Search-Based Generalization and Refinement of Code Templates

Tim Molderez and Coen De Roover

Software Languages Lab, Vrije Universiteit Brussel, Belgium
tmoldere@vub.ac.be, cderoove@vub.ac.be

Abstract. Several tools support code templates as a means to specify searches within a program's source code. Despite their ubiquity, code templates can often prove difficult to specify, and may produce too many or too few match results. In this paper, we present a search-based approach to support developers in specifying templates. This approach uses a suite of mutation operators to recommend changes to a given template, such that it matches with a desired set of code snippets. We evaluate our approach on the problem of inferring a code template that matches all instances of a design pattern, given one instance as a starting template.

Keywords: Templates · Evolutionary Algorithms · Recommender Systems

1 Introduction

In program search and transformation tools, source code templates are a means to concisely describe source code snippets of interest. For example, templates can describe all instances of a particular bug, snippets that need to be refactored or transformed, instances of design patterns, ... However, code templates can still prove difficult to specify: when a user has little experience working with templates, or needs to write a larger or more complex template, the templates may not always produce the desired results. A template could be too general and produce too many matching snippets. It could also be too specific and produce too few matches. In this paper, we introduce a search-based [10] approach and a suite of mutation operators to assist users of Ekeko/X [5], a template-based search and transformation tool for Java. [1]

Automated generalization and refinement - When a template produces too few or too many matches, the Ekeko/X user can mark which ones are either undesired or missed, and invoke our search-based approach. It automatically looks for a sequence of mutations to the template, so it does produce *only* the desired matches. This approach uses a single-objective evolutionary algorithm (EA). To evaluate this EA, we perform an experiment in the context of generalizing design pattern instances: the EA is given one instance of a design pattern, and is then tasked to find a template to match all instances of that pattern.

[1] The Ekeko/X program transformation tool and the extensions presented in this paper are available at https://github.com/cderoove/damp.ekeko.snippets

Mutation operators - A key component of the EA is its suite of mutation operators, which determine the different types of modifications that the EA can perform on a template. Important to note is that these mutation operators can also be used directly by the Ekeko/X user to edit templates. This provides two benefits: first, mutation operators can only be applied if they lead to a syntactically valid template, which prevents syntax errors. Second, some of the operators automate common scenarios such as abstracting away the name of a particular variable declaration and its uses.

In summary, after giving a brief overview of the Ekeko/X tool in Sec. 2, this paper presents the following contributions: Sec. 3 provides a suite of all template mutation operators. Sec. 4 presents our search-based approach to automatically generalize and refine code templates. Finally, Sec. 5 discusses the experiment to evaluate whether the approach is able to automatically find a suitable solution.

2 The Ekeko/X Program Transformation Tool

2.1 Overview

Ekeko/X is a program search and transformation tool for Java, where searches and transformations are specified in terms of code templates. A code template is a snippet of Java code, in which parts (corresponding to AST nodes) can be replaced by wildcards and metavariables, and different annotations called *directives* can be added. These constructs are used to either add or remove constraints to/from parts of a template. The process of *matching* a template involves looking for all concrete snippets of Java code that satisfy all constraints specified in that template. A simple example of a template is the following:

```
public class ... {[public void toString(){...}]@[match|set]}
```

It describes any public class that defines a `toString` method. To abstract away the class name and the `toString` method body, wildcards (shown as "...") are used. A `match|set` directive is also attached to the `toString` method; it indicates there may be other class members beside the `toString` method. If the directive were absent, the template would describe classes that *only* define `toString`. In general, attaching one or more directives to a piece of code uses the following notation: `[`*`code`*`]@[`*`directives`*`]`.

Ekeko/X also provides support for template *groups*, in which multiple templates can be related to each other. An example of such a template group is given in Fig. 1. This example can be used to check the code convention that fields should not be accessed directly if a getter method is available. Any matches produced by this template group indicate a violation against the code convention. An example match is shown in Fig. 2. The group consists of two templates: the first (lines 1-5) describes a class with a field and its getter method; the second (lines 7-8) describes a method containing a reference to that field.

