift statement depends on s into s_1 before shifting s (lines 11-13, 25-28 in the algorithm). ◀

Example. Given the example in Figure 6, the first stage transformation is shifting, and it is performed as follows in Figure 7.

```
1 btn = new JButton();
2 btn.setAlignmentX(alX);
3 System.out.print(alX);
4 b = panel != null;
5 if (b) {
6   panel.add(btn);
7 } else {
8   defaultBtn = btn;
9   defaultPnl.add(defaultBtn);
10 }
```

$\xrightarrow{\text{GuidedShift}}$

```
1 System.out.print(alX);
2 b = panel != null;
3 if (b) {
4   btn = new JButton();
5   btn.setAlignmentX(alX);
6   panel.add(btn);
7 } else {
8   btn = new JButton();
9   btn.setAlignmentX(alX);
10  defaultBtn = btn;
11  defaultPnl.add(defaultBtn);
12 }
```

Figure 7 Guided-shift result of the example in Figure 6 left. Based on the match instances defined in Equation 2, the statement in line 1 is the target statement to be shifted into branches. Besides, the statement in line 2 is also shifted into branches as it depends on the statement in line 1.

5.3.2 GuidedRename Algorithm

The second stage of guided-normalization is to deal with variable names in $\bar{s}_M$, so that if there exist two variable occurrences x^{l_1}, y^{l_2} matched to the same metavariable u in a match instance (x^{l_1}, y^{l_2} are aliases to form such match), we will rename them into a uniform name via semantics-preserving transformation. Since a variable may be re-assigned or used outside a matching instance, to avoid disturbance to the code outside a matching instance, we always introduce a new variable and renaming to the new variable.

Similar to the previous normalization stage, the renaming targets $RT_{\mathcal{B}^*}$ are calculated first based on the $\mathcal{B}^*$. Given a match instance $b^* = (\bar{s}^-, \pi, \sigma^*) \in \mathcal{B}^*$ and $u \mapsto [x_1^{l_1}, ..., x_n^{l_n}] \in \sigma^*$, we add $[x_1^{l_1}, ..., x_n^{l_n}]$ into the renaming targets, as these variable occurrences are matched by the same metavariable u in b^*. When we obtain $RT_{\mathcal{B}^*}$ from $\mathcal{B}^*$, we will run GuidedRename on $RT_{\mathcal{B}^*}$ and M to obtain the desirable normalized program.

In our algorithm, we use the following auxiliary functions: 1) Occurs(RT) calculates the set of all variable occurrences appearing in RT, 2) FreshName(M) generates a fresh variable name that is not used in M, 3) UpdateAlias() updates variable locations in both M and RT after transformation and 4) rename(x, y, l) renames x into name y at the location l.

In GuidedRename algorithm, we will deal with left-value renaming in lines 1-9 and deal with right-value renaming in lines 10-21.

Left-value renaming (line 1-9) is used to deal with situations that the first element $x_1^{l_1}$ in a renaming target $rt = [x_1^{l_1}, ..., x_n^{l_n}]$ is a left-value in M. (As we can prove, the only possible left-value in a rt is the first element, as the patterns are required to be in SSA form and all $x_i^{l_i}$ are matched by a same metavariable u in a rule.) Firstly, we will find the statement that defines the value of $x_1^{l_1}$ and collect all such statements in a set S_1 (line 1). And then, for each collected assignment statement, we will generate a fresh name to rename it through the *Left-value renaming primitive* defined in Definition 8 (lines 2-5). Then, as all these $x_i^{l_i}$ appear in a same basic block (as a result of the GuidedShift algorithm), we will rename all variable occurrences in rt into the new name (lines 7-8). Similarly, Right-value renaming (lines 10-23) helps renaming a target such that all variable occurrences are right values.

Example. The guided-renaming phase for the program in Figure 7 is shown below in Figure 8. As in the match instance between statements in lines 8-11 and the rule `rButton` involves an alias-pair (i.e. `btn`^{8} and `defaultBtn`^{11}), they will be renamed to have the same name. The renaming process is done by introducing a new variable name `x` following the left-value renaming part defined in the GuidedRename algorithm (lines 8,9,12 in the right of Figure 8).

