Automated Generalization and Refinement of Code Templates with EKEKO/X

Tim Molderez and Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Belgium
tmoldere@vub.ac.be, cderoove@vub.ac.be

***Abstract*—Code templates are an intuitive means to specify source code snippets of interest, such as all instances of a bug, groups of snippets that need to be refactored or transformed, or instances of design patterns. While intuitive, it is not always straightforward to write a template that produces only the desired matches. A template could produce either more snippets than desired, or too few. To assist the users of EKEKO/X, our template-based search and transformation tool for Java, we have extended it with two components: The first is a suite of mutation operators that simplifies the process of modifying templates. The second is a system that can automatically suggest a sequence of mutations to a given template, such that it matches only with a set of desired snippets. In this tool paper, we highlight the key design decisions in implementing these two components of EKEKO/X, and demonstrate their use by walking through an example sequence of mutations suggested by the system.**

I. INTRODUCTION

Code templates are ubiquitous in search and transformation tools, and can be used to concisely describe various kinds of source code snippets of interest. The learning curve to start writing templates also is quite low, as they are written in terms of concrete source code. However, code templates can still prove difficult to specify. A template may either be too general and produce false positives, or it could be too specific and result in false negatives. Additionally, the templates we consider not only specify syntactic constraints to describe a set of snippets, but they can also use several semantic constraints. While offering additional expressivity, it emphasizes the need for some type of assistance when writing templates.

In this tool demonstration paper, we present an extension to EKEKO/X [4], our template-based search and transformation tool for Java.[1][2] This extension consist of two components, both aiming to assist EKEKO/X users: the first component is a suite of *mutation operators*, which gradually grew in our own experience of writing templates. Rather than editing a template manually without any indication whether or not the template is valid, they can also be edited using our suite of mutation operators. This suite was designed such that an operator can only be applied when it leads to a syntactically valid template. There also are two types of operators: *atomic* and *composite* operators. Where an atomic operator performs a local change in a template, composite operators can affect multiple parts. Such composite operators can avoid making accidental errors in performing common scenarios, such as abstracting away the name of a particular variable declaration and all of its uses.

The second component to assist EKEKO/X users is a search-based [9] system that can automatically generalize or refine a given template such that it matches only with a desired set of snippets. The idea is that the user can first write a rough draft of a template, and use the system to suggest a sequence of mutations that would bring the template closer to a solution.

II. OVERVIEW OF EKEKO/X

The EKEKO/X program transformation tool is built on top of the EKEKO [6] meta-programming library, which provides a logic API to perform code searches and transformations at the level of Java ASTs. As reasoning about code in terms of AST nodes requires a certain level of expertise, EKEKO/X was created to specify program searches and transformations in a more intuitive manner, in terms of code templates: A template is a snippet of Java code, in which parts can be replaced by wildcards and metavariables, and annotations called *directives* can be added. These constructs are used to either generalize or refine parts of a template. The process of *matching* a template essentially involves converting the template into a set of logic EKEKO constraints, and to find all concrete snippets of Java code that satisfy all constraints. EKEKO/X also provides support for template *groups*, in which multiple templates can be related to each other, making it possible to describe groups of related snippets. Consider the following example:

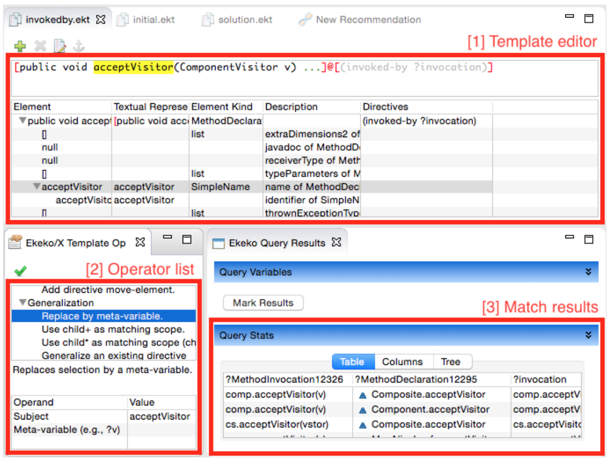
```
[....acceptVisitor(...)]@[(equals ?invocation)]

[public void acceptVisitor(ComponentVisitor v) ...]
@[(invoked-by ?invocation)]
```

This template group contains two templates: The first template matches with all calls to methods called `acceptVisitor`, and the second then looks for the corresponding method declarations. The use of an ellipsis indicates a wildcard. The `acceptVisitor` call is wrapped in square brackets, followed by `@[(equals ?invocation)]`. This notation indicates that the `acceptVisitor` method call is annotated with an `equals` directive, which binds the call to the `?invocation` metavariable. To link the method call to its declaration, the `invoked-by` directive has `?invocation` as its operand.