Aside from wildcards and directives, this example also makes use of *metavariables* (shown as an identifier starting with a "?"). These are logic variables whose values are concrete snippets of code. Directives can also refer to metavariables.

```
1  public class ... {[
2     [private ... ?field;     // Field
3      public ... ... () {     // Getter method
4          return [...]@[(refers-to ?field)];}
5  ]@[match|set]}
6
7  public ... ... (...) {
8     [...]@[(refers-to ?field) child*]}
```

Fig. 1: Any direct field reference for which a getter is available

In line 4, the `refers-to` directive has `?field` as its operand. Because the directive has an operand, it is contained in parentheses. The directive specifies that the variable in the return statement must directly refer to the value of `?field`. This ensures that the method in line 3-4 is the getter method of field `?field`.

```
public class Square extends Shape {
    private int length = 5;
    public Square(int length) {this.length = length;}
    public int getLength() {return length;}}

public int area(Square s) {return s.length*s.length;}
```

Fig. 2: One of the matches of the template group

The second template describes a method declaration, also using the `refers-to` directive to specify that there should be a reference to `?field` in the method's body. It also has a second directive, `child*`. This indicates that the reference to `?field` may occur anywhere in the method body (at any nesting depth). Without this directive, the method body's would consist *only* of the field reference.

2.2 Definitions

To define our suite of different mutation operators, we should first make some of the core concepts related to templates more precise:

Template - A template is a snippet of code, where parts can be replaced by wildcards or metavariables, and parts can be annotated with directives. To make this more precise, it is more convenient to define a template as a tree structure rather than a piece of text. In particular, a template is a decorated abstract syntax tree (AST), where every node is decorated with a set of directives.

We will refer to these decorated AST nodes as *template nodes*, or simply nodes. When referring to AST nodes that are part of the program being searched, we will call these *source nodes*.

Template group - A template group is a set of templates. Relations between templates in a group can be established as well: if a metavariable occurs in multiple templates, these occurrences all refer to the same metavariable. In the example of Fig. 1, the `?field` metavariable is used to link both templates.

Metavariable - A metavariable is a variable (in a logic programming context) of which the value is a source node.

Directive - A directive attaches additional constraints to a node. These constraints will be taken into account whenever the template is matched. A

directive is always attached to one node in a template, which is referred to as the *subject*. A directive can also have operands, where most directives use metavariables as operand values.

Matching - A template group produces a match if a mapping is found between template nodes and source nodes, such that all of the template group's constraints are satisfied.

Note that, while this is not visible in the textual representation of a template, all nodes except the root implicitly have a directive (typically the `child` directive), which adds the constraint that this node should be a child of its parent. This is necessary to reflect the template's tree structure in the list of constraints.

Matching node - During matching, when a mapping is found between a template node and a source node, that source node is called the *matching node*. For example, there is a mapping between the wildcard template node in line 1 of Fig. 1 and `Square`, the corresponding matching node in Fig. 2.

3 Mutation Operator Suite

An operator, or "mutation operator" in full, performs a modification in a template group. An operator is always applied to one node, also referred to as the operator's subject. There are two types of operators: *atomic* and *composite* operators. Atomic operators only modify a single node in a template; composite operators may modify multiple nodes in multiple templates of a group.

3.1 Atomic Operators

An overview of all available atomic operators is given in Table 1, listing each operator's name and its operands, which subjects it can be applied to, and a brief description. We will then highlight a selection of operators in more detail:

Replace by variable (var) - The subject and its children are replaced by a metavariable node. Any directives present in the subject are preserved, except `match`. Additionally, a directive is added that will bind the matching node to the given metavariable (var). In the following example, the operator is applied to the "Hello world" string, such that the resulting template matches any `println` call, and metavariable `?arg` is bound to the call's actual argument:

```
System.out.println("Hello world");
```

⇒ *Subject "Hello world", Operands* ⟨*?arg*⟩

```
System.out.println(?arg);
```

Add directive (dir, operands) - This operator attaches the given directive, with the given operand values, to the subject node. As there are several directives available, shown in Table 2, we only highlight a selection:

• `child / child+ / child*` - This directive relates the subject to its parent node x. In case of `child`, x's matching node is the parent of the subject's matching node. For `child*`, x's matching node is a direct or indirect ancestor of the

Table 1: Overview of atomic operators related to program search

Operator	Subject	Description
Replace by variable (?var)	Any non-root, non-protected	Replaces the subject with a metavariable.
Replace by wildcard	Any non-root, non-protected	Replaces the subject with a wildcard.
Add directive (dir , operands)	Depends on selected directive	Adds a directive to the subject, with the given operand values.
Remove directive (dir)	Any	Removes a given directive from the subject.
Remove node	Non-mandatory child of parent, non-protected	Removes the subject node.
Insert node at (type, index)	List	Inserts a new node of the given type into the subject list, at the given index.
Replace node (type)	Non-primitive, non-root and non-protected	Replaces the subject by a new node of the given type.
Replace value (value)	Primitive, non-protected	Replaces the subject by the given value.
Replace parent statement	Statement in body of another Statement	Statement in which the subject occurs is replaced by the subject.
Erase list	List	Removes all list elements of the subject.