Algorithm 2: GuidedRename algorithm

```
   Input: method $M = \tau\ m(\bar{C}\ \bar{x})\{vd;\ \bar{s}_M;\ \texttt{return}\ x;\}$,
          renaming targets $RT = \{[x_{11}^{l_{11}}, ..., x_{1n_1}^{l_{1n_1}}], ...\}$
   Output: method $M$ with some variables in the body renamed
1  $S_1 \leftarrow$ Assignment statements with left-value in Occurs($RT$);
2  foreach $s_d \in S_1$ ($s_d$ of form $y^l = e;$) do
3      $x_1 \leftarrow$ FreshName($M$);
4      $M.vd \leftarrow M.vd \cup \{\text{type}(y^l)\ x_1\}$;
5      $M.\bar{s}_M \leftarrow [s_d \mapsto x_1 = e; y = x_1;]M.\bar{s}_M$;
6      UpdateAlias();
7      foreach $y^{l'}$ where $y^{l'}$ and $y^l$ are in a $rt \in RT$ and $l \neq l'$ do
8          rename($y, x_1, l'$); UpdateAlias();
9  $S_2 \leftarrow \emptyset$;
10 foreach $rt = [x_{i1}^{l_{i1}}, ..., x_{in_i}^{l_{in_i}}] \in RT$ do
11     if Foreach $x_{ij}^{l_{ij}} \in rt$, $x_{ij}^{l_{ij}}$ is a right-value then
12         $S_2 \leftarrow S_2 \cup \{$statement use $x_{i1}^{l_{i1}}\}$;
13 foreach $s \in S_2$ do
14     foreach variable $y^l$ used in $s$ do
15         if $y^l \in$ Occurs($RT$) and $y^l$ is the first element of a $rt \in RT$ then
16             $x_1 \leftarrow$ FreshName($M$);
17             $M.vd \leftarrow M.vd \cup \{\text{type}(y^l)\ x_1\}$;
18             $M.\bar{s}_M \leftarrow [s \mapsto (u = y; [y \mapsto x_1]s;)]M.\bar{s}_M$;
19             UpdateAlias();
20             foreach other $y^{l'}$ in $rt$ do
21                 rename($y, x_1, l'$); UpdateAlias();
22 return $M$;
```

Similar to GuidedShift algorithm, GuidedRename is also semantics preserving, and we present the proof below.

▶ **Property 10** (GuidedRename Semantics-preserving)**.** *Let M be a method, Π be a transformation program, $\mathcal{B}^*$ be a set of match instances between M and Π, and $RT_{\mathcal{B}^*}$ be a set of renaming targets generated from $\mathcal{B}^*$. We have $M \xrightarrow{\sim} M'$ if* GuidedRename$(M, RT_{\mathcal{B}^*}) = M'$.

Proof. The places we will modify M are lines 3-5, line 9, lines 17-19 and line 23 in the algorithm. Firstly, the operation in lines 3-5 directly corresponds to a left-value renaming primitive, lines 17-19 directly corresponds to a free variable introduction primitive and an alias renaming primitive so that they are semantics preserving. The key point here is to show that modification in line 9 (similarly, in line 23) is also semantics preserving by showing that x_1 and y are aliases at l' (if they are aliases, then we are transforming with alias renaming primitive). This can be proved by the transitivity of alias relations: $x_1^{l'}$ is an alias to x in s_d, x in s_d is an alias to y^l, and y^l is an alias to $y^{l'}$ (as they are in target set generated from $\mathcal{B}^*$), so that x_1 and y are aliases at location l'. ◀

5.3.3 GuidedReorder Algorithm

The last phase of guided normalization is *guided-reordering*, in which we want to reorder statements in blocks so that given a match instance $b^* = (\bar{s}^-, \pi, \sigma^*) \in \mathcal{B}^*$, statements $\bar{s}^-$, will appear consecutively.

The reordering targets OT required by the algorithm will first be built: starting from an empty set, for each $b_i^* = (\bar{s}_i^-, \pi_i, \sigma_i^*) \in \mathcal{B}^*$, $\bar{s}_i^-$ will be added into OT, i.e. the goal is to make all such statement sequences matched by source patterns appeared consecutively. The dependency checker Θ here is same as the checker used in GuidedShift.

```
1  System.out.print(alX);
2  b = panel != null;
3  if (b) {
4    btn = new JButton();
5    btn.setAlignmentX(alX);
6    panel.add(btn);
7  } else {
8    btn = new JButton();
9    btn.setAlignmentX(alX);
10   defaultBtn = btn;
11   defaultPnl.add(defaultBtn);
12 }
```

$\xrightarrow{\text{GuidedRename}}$

```
1  System.out.print(alX);
2  b = panel != null;
3  if (b) {
4    btn = new JButton();
5    btn.setAlignmentX(alX);
6    panel.add(btn);
7  } else {
8    x = new JButton();
9    btn = x;
10   btn.setAlignmentX(alX);
11   defaultBtn = btn;
12   defaultPnl.add(x);
13 }
```

Figure 8 Guided-rename result following the previous result in Figure 7, a new variable name `x` is introduced to rename `btn`^{8} and `defaultBtn`^{11}.