[1]The EKEKO/X program transformation tool, including our extension, is available for download at https://github.com/cderoove/damp.ekeko.snippets

[2]A screencast demonstrating our extension to EKEKO/X is available at https://www.youtube.com/watch?v=mWslpM8WuPs



```
[public void acceptVisitor(ComponentVisitor v) ...]@[(invoked-by ?invocation)]
```

Element	Textual Represe	Element Kind	Description	Directives
public void accept	[public void acc	MethodDeclara		(invoked-by ?invocation)
[]		list	extraDimensions2 of	
null			javadoc of MethodD	
null			receiverType of Meth	
[]		list	typeParameters of M	
acceptVisitor	acceptVisitor	SimpleName	name of MethodDec	
acceptVisitc	acceptVisitor		identifier of SimpleN	
[]		list	thrownExceptionTyp	

Add directive move-element.
Generalization
Replace by meta-variable.
Use child+ as matching scope.
Use child* as matching scope (ch
Generalize an existing directive
Replaces selection by a meta-variable.

Operand	Value
Subject	acceptVisitor
Meta-variable (e.g., ?v)	

?MethodInvocation12326	?MethodDeclaration12295	?invocation
comp.acceptVisitor(v)	▲ Composite.acceptVisitor	comp.acceptV
comp.acceptVisitor(v)	▲ Component.acceptVisitor	comp.acceptV
cs.acceptVisitor(vstor)	▲ Composite.acceptVisitor	cs.acceptVisitc

Figure 1: Overview of the EKEKO/X template editor

A screenshot of EKEKO/X's user interface is given in Fig. 1, in which our example is shown in a template editor (see [1] in Fig. 1). To modify the template, the user first selects a part to be modified, either in the textual view (top part of [1]) or the tree view (bottom of [1]). Next, a mutation operator is chosen in the list of operators (see [2]), operands are filled in (if any), and the operator can be applied. Note that the list of operators is context-sensitive to the selected part, which ensures the template remains syntactically correct.

At any time, the user can match the template and see which snippets it finds (see [3]). In case of our example, the match results contain all `acceptVisitor` declaration-call pairs found in all EKEKO/X-enabed Java projects.

III. MUTATION OPERATOR SUITE

A mutation operator, or simply operator, performs a modification in a template group. In EKEKO/X's implementation, a template is represented as a Java abstract syntax tree (AST), where nodes can be decorated with a list of directives and their operand values. As such, operators can modify an AST's structure, or add/remove directives to nodes. Table I presents a list of representative atomic operators (top half of the table) and composite operators (bottom half). Each operator can have its own list of operands. Each operator can only be applied to certain *subject* nodes, as described in the Subject column of Table I. The subject node is the "part of the template" that is selected by the user, effectively corresponding to an AST node in the template. As the subject of each operator is constrained to certain types of nodes, this also makes it possible to create EKEKO/X's operator list context-sensitive.

While not shown here, the "Add directive" operator can add several (30+) different types of directives to a node, among which are directives to relate a method call to a method declaration (`invokes`), to bind nodes to a metavariable (`equals`), relate the subject node to an ancestor node (`child`/`child*`/`child+`), relate a method to an overriding method (`overrides`), etc. The "Add directive" operator in itself is quite straightforward, as it only attaches a directive to the subject node. The actual behavior of each directive only takes effect while matching a template, where each directive specifies which logic constraints need to be satisfied.

IV. SUGGESTING TEMPLATE MUTATIONS

Our second component to assist EKEKO/X users is a search-based system that can automatically generalize and refine a template group, such that it matches only with a given set of desired code snippets. This system is based on a single-objective genetic search algorithm, and it makes use of of our suite of mutation operators.