Table 2: Overview of the available matching directives

Directive signature	Subject	Description
`child,child+,child*`	Any	Relates the subject node to its parent template node x. The matching node of x is the parent (`child`) / indirect ancestor (`child+`) / ancestor (`child*`) of the subject's matching node.
`(equals ?var)`	Any	The subject now unifies with the given metavariable.
`match`	Any	Checks that the subject node type and its properties correspond to the matching node's.
`match|set`	List	The list elements of the subject must also appear (in any order) in the matching node's list elements.
`(type ?type), (type|sname <str>), (type|qname <str>)`	Type, variable declaration/reference or expression	The matching node should resolve to or declare the given type. (specified as a metavariable, its simple name or its qualified name)
`(subtype+/* ?type), (subtype|sname+/* <str>), (subtype|qname+/* <str>),`	Type, variable declaration/reference or expression	The matching node should resolve to or declare a (reflexive) transitive subtype of the given type.
`(refers-to ?var)`	Identifier in method body	Matching node lexically refers to a local variable, parameter or field denoted by the argument.
`(referred-by ?expr)`	Field/var. decl. or formal method parameter	Matching node declares a local variable, parameter or field lexically referred to by `?expr`.
`(invokes ?method), (invokes|qname <string>)`	Method call	Matching node is an invocation to the given method, considering the receiver's static type.
`(invoked-by ?call)`	Method declaration	Inverse of the above: matching node is a method declaration that was invoked by `?inv`.
`(constructed-by ?ctor)`	Constructor	Matching node is a constructor that was invoked by `?ctor` instantiation.
`(constructs ?ctor)`	Instantiation expression	Matching node is an instantiation that invokes the constructor `?ctor.`
`(overrides ?methdecl)`	Method declaration	Matching node is a method declaration that overrides the `?methdecl` declaration.
`protect`	Any	Prevents operators from removing or abstracting away this node.

subject's matching node. For `child+`, it is an indirect ancestor. Exactly one of these three directives must be present in every template node (except the root).

• `(invoked-by ?call)` - This directive adds a constraint that relates a method call to a method declaration. Consider that the subject is a method declaration in class x. This method declaration should be invoked by `?call`, a method call where the receiver's static type is x.

• `protect` - "Protects" the subject and all of its parents. If a node is protected, it cannot be accidentally removed or abstracted away, because any operators that could do so are now disallowed. This means the `protect` directive only affects the subject applicability of other operators, and does not add any constraints.

3.2 Composite Operators

Table 3: Overview of all composite operators

Operator	Subject applicability	Description
Isolate statement in block	Statement, cannot have protected ancestor	Parent is replaced by any block in which the subject statement occurs as a descendant.
Isolate stmt/expr in method	Statement/Expression, cannot have protected ancestor	Method body in which the subject occurs is replaced by any method body in which the subject occurs as a descendant.
Generalize references	Local var., field decl. or formal parameter	Abstract away the name of a variable, both in the declaration and all lexical references to it.
Generalize types (qname)	Type, non-protected	Abstracts away all occurrences of a particular type (while preserving its qualified name).
Extract template	Any non-root, non-primitive	Extracts the subject into a new, additional template in the template group.
Generalize invocations	Method/ctor. decl.	Abstracts away all invocations to the subject.