In the GuidedReorder algorithm, firstly, for each block $\bar{s}$, we assign each statement $s \in \bar{s}$ with a field l indicating its target location. Then, we encode the goal as constraints using by these locations, and then these locations can be calculated by solving the constraints. Concretely, the constraints are built in the following way:

- For each $s_i, s_j \in \bar{s}$, if $\Theta(s_i, s_j) = \texttt{true}$, add $s_i.l < s_j.l$ into the constraint set if $i < j$, otherwise add $s_j.l < s_i.l$ if $j < i$ (line 6-10). These constraints ensure that the statement dependencies are kept after reordering.
- For each statement sequence $\bar{s}_k \in OT$, add $s_{k(i)}.l + 1 = s_{k(i+1)}$ into the constraint set (line 11-13). These constraints ensure that target statements will appear consecutively in the right order.

As these constraints form a *system of difference constraints* [10], and we can solve it through shortest path algorithm. When we successfully solve the constraints, we will then reorder the statements accordingly in M to obtain the desirable result. If there exists no solution to the constraints, warnings will be generated to users, as we cannot make all matched statements appear consecutively due to dependency issues.

Algorithm 3: Statements Reordering

```
   Input: method $M$, dependency checker $\Theta$,
          reordering targets $OT = [\bar{s}_a, ...]$
   Output: Re-ordered statement sequence $\bar{s}'$
1  foreach basic block $\bar{s}$ in $M$ do
2      $OT_s \leftarrow$ {statement sequences from $OT$ and in $\bar{s}$};
3      (Suppose $OT_s$ is $\{\bar{s}_1, ..., \bar{s}_m\}$)
4      $n \leftarrow \mathsf{length}(\bar{s})$;
5      $Constraints \leftarrow \emptyset$;
6      foreach $s_i \neq s_j \in \bar{s}$ do
7          if $\Theta(s_i, s_j)$ and $i < j$ then
8              $Constraint \leftarrow Constraint \cup \{s_i.l + 1 \leq s_j.l\}$;
9          else if $\Theta(s_i, s_j)$ and $j < i$ then
10             $Constraint \leftarrow Constraint \cup \{s_j.l + 1 \leq s_i.l\}$;
11     foreach $\bar{s}_k \in OT$ do
12         foreach $i = 1$ to $\mathsf{length}(\bar{s}_k) - 1$ do
13             $Constraint \leftarrow Constraint \cup \{s_i.l + 1 = s_{i+1}.l\}$;
14     if TrySolve($Constraints$) successful then
15         $\bar{s}' \leftarrow$ Sort $\bar{s}$ according to $s.l$;
16         $M \leftarrow [\bar{s} \mapsto \bar{s}']M$;
17     else
18         report();
19 return $M$;
```

Example. After performing GuidedRename the program in Figure 8, we only need to reorder the statements in lines 4,6 and statements in lines 8,12 in program (right of Figure 8) to obtain the desired program in Figure 6 left. At this point, we successfully guided-normalize the program as presented in 6.

The following property shows that GuidedReorder is semantics-preserving.

▶ **Property 11** (GuidedReorder semantics-preserving)**.** *Let M be a method, Θ be a dependency checker containing all statement dependencies in M, and OT be a set of reordering target. We have $M \overset{\sim}{\longrightarrow} M'$ if $M' = \mathsf{GuidedReorder}(M, \Theta, OT)$.*

Proof. We prove the property by showing that for each block $\bar{s}$, if $\bar{s}$ is reordered into $\bar{s}'$, then $\bar{s}$ can be transformed into $\bar{s}'$ through a series of *Swapping Primitives* defined in Definition 8.

We first prove that the dependencies in the original program are preserved in $\bar{s}'$, i.e. if $\Theta(s_i, s_j) == \texttt{true}$ and s_i appear before s_j in M, then $s_i.l < s_j.l$: suppose s_i and s_j have dependency in $\bar{s}$ and $j < i$, then $s_j.l < s_i.l$ is added into the constraint set. By solving the constraints, we still have s_j appears before s_i after transformation in $\bar{s}'$, $s_j.l < s_i.l$. Thus we have all dependencies preserved in $\bar{s}'$.

Then we present a constructive method on how to generate a sequence of *Swapping Primitives* to transform $\bar{s}$ into $\bar{s}'$: 1) label the statements in $\bar{s}'$ with $[1, ..., n]$, and assign that label to statements in $\bar{s}$ (e.g. if s_1 in $\bar{s}'$ is labeled as k, then the s_1 in $\bar{s}$ also have label k, as a result, labels in $\bar{s}'$ are sort while not in $\bar{s}$), 2) perform a bubble sort algorithm on labels in $\bar{s}$ and record swaps when running the algorithm. We now show that all these swaps are *Swapping Primitives*: as indicated in the last part, if s_i depends on s_j, then s_j appears before s_i in both $\bar{s}$ and $\bar{s}'$, then before performing bubble sort on the labels, the label of s_i and s_j is already sorted. As bubble sort algorithm will never swapping sorted pairs, any of the swap operations used are *Swapping Primitives*. Thus, $M \overset{\sim}{\longrightarrow} M'$ and the property is proved. ◀

5.4 Main Theorem

Here we present the main theorem on the semantics-preservation of guided normalization.

▶ **Theorem 12** (Main Theorem)**.** *Let M be a method, Π be a rule set, and $\mathcal{B}^*$ be a set of match instances between statements in M and rules in Π. We have $M \overset{\sim}{\longrightarrow} M'$ if the guided normalization of M with $\mathcal{B}^*$ returns M'. In other words, the transformation is semantics-preserving.*

Proof. This theorem is a simple derivation of Property 9, 10 and 11. ◀

6 Full Language Implementation

Based on the core PATL presented in previous sections, we implemented a version of PATL for Java (Patl4J). On top of core PATL, there are several extensions in the full language.

First, the full language supports standard Java programs that are not necessarily in three-address code. Before matching and transformation, we first convert the Java program into three-address code. This is achieved by introducing a set of temporary variables to decompose statements that are not in three-address form. We give special names to the temporary variables, and after the transformation, we try to apply the "inline variable" refactoring to inline these variables to recover the original program. In this way we can ensure the structure of the original program is retained to some degree.

Second, the full language supports context-sensitive matching, similar to the context-sensitive matching in SmPL [27] and TXL [9]. We introduce a new type of pattern to match a context of a rule. A context is a sequence of statements (not necessarily consecutive) that always appear before the matched statements in all paths in the control flow graph, and transformation only when its context is matched by the rule. An example of the context-sensitive matching can be found in the evaluation section.

Finally, the full language provides a new type of transformation rules to change method definitions. In Java, we may define a class that extends a library class or implements a library interface. When the library class/interface is mapped to a new class/interface, the client definition should also be changed. Our rule works similarly to refactorings [13], allowing to rename a method, reorder the parameters of a method, or introduce a new parameter.

Currently, the full language is implemented as an Eclipse Plug-in using the Eclipse JDT parser to manipulate the syntax tree and obtain type and use Soot [35] to perform program analysis for the client program. Concretely, in our implementation, desired program analysis results are obtained using SOOT Spark pointer analysis tool, where 1) the alias relation between two variable occurrences is determined by querying the anaylsis tool whether two variable occurrences always points to the same memory location and 2) the dependence relations between two statements is determined by querying the tool whether there exists variable occurrences in the two statements accessing the same memory location. Our prototype can be found in its web site[f] [g].

7 Evaluation

How effective is PATL for transforming real-world industrial cases? To answer this question, we evaluate our approach on three groups of widely-used Java API cases, i.e. Jdom to Dom4j, Google Calendar version 2 to version 3, and Swing to SWT.

7.1 Data Set

In our experiments, we chose three case studies that migrate programs from Jdom[h] to Dom4j[i], from Google Calendar[j] version 2 to version 3, and from Swing[k] to SWT[l], respectively. Jdom and Dom4j are two popular XML parsers, but Dom4j has better performance over Jdom on a number of tasks, so it is desirable to migrate programs from Jdom to Dom4j. Google Calendar API is a web service provided by Google to access personal calendar data. The interface for version 2 has been shut down, and version 3 is not compatible with version 2 clients, so the clients will not work unless migrated to version 3. Swing and SWT are two Java GUI libraries. Swing uses platform-independent components while SWT is designed as a light-weight wrapper of native GUI. SWT is sometimes considered faster than Swing, and some platforms such as Eclipse only supports SWT. We chose the three case studies because they are real-world program migration cases, and a large number of clients are available for the evaluation. Also, the three cases cover the two main types of program migration: API switching and API upgrading.