The list of available composite operators is given in Table 3. A selection of these operators is highlighted in more detail:

Isolate statement in method - The method body in which the subject occurs is replaced with "any method body that contains the subject". This is useful in cases where we are only interested in one particular statement of a method. This composite operator repeatedly applies the "Replace parent statement"-operator, until the statement appears directly in the method body. All other statements are removed from the body, and a `match|set` is added. Finally, a `child*` is added to the subject. This example isolates the `insertPointAt` call such that any `splitSegment` method containing this call will match:

```
public int splitSegment(int x, int y) {
    int i = findSegment(x, y);
    if (i != -1){insertPointAt(new Point(x, y), i+1);}
    return i+1;}
```

⇒ *Subject insertPointAt(new Point(x,y), i+1);*

```
public int splitSegment(int x, int y) {
    [[insertPointAt(new Point(x, y), i+1);]@[child*]]@[match|set]}
```

Generalize types - This operator abstracts away the name of a particular type, while preserving the information that all occurrences of that type still have the same type. This is done by replacing each occurrence of the type by

a wildcard, and attaching a `type` directive to it with the given metavariable. In this example all instances of type `Expression` have been abstracted away:

```
public class ... extends Statement {
    private ASTId<Expression> ...;
    public ASTId<Expression> getExpression() {...}
    public void setExpression(ASTId<Expression> e) {...}}
```

$\Rightarrow$ *Subject Expression , Operands* $\langle ?etype \rangle$

```
public class ... extends Statement {
    private ASTId<[...]@[(type ?etype)]> ...;
    public ASTId<[...]@[(type ?etype)]> getExpression() {...}
    public void setExpression(ASTId<[...]@[(type ?etype)]> e) {...}}
```

4 Recommending Template Mutations

After providing an overview of our suite of mutation operators, this section introduces our search-based approach, which uses an EA to automatically generalize or refine a template until it matches only with a desired set of snippets.

4.1 Evolutionary Algorithm

The idea is that the user first creates a rough draft of the desired template group, which may produce too few or too many matches. The user then marks which results were missed and/or which matches are undesired. Next, the EA is invoked, which continually modifies the template group with the aim of improving its match results. This continues until either a solution is found that matches exactly the desired set of source nodes, or the user interrupts the search process and uses the best template groups produced up to now.

Our motivation for choosing a search-based approach is three-fold: first, it is a relatively simple solution to a complex problem. Second, even if the approach does not find a solution that produces the desired matches exactly, it can still recommend a template group that is an improvement over the initial group. Third and finally, using this approach the suite of operators and directives of Sec. 3 can be extended without altering the EA.

The EA we are using in particular is single-objective. The individuals in the EA are represented directly as template groups. Pseudocode of the EA is presented as the *evolve* function in Fig. 3. This function takes a set of template groups (*init_templates*) and a set of desired source nodes (*d_matches*) as input. The *cur_gen* variable contains the current generation of template groups. Initially, it contains the input template group(s). Every iteration of the EA's while loop produces a new generation of template groups based on the previous one, until one of the groups has a *fitness* of 1, which indicates we found a solution that produces only the set of desired matches. The fitness function, which computes fitness values, is described in more detail in the next section.

Creating a new generation is done only by a process of selections and mutations. The *selections* set is created by performing tournament selection S times in the current generation, where S is user-chosen. Tournament selection chooses

```
evolve(init_templates, d_matches) {
  cur_gen := init_templates
  history := init_templates
  while(∄t ∈ cur_gen : fitness(t, d_matches) = 1) {
    selections := ⋃_{i=1}^{S} tourn_select(cur_gen, d_matches, R)
    mutants := ⋃_{i=1}^{M} ∃t : t = mutate(tourn_select(cur_gen, d_matches, R))
      and fitness(t,d_matches) ≠ 0 and t ∉ history
    cur_gen := selections ∪ mutants
    history := history ∪ mutants}}
```

Fig. 3: Pseudocode describing the evolutionary algorithm

one template group by randomly picking R (user-chosen) groups from the current generation, and returning the one with the best fitness out of those R.

A *mutants* set is also created: M (user-chosen) template groups are chosen via tournament selection, followed by applying a mutation operator to each group. This is done by first randomly choosing a subject node in one of templates of a template group. Next, a mutation operator is chosen at random from the operators presented in Sec. 3, followed by randomly choosing operand values. Most operators use metavariables as operands. To find operand values, a metavariable is chosen that already occurs in the template group, or a new one is generated.

Once a mutation is applied, it becomes part of the next generation on two conditions: first, it cannot have a fitness value of zero. This typically indicates that the mutant does not produce any matches whatsoever, and is highly unlikely to lead the search process in the right direction. Second, the new generation cannot contain mutants that were already seen in earlier generations, which is checked using the *history* set. The new generation is then created by concatenating the *selections* and *mutants* sets, and the EA can either move on to the next generation, or stop if a solution is found.

4.2 Fitness Function

We make use of a single-objective EA; there is a single fitness value that it aims to optimize. In our case, the fitness value is a real number in the [0,1] range, where higher is better. The fitness function, which computes the fitness of a template group, is defined in Fig. 4. It is given a template group t and a set of desired matches m as input. It is defined in terms of the F_1 score and the *partial score*, where each component is given a user-specified weight (W_1 and W_2).

$$fitness(t,m) = W_1.F_1(t,m) + W_2.partial(t,m) \quad , \text{ where } W_1 + W_2 = 1 \text{ and } W_1 \geq 0$$

$$partial(t,m) = (\sum_{i=1}^{n} \frac{matchCount(t,m_i)}{nodeCount(t)})/n$$

$$F_1(t,m) = \frac{prec(t,m).rec(t,m)}{prec(t,m) + rec(t,m)} \quad prec(t,m) = \frac{tp(t,m)}{tp(t,m) + fp(t,m)} \quad rec(t,m) = \frac{tp(t,m)}{tp(t,m) + fn(t,m)}$$

Fig. 4: Computing the fitness of a template group t

The main component of the fitness value is the F_1 score, a number in the [0,1] range defined in terms of how many desired (true positives, tp) and undesired (false positives, fp) matches were found by a template group, as well as how many desired matches were not found (false negatives, fn). The closer it is to 1,

the closer the template group is to producing *only* the desired matches. If false positives are found, the score lowers, which prevents the EA from producing solution template groups that simply match with anything.

While the F_1 score in itself is sufficient to recognize a solution template group, it also is a rather coarse-grained measure. It often takes a sequence of several mutations before a template group's F_1 score increases. For example, several wildcards may need to be introduced to produce an additional match. To make the fitness function more fine-grained, a second component is necessary, the partial score. The idea is that a template group that *almost* produces an additional desired match is better than one that does not. We want to measure how "close" a template group is to matching with each of the desired matches: for each of the desired matches, the template group is applied only against this desired match. Every node that is successfully mapped is one step closer to the template group actually producing that desired match. The ratio of mapped nodes (*matchCount*) to the total number of nodes in the template group (*nodeCount*) indicates how close the template group is to finding this desired match. The average of these ratios (one per desired match) is the partial score.

4.3 Reducing the Search Space

An important factor to consider in search problems is the size of the search space. To reduce it, we have taken several design decisions:

The first is related to the fact that many directives use metavariables in their operands. For the directive to have any effect, that metavariable must be bound to a value elsewhere: if a mutation adds an `invoked-by`, the operand needs to be bound to a method declaration. If it is not bound yet, the mutation operator also adds an `equals` directive to a method declaration in the template group. We use this shorthand, where an `equals` directive is automatically added, for the following directives: `invokes`, `invoked-by`, `refers-to`, `referred-by`, `overrides`, `constructs`, `constructed-by` and all variants of the `type` directive.

A second decision is the lack of crossover operations, where two new template groups are created by swapping a randomly chosen subtree in one template group, with a random subtree in another template. We found that crossovers mostly produce invalid templates, or templates that do not produce any matches.

The third decision is the ability to choose which operators need to be enabled or disabled. This is useful to reduce the search space as there are several "redundant" directives that are the inverse of each other, e.g. `invokes` and `invoked-by`.

The final decision concerns the use of the `protect` directive. While it prevents users from accidentally removing or abstracting away an important node, the same holds true for the EA. Adding a `protect` is useful to avoid getting the EA stuck in a local optimum, because it abstracted away too much information.

5 Generalizing Design Pattern Templates

To evaluate the EA's ability to automatically generalize or refine a template group, we will use it in the context of design patterns [7]. Given one instance of

a particular design pattern as an input template group, and all instances of the pattern as the set of desired matches, the EA is tasked to find a template group that produces all desired matches. We have chosen this context, as most design patterns involve multiple roles, played by different classes, which are related to each other in various ways. To represent a design pattern as a template group then involves multiple templates making use of several different directives. As such, we consider design patterns well-suited to put the EA to the test. The main research question to be answered in this experiment is how effective the algorithm is at finding a solution template group.[2] Can a solution be found? How many generations are required to find a solution, and how much time?