[f] https://github.com/Mestway/Patl4J.
[g] As a proof-of-concept, the type analysis, interprocedural dependency analysis, and some transformation steps in our implementation is not fully automated and may require user input. Nevertheless, this is purely a pragmatic problem due to our limited resource on implementation; our approach can be implemented fully automatically.
[h] http://www.jdom.org/
[i] http://www.dom4j.org/
[j] https://developers.google.com/google-apps/calendar/
[k] http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
[l] http://www.eclipse.org/swt/

Table 1 Subject Client Programs

Client	KLOC	Classes	Methods	Case
husacct	195.6	1187	5977	Jdom/Dom4j
serenoa	12.2	52	523	Jdom/Dom4j
openfuxml	112.5	727	4098	Jdom/Dom4j
clinicaweb	3.9	74	213	Calendar
blasd	9.7	199	729	Calendar
goofs	8.6	78	643	Calendar
evochamber	12.8	132	868	Swing/SWT
swingheat	2.3	30	186	Swing/SWT
marble	1.6	10	56	Swing/SWT
Total	359.2	2489	13293	–

To evaluate the transformation program we wrote, we also collected a number of client programs that use the source API. These clients are obtained by searching the source API methods in searchcode[m]. In total, we used nine client programs in our evaluation. The statistics of the subject programs can be found in Table 1. In total, the projects totally contain 342.5 KLOC, including 2317 classes and 12183 methods, details about these projects can be found in our implementation website.

7.2 Procedures

For each case, we first wrote PATL rules that capture the correspondence between the old and the new APIs. Since the changed portion of the API is large, and it is important to test the rule under a real client, we only dealt with the portion of API that is used in our subject client programs. However, the rules we wrote are generic rules for any possible client, not specific to the subjects in our evaluation.

Next we transformed client programs that contain source API invocations using our transformation tools. To produce working clients, we manually resolved warnings reported during the transformation. A few API invocations are impossible to be transformed due to the limitation of PATL, and we create mock objects for them.

To ensure that the transformation is performed correctly, we performed three different tests for these converted clients. (1) Client EvolutionChamber comes with a set of 28 functional and 4 performance tests, and we ensure that the transformed client passes all transformed tests. (2) We check whether they behave normally without exceptions, error messages, or crashes.. (3) For the clients in JDom/Dom4j and Swing/SWT, we side-by-side executed both the original and transformed clients to ensure they behaved the same. Note that we cannot apply the last test to the clients of Google Calendar because Calendar API v2 is already shut down.

7.3 Results

Rules Written. The statistics for the rules and transformed source APIs are summarized in Table 2: the ‘Rule’ column shows the number of the rules. The ‘Class’ and the ‘Method’ columns contian the number of API classes and methods covered by the rules. In total, we wrote 236 rules for the three case studies, and 66 classes and 204 methods in the source API are covered.

[m] http://searchcode.com

Table 2 Transformation Rules

Transformation	Rules	Classes	Methods	M-to-m
Jdom/Dom4j	84	12	77	12(14.3%)
Calendar	42	14	45	21(50.0%)
Swing/SWT	110	40	82	54(49.1%)
Total	236	66	204	87(36.9%)

Table 3 Results of the Transformations

Client	CF	CL	W	U	I	MM	GN
husacct	42	852(100%)	0(0%)	0(0%)	0	0(0%)	0(0%)
serenoa	8	273(98.9%)	0(0%)	3(1.1%)	0	9(3.3%)	0(0%)
openfuxml	72	983(94.8%)	0(0%)	54(5.2%)	15	2(0.2%)	0(0%)
clinicaweb	5	81(100%)	0(0%)	0(0%)	8	34(42%)	0(0%)
blasd	5	26(63.4%)	8(19.5%)	7(17.1%)	0	13(50%)	2(15.4%)
goofs	13	100(80.0%)	12(9.6%)	13(10.4%)	27	27(27%)	0(0%)
evochamber	9	587(98.3%)	10(1.7%)	0(0%)	0	330(56.2%)	109(33.0%)
swingheat	21	653(100%)	0(0%)	0(0%)	0	461(70.6%)	394(85.5%)
marble	6	488(98.6%)	0(0%)	7(1.4%)	0	240(49.2%)	220(91.7%)
Total	181	4043(97.3%)	30(0.7%)	84(2.0%)	50	1116(27.6%)	725(65.0%)

CF = number of changed files, CL= number of changed lines, percentages in CL = CL / (CL+W+U), W = the number of lines of code that have warnings, percentages in W = W / (CL+W+U), U = number of lines that PATL cannot transform, percentages in U = U / (CL+W+U), I = number of lines impossible to transform, MM = number of lines that are involved in many-to-many mappings, percentages in MM = MM / CL, GN = number of lines that require guided normalization, percentages in GN = GN / MM.

Basically, the number of rules is close to the number of covered methods. Furthermore, most rules are easy to write: only 14 rules in total that have a body longer than 4 lines. This indicates that PATL rules are friendly to users: they only need to capture basic forms in writing the rules without worrying about complex program context.

Effectiveness of Client Transformation. To evaluate the effectiveness of our approach on transforming clients, we counted the following numbers in our evaluation: (1) the number of files transformed by the rules (CF), (2) the number of lines transformed by the rules (CL), (3) the number of lines in the source client code on which warnings are generated (W), (4) the number of lines where transformed code exists but the transformation rule cannot be specified in PATL (U). Details of the data are presented in the first five columns in Table 3.

As we can see from the table, the 236 rules changed in total 4043 lines distributed in 181 files. This result reflects the effort saved from manual migration. The efforts saved are twofold: (1) to locate these lines from different files, (2) to derive transformation solution for each line based on the context surrounding this line.

Furthermore, the number of lines transformed by PATL consist of 97.3% of the lines that need to be converted. The rest of lines need manual resolution: the lines where warnings are reported and the unconvertible lines. These lines amount to 114 lines, consisting of 2.7% of total lines. This result indicates that PATL is able to handle the majority of the cases in practice.

Many-to-Many Rules. A significant portion of the transformation rules are many-to-many rules. As we can see from the last column of Table 2, in total 36.9% of the rules are many-to-many rules, and in the case of Google Calendar upgrade the percentage is as high as 50%. These many-to-many rules is responsible to transform a significant portion of the client code. As shown in column "MM" in Table 3, in total 27.6% of the lines are transformed by many-to-many rules, and in project swingheat the percentage is as high as 70.6%.

From our running example we have seen a typical case where many-to-many rules can

```
// Rule-1                                  //Rule-2
(x: CalendarEntry->CalendarListEntry,      (y: TextConstruct -> String){
 y: TextConstruct->String,                     - y = new PlainTextConstruct(z);
 z : String -> String){                        + y = "";
    m y = new PlainTextConstruct(z);       }
    - x.setTitle(y);
    + x.setSummary(z);
}
```

Figure 9 Two transformation rules used in transforming Google Calendar v2 to v3.

be applied. Here we show another typical case: a wrapper class in the source API does not exist in the target API. For example, in the old version of Google Calendar, a wrapper class called `PlainTextConstruct` is used to wrap a string. For example, client goofs contains the following two lines of code (`e` in the statement is a variable of type `CalendarEventEntry`).

```
e.setTitle(new PlainTextConstruct(name));
```

In the new API, the use of the wrapper class is removed as many as possible. As a result, the above two lines should be converted to the following line of code.

```
e.setSummary(name);
```

This instance is a typical many-to-many mapping instance, and the transformation can be captured by the two rules in Figure 9.

The first rule uses context-sensitive matching described in Section 6. The statement annotated with "m" is a context pattern, which indicates that `x.setTitle(y)` is only matched and transformed if there is `y = new PlainTextConstruct(z)` before it. The first rule captures the wrapping of a string and the use of the wrapper, and converts it into a proper method invocation based on how the wrapper is used. The second rule removes the useless variable `y`. Note that the target method `setSummary` is decided by both the argument passed to `PlainTextConstruct` (in this case, a string but not an html object) and the invoked source method on `x` (in this case `setTitle`), so this rule has to be many-to-many.

Guided Normalization. The number and percentage of lines in the source clients that require guided normalization are shown in the last column of Table 3. As we can see from the table, a significant portion of lines involved in many-to-many rules, i.e., 65.0% of lines, requires guided normalization to correctly perform the transformation. This result indicates (1) guided normalization is necessary; (2) guided normalization is only performed on necessary lines but not all lines, avoiding excessive change to the layout of the source code.

Besides guided normalization, a conversion to three-address code is performed at the beginning of the transformation, and the introduced temporary variables will be inlined at the end of the transformation. In our experiment, all temporary variables are inlined at the end. This result indicates that the conversion to three-address code would not significantly affect the source code, either.

In particular, most lines requiring guided normalization exist in the clients of the Swing/SWT. This is because a lot of UI elements are first initialized and then added to their parents, such as the following example, in which we need to first swap the statements at line 2 and line 3 and then the rule in our running example becomes application.

```
button = new JButton();
button.setText("OK");
panel.add(button);
```

Unspecifiable Cases. As shown by the column "U" in Table 3, there are in total 84 lines of code whose transformation patterns cannot be captured by PATL rules. These unsupported mapping patterns are summarized into the following three categories.