5.1 Experiment Setup

For this experiment, we chose two Java applications of a reasonable size, and where design pattern instances have been documented in the P-MARt dataset [8]: the JHotDraw v5.1 drawing application (16019 LOC; 173 classes; 1134 methods), and the Nutch v0.4 web crawler (37108 LOC; 321 classes; 1864 methods). For JHotDraw, we generalized the observer, prototype, template method, strategy and factory method patterns. For Nutch, we generalized the template method, strategy and bridge patterns. Other patterns in these projects were excluded either because the pattern documentation in P-MARt was incomplete, or because the pattern only has one instance (so there is nothing to generalize).

For each of the selected design patterns, the experiment is set up as follows. We first need to ensure an exact solution (with a fitness equal to 1) exists in the EA's search space: if it is unknown whether a solution exists, the experiment would simultaneously evaluate how expressive our template language is, which complicates evaluating the EA's effectiveness. We ensure there is an exact solution by designing it manually using only our suite of mutation operators.

Next, one instance of the design pattern is used as the EA's input template group. We do perform some preprocessing on this template by removing irrelevant methods and adding `protect` directives to those parts of the template that may not be removed. Our assumption is that the user has a notion of which parts are considered important. While this preprocessing is optional, the odds of only finding a local optimum are greater because the EA could abstract away too much (otherwise protected) information by e.g. replacing a node with a wildcard. An example of an input template group for the factory method pattern is shown in Fig. 5: most of the methods irrelevant to the pattern are removed.[3] The factory method itself is important for the pattern, and so is the fact that it instantiates something, so both have a `protect` directive (lines 2 and 6).

Finally, the EA is started using the input template group, and all instances of the design pattern as desired matches. The configuration we have used is the following: $S = 8$; $M = 22$; $R = 5$; $W_1 = 0.6$; $W_2 = 0.4$; the maximum

[2] Experiment data and instructions to reproduce the experiment are available at the Ekeko/X website: https://github.com/cderoove/damp.ekeko.snippets

[3] If a method is removed, `match|set` is always added so the template still matches.

```
1  public interface Figure extends Storable, Cloneable, Serializable {
2      [[public Connector connectorAt(  int x, int y);]@[protect]]@[match|set]}
3
4  public class RoundRectangleFigure extends AttributeFigure {
5      [public Connector connectorAt(  int x,  int y) {
6          return [new ShortestDistanceConnector(this)]@[protect];}]@[match|set]}
7
8  public interface Connector extends Serializable, Storable {
9      [public abstract Figure owner();
10     public abstract Rectangle displayBox();
11     public abstract boolean containsPoint(int x, int y);]@[match|set]}
```

Fig. 5: Input template group

number of generations is 150. Each generation contains 30 individuals, of which 8 are selections, and 22 are mutants. Tournament selection is performed using 5 rounds. The F_1 score is given a weight of 0.6; the partial score has a weight of 0.4, which we will discuss in Sec. 5.2. S, M and R are chosen based on the Essentials of Metaheuristics book [14]. The number of individuals was kept fairly low as template matching is memory-intensive, especially because the fitness of individuals is computed in parallel [4]. Finally, the following 16 operators are enabled for all experiments: Replace by wildcard/variable, Add directive (`equals`, `invokes`, `constructs`, `overrides`, `refers-to`, `type`, `subtype*`, `child*`, `match|set`), isolate expression in method, generalize references/types/invocations/constructor invocations. The disabled operators are either inverse relations of other directives, or would insert/remove AST nodes, which is unlikely to produce templates with any matches.

5.2 Experiment Results

Table 4: Experiment results

Pattern	TG	Match	Succ	Time(m)	BestFit	StdDev	GenTS	Rand	Hill
Observer	3	21	7	13.22	0.922	0.098	26.428	0.422	0.526
Prototype	3	27	4	6.75	0.814	0.231	46.75	0.172	0.307
Template method	2	47	5	76.43	0.817	0.170	56.8	0.271	0.369
Strategy	3	13	2	58.52	0.660	0.186	110.5	0.176	0.200
Factory method	3	22	2	99.68	0.682	0.187	118.5	0.201	0.239
Template method	2	7	9	18.27	0.977	0.052	83.888	0.368	0.459
Strategy	3	74	1	91.49	0.545	0.279	51	0.100	0.124
Bridge	3	69	0	64.24	0.803	0.120	-	0.168	0.260