The first category is that type mapping does not form a function. For example, a class may split into two classes, each inherits part of the functionalities of the original class. Since in the meta variable declaration we only allow to map one class to another class, we cannot write a rule for such cases. Most lines, 74 out of 84 belong to this category.

The second category is that the transformation needs high-level coordination between match instances, and only one such instance of three lines is observed. This instance is from Jdom/Dom4j. Below is the source client code (left of Figure 10), which converts elements in `rulesElements` into a string in `rulesToRegister`. An ideal transformation of this case is shown on right: instead of directly converting each element into string, they are written into a `StringWriter` to get the concatenated string. While we can write rules to generate the first three lines of the target code, We cannot generate the call to `sw.toString()` at the end of the target client code because there is no corresponding source statement. Generating this type of "closing" statement remains future work.

```
XMLOutputter out = new XMLOutputter();
for (Element e : rulesElements)
  rulesToRegister += out.outputString(e);
```

```
StringWriter sw = new StringWriter();
XMLWriter out = new XMLWriter(sw);
for (Element e : rulesElements)
  out.write(e);
rulesToRegister += sw.toString();
```

Figure 10 An example using the JDom that PATL cannot handle.

The last category relates to the use of `JPopupMenu` in Swing. In the evaluation 7 out of the 84 lines belong to this category. In Swing, to show and hide a pop menu, we need to implement a listener in which we write code to show and hide the pop menu. In SWT, we only need to call a method `setMenu`. To perform this transformation, we need to match both a method definition as well as the method body, which is not supported in PATL.

Warnings. As shown in the "W" column in Table 3, there are in total 30 lines of code on which warnings are generated. There are several reasons why warnings are generated. The first one, also the largest category, is that some code pieces cannot be transformed or shifted together because of dependencies among statements. 18 out of 30 lines belong to this category. The second one is that one line is matched by multiple "-" block in different rule instances. 2 out of 30 lines belong to this category. These two categories actually show the usefulness of guided normalization: unsafe transformations are disabled by guided normalization.

The last category is that the matched code is scattered in several methods. Since we do not move statements across method boundaries, our approach reports warnings on these lines. 10 out of 30 lines belong to this category. This shows a limitation of our current approach.

Untransformable code. As shown in the "I" column in Table 3, there are in total 50 lines of source code that are impossible to transform. This is because we cannot find any counterpart in the target API for a portion of the source API, and any lines invoking this portion cannot be transformed. The portion of untransformable API in the Google Calendar case is confirmed by the official Google Calendar upgrade guide, which states some functionalities are removed and will not be supported any more.

8 Related Work

Pattern finding in programs. Several works have been done in enhancing the pattern finding technique to enable finding program patterns with simpler rules [28, 7]. Among them, the most related is the approach proposed by Brunel et al. [7], which enables users to specify a sequence of consecutive statements to match different form statements against patterns with same semantics via model checking. The match part of our transformation technique is similar to this matching mechanism, while differently, besides matching, our language also performs transformation using these consecutive statement patterns.

Transformation languages. A lot of different program transformation approaches have been proposed for different purposes. Some of these approaches [24, 18, 3, 36, 21, 20] are specialized for handling one-to-many mappings: one method invocation is replaced by a sequence of invocations. Among them the most related are Twinning [24] and SWIN [18], both of which are designed for handling the API migration problem with one-to-many mapping rules. Our basic semantics and the type mapping design are similar to them. Another related approach is update calculus [21, 20], which ensures the type-safe update of programs. Our approach currently does not ensure type safety, and potentially can be combined with update calculus and SWIN to ensure type safety. Overall, compared with these languages designed for one-to-many mappings, our approach supports many-to-many mappings, the importance of which is recognized by several empirical studies [4, 34] as well as our evaluation.

Besides, a number of program transformation languages or frameworks can be used to handle program transformation involved in many-to-many mappings.

SmPL [27] is a transformation language designed to document and automate the collateral evolutions of large C programs with patch rules. Our syntax for specifying the three patterns are inherited from semantic patches. SmPL also includes a state check [7] to check transformations that may potentially break data dependency relations, and a failure will be report on Case-2 in introduction (which our approach successfully handles). Stratego [6] is a general purpose transformation language, allowing users to define rules to rewrite the abstract syntax tree of a program. Stratego handles the context-sensitive transformation with the support of dynamic rules, enabling users rewrite rule at places where context information is currently unavailable. Dataflow facts in Stratego can also be collected and propagated using dynamic rules [25]. TXL [9] is another general purpose transformation language manipulating the abstract syntax trees, which allows acquiring the context information in tree traversal. Lacey and De Moor [17] propose a graph rewriting language where conditions on execution paths can be specified using temporary logic. Crossver [32] proposed a way to combine dataflow analysis with aspect-oriented programming to transform programs.