The results of our experiment are presented in Table 4. The top 5 rows are JHotDraw patterns, and the bottom 3 are Nutch patterns. For each pattern, 10 runs were executed. The following data is provided in the table: the no. of templates in each template group (TG); the no. of desired matches (Match); no. of runs (out of 10) that found an exact solution (Succ); total time taken on average, in minutes (Time); average best fitness (BestFit); standard deviation of best fitness (StdDev); for runs that found an exact solution, the average no. of generations needed to find the solution (GenTS); average best fitness

[4] The system used in the experiment has 16GB RAM and an Intel Core i7 (Haswell).

found by a random search algorithm (Rand); average best fitness found by a hill climbing algorithm (Hill). Fig. 6 gives an idea of how the best fitness evolved per generation for one run of each pattern. Fig. 7 shows the evolution of F_1 and partial score fitness components seperately (for one run of the factory method pattern); it clearly shows the fine-grained nature of the partial score, compared to the coarse F_1 score. Because we gave the partial score a weight of 0.4, it can cause the F_1 score to temporarily lower, as can be seen around generation 75. This can occur when the EA is close to finding many more true positives, and may need to temporarily tolerate an increase in false positives.

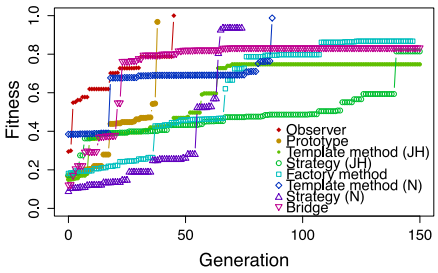

Fig. 6: Overall fitness

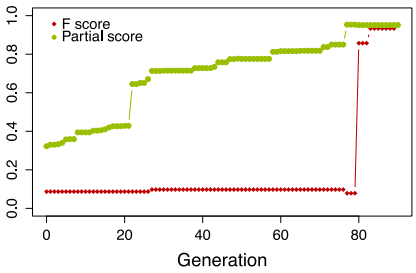

Fig. 7: Fitness components

An example solution that was generated for the factory method pattern in JHotDraw is shown in Fig. 8. It was generated from the input template of Fig. 5. The three templates respectively represent the creator (line 1-2), concrete creator (line 4-6), and product (line 8) roles of the design pattern. As the EA chooses random metavariable names, we renamed them here to improve readability. The EA has abstracted away several parts with wildcards, but retained just enough information: the `connectorAt` factory method in line 2 (which appears in all instances), the instantiation expression in line 6 and which types need to be either classes or interfaces. More importantly, the EA added directives to relate the three templates to each other: the concrete creator must be a subtype of the creator interface due to the `type` and `subtype` directives in lines 1 and 4. The factory method must return an instance of the product due to the `type` directives in lines 5 and 8. Additionally, due to the `child*` directive in line 5, the factory method may also a return a generic type where the product is a parameter. This is needed, as some instances of the pattern return a `Vector` of the product type.

```
1  public interface [...]@[(type ?creator)] extends ... {
2      [[public ... connectorAt(... x,  ... y)...]@[protect]]@[match|set]}
3
4  public class ?creator-impl extends [...]@[(subtype* ?creator)] {
5      [public [[...]@[(type ?product)]]@[child*] ...(...)... {
6          [[....new ...(...)]@[child* protect]]@[match|set]}]@[match|set]}
7
8  public interface [...]@[(type ?product)] extends ... {...}
```

Fig. 8: Generated solution for the factory method pattern

Based on the data of Table 4, we observe that the search algorithm is able to find solutions producing only the desired matches in several runs. However, we do not consistently find exact solutions in all runs, and in case of Nutch's bridge pattern no solutions were found. This indicates that the search process can get stuck in a local optimum. This happens for several reasons; for example: 1. a wildcard is added too eagerly and abstracts away information that is needed later on; 2. a relation may need to be established between two nodes using a common metavariable, but both have already been bound to two different metavariables, or 3. the fitness score was increased by relating a subclass to a superclass, but it would be better to relate it to its interface. The current suite of operators is primarily designed for ease-of-use when editing templates manually, which may not entirely correspond to operators designed for an EA. Improving the current suite to this end is considered future work.