Overall, though these languages have provided more general and expressive operations for matching and transforming programs, they are low-level and require more effort to create correct and generic transformation. Differently, our language focuses on a more restrict but simpler language interface for the API transformation tasks.

Automated Transformation for APIs. Several approaches try to further reduce the cost of program adaptation between APIs by automatically discovering the transformation program. Typical approach includes recording the API refactorings and replay them on the client code [13, 12], and analyzing manually adapted code pieces [2, 23, 22] or clients [39]. Currently, none of these approaches supports the automatic transformation of many-to-many mappings in a flexible and safe manner. Our approach can be potentially combined with those approaches to reduce the effort of writing code.

Normalization. Normalization techniques are commonly used in compiler optimization[15, 1, 14, 37, 8], semantics-preserving refactoring [26, 31] and program extraction [16].

Komondoor and Horowitz [16] presented a normalization technique that helps to move a set of identified statements to form a consecutive sequence, for program extraction purpose. However, due to the different domain requirements, their approach [16] handles only the situation that the normalization target is one sequence of statements, while our approach is required to normalize multiple sequences at the same time. As a result, we restrict the transformation primitives to retain the semantics-preserving property with the more complex normalization goal. Furthermore, transforming programs between APIs also requires performing renaming between aliases, which is not supported by Komondoor and Horowitz's approach [16] as it is not a requirement in code extraction.

Code motion [15] is an optimization technique which, when applied to if-branches, behaves similarly to our shift operation. However, these operations are not designed for user-specified program transformations. In addition, different from their approach and other compiler optimization approaches [1, 14, 37, 8], normalization in our apporach are guided by match instances so that most original program structure can be kept after normalization.

9 Conclusion

In this paper we have presented PATL for safe transformation of complex programs between different APIs with simple Many-to-Many mapping rules. The key insight for PATL is to bridge the gap between simple in-block transformation semantics and complex program structures using guided normalization, which automatically changes the program so that code in different forms can be transformed in a unified manner. We applied PATL to three real world program transformation cases. The evaluation showed that PATL is expressive in handling real world scenarios, and with the help of guided normalization, only a small amount of manual resolution is required.

References

1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley, 2006.

2 Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *ASE*, pages 382–385, 2012.

3 Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA*, pages 265–279, 2005.

4 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an api migration for two xml apis. In *SLE*, pages 42–61, 2010.

5 G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003.

6 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008.

7 Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

8 David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *CC*, pages 152–161, 1986.

9 James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.

10 Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

11 Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006.

12 Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. *ReBA*: *re*factoring-aware *b*inary *a*daptation of evolving libraries. In *ICSE*, pages 441–450, 2008.

13 Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE*, pages 274–283, 2005.

14 Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM*, pages 108–117, 2006.

15 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *PLDI*, pages 224–234, 1992.

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

17 David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *CC*, pages 52–68, 2001.

18 Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. Swin: Towards type-safe java program adaptation between apis. In *PEPM*, pages 91–102, 2015.

19 Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service api evolution affect clients? In *ICWS*, pages 300–307, 2013.

20 Deling Ren Martin Erwig. Type-safe update programming. In *ESOP*, pages 269–283, 2003.

21 Deling Ren Martin Erwig. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67:199–222, 2007.

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

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

24 Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *ICSE*, pages 205–214, 2010.

25 Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *CC*, pages 204–220, 2005.

26 William F. Opdyke. *Refactoring Object-oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

27 Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys*, pages 247–260, 2008.

28 Nicolas Palix, Jean-Rémy Falleri, and Julia Lawall. Improving pattern tracking with a language-aware tree differencing algorithm. In *SANER*, pages 43–52, 2015.

29 Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

30 Mark Pilgrim. *Dive Into Python 3.* Apress, Berkely, CA, USA, 2009.

31 Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

32 Kouhei Sakurai and Hidehiko Masuhara. Crossver: a code transformation language for crosscutting changes. In *AOAsia/Pacific Workshop*, 2014.

33 Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for java. In *OOPSLA*, pages 277–294, 2008.

34 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *ICSM*, 2010.

35 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, 1999.

36 Louis Wasserman. Scalable, example-based refactorings with refaster. In *WRT*, pages 25–28, 2013.

37 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.

38 Daniel Weise and Roger Crew. Programmable syntax macros. In *PLDI*, pages 156–165, 1993.

39 Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *ICSE*, pages 195–204, 2010.