As a basic comparison, we also performed the same experiments using a random search algorithm, as well as a hillclimbing algorithm (also 10 runs per pattern). The random algorithm continually produces random template groups, and only keeps the one with the best fitness. To generate a random group, a random number (between 0 and 50) of mutations is applied to the initial input group. Our reasoning here is that, considering the number of operations we needed to manually construct a solution, solutions must be within 50 mutations of the input template group, which is a much smaller search space than generating entirely random template groups from scratch. The hillclimbing algorithm continually applies a random mutation to the best template group. If the mutant has a better fitness, it becomes the best template group. Both the random and hillclimbing algorithm's maximum number of iterations is 500. We only show the average best fitness of both the hillclimbing and random search algorithm in Table 4, as neither of the algorithms could find an exact solution in any of its runs. This mainly indicates that the search space is too large to accidentally find a solution, and that it is possible to get stuck in local optima. As per the guidelines of Arcuri and Briand [2], we also performed a Mann-Whitney U-test to compare the BestFit with the Rand column, as well as the BestFit with the Hill column. In both cases we obtain a p-value smaller than 0.0001, confirming that the EA outperforms both the random search and hillclimbing algorithms.

5.3 Threats to Validity

The experiment was performed within only two software systems. While our focus is on snippets of code, the entire code base affects the fitness value.

Our experiment is focused on generalizing design pattern instances, so our results may not carry over to other uses of templates. While the EA itself should not change, the suite of mutation operators may need to be extended.

Finally, some combinations of directives on the same node, or a node and its children, are incompatible combinations, or require special-case behavior. We have discovered and fixed several bugs in our code because the search algorithm is exercising so many combinations of directives, but it is difficult to be exhaustive.

6 Related Work

Several program search and transformation tools exist that are, to some extent, based on code templates. This includes languages and tools such as Stratego [17], TXL [4] and JTransformer [12]. However, the constraints that are available for these languages is limited to expressing syntactic and structural characteristics, but not semantic ones (such as the directives `refers-to`, `invokes`, `overrides`, ...). The Coccinelle [3] tool does allow for semantic relations based on temporal logic within a function, but not between different functions.

A closely related tool is ChangeFactory, in which transformations can be generalized by attaching constraints/conditions to recorded changes. The conditions that can be specified are only of a syntactic nature, which limits expressivity. When considering languages that focus solely on program searches, such as BAZ [6], JQuery [11], CodeQuest [9] or PQL [15], these languages do support various semantic constraints, but they are not template-based.

With regards to our EA, several works in the field of program repair make use of genetic search or genetic programming techniques to either generate or evolve patches that fix an instance of a bug [1], [13], [18]. However, these approaches focus on repairing one instance, without looking for similar instances of the same bug. While our approach does not perform any program repairs, we can use it to describe multiple instances of a bug in one template. In this regard, the work of Meng et al. [16] is more closely related. Based on two sequences of source code modifications, each fixing an instance of the same bug, their approach can create a transformation that should find and fix all instances of a bug. The approach in this work however does not consider interprocedural modifications.

7 Conclusion and Future Work

In this work, we have presented a suite of mutation operators to modify template groups and a search-based approach that automatically generalizes and refines templates, which we have tested in the context of producing template groups that match with design pattern instances. While we found that the approach is able to either substantially improve a template or find solutions that match exactly with a desired set of snippets, a substantial amount of time is often required. However, time is less of an issue in our direction of future work. The current work has focused only on template groups performing program searches. This is a stepping stone towards also supporting program transformations, in which e.g. a patch/transformation that fixes one instance of a bug can be generalized to a transformation that fixes all instances of that bug.

References

1. Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1427–1434. ACM, 2011.

2. A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.
3. 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 *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 114–126, 2009.
4. James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
5. Coen De Roover and Katsuro Inoue. The Ekeko/X Program Transformation Tool. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 53–58, September 2014.
6. Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. The soul tool suite for querying programs in symbiosis with eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 71–80. ACM, 2011.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.
8. Yann-Gaël Guéhéneuc. P-mart: Pattern-like micro architecture repository. *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.
9. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP 2006*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2006.
10. Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.
11. Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM, 2003.
12. Gunter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd Workshop on Linking aspect technology and evolution (LATE07)*. ACM, 2007.
13. C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.
14. Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.
15. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, 2005.
16. Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.
17. Eelco Visser. Program Transformation with Stratego/XT. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 315–349. Springer Berlin / Heidelberg, 2004.
18. Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE CS, 2009.