Genetic algorithm - In short, the algorithm consists of a loop that "evolves" a set of template groups, with the aim of approaching a solution template, with a fitness value of 1. The fitness value indicates "how good" a template group is, i.e. how close its results approach the desired set of snippets. Initially, the set of template groups to be evolved only consists of the template group we would like to improve. In each iteration of the evolution loop, we produce a new set/generation of template groups based on the previous one, in which S tournament selections are made, and M mutations (S and M are user-defined constants).

A tournament selection will choose one template group by randomly picking R (user-defined) groups from the current generation, and returning the one with the best fitness out of those R. Next to making S selections, M mutations are performed: This is done by first selecting a template group (also using tournament selection), and subsequently applying a random operator, chosen from our suite of mutation operators. This operator is applied to a random template from the template group, applied to a random applicable subject node, with random operand values (if any). The most common type of operand is a metavariable, so we can randomly choose among the metavariables already present in the template group, or generate a new metavariable. Once all S selections and M mutations are made, they are combined to form a new generation of template groups. If any of the template groups produces only the desired snippets, a solution is found and the algorithm stops. Otherwise, we repeat the selection and mutation process for the new generation.

Fitness function - To compute how good a template group is, a fitness function is required. Ours is defined as follows, in terms of template group t and the set of desired matches m (where $|m| = n$):

$$fitness(t,m) = W_1.F_1(t,m) + W_2.partial(t,m)$$
$$partial(t,m) = (\sum_{i=1}^{n} \frac{matchCount(t,m_i)}{nodeCount(t)})/n$$

The fitness function consists of two components, F_1 and *partial*, where each is associated with a weight (W_1 and W_2, user-defined). The F_1 component is the traditional F-score, which considers how many true positive, false positive and false negative matches are produced by t. This results in a number in [0,1] where a value of 1 indicates that t only produces the matches in m. While this accurately describes our goal, F_1 is quite coarse-grained in the sense that F_1

Table I: A selection of EKEKO/X's suite of mutation operators

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 node	Non-mandatory child of parent	Removes the subject node.
Insert node before, after (type)	List element, non-root node	Inserts a new node of the given type before or after the subject.
Replace parent expression	Expression, and parent as well	Parent of the subject is replaced by the subject.
Isolate statement	Statement	Method body in which the subject occurs is replaced by any method body in which the subject occurs as a descendant.
Generalize references	Local variable, field declaration or formal parameter	Abstract away the name of a variable, both in the declaration and all lexical references to it.
Generalize types (qname)	Type	Abstracts away all occurrences of a particular type.
Extract template	Any non-root, non-primitive	Extracts the subject into a new template in the template group.
Generalize invocations	Method/constructor declaration	Abstracts away all invocations to the subject.

only changes when a template group produces an additional (un)desired match. This is why the more fine-grained *partial* function is introduced. In short, the partial score measures for each desired match how many template nodes could be mapped to an AST node in the desired match ($matchCount$), out of all template nodes ($nodeCount$). The intuition here is that we want to measure how close a template is to producing each desired match, i.e. a template that almost produces a desired match is better than one that is far off.

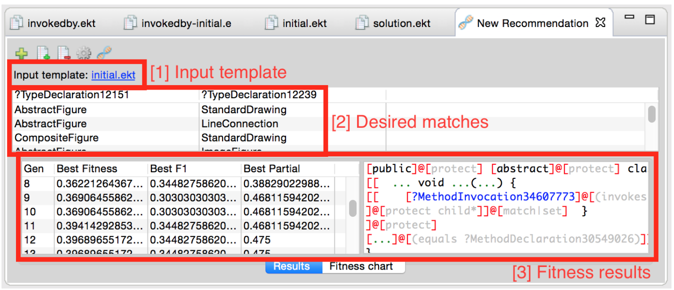
Usage - The user interface to access the automated generalization and refinement system is presented in Fig. 2. Consider the scenario where the user is currently working on a template group to detect all instances of the Template Method design pattern in a Java project. He/she currently has a template group that only describes one instance of the design pattern, and would like to invoke our system to suggest an improved template. The first step consists of selecting the template to be improved ([1] in Fig. 2). The next step is to specify the complete list of matches that are desired (see [2]), which would be all instances of the Template Method pattern. These matches can be gathered by selecting code snippets one by one, or they can be added more quickly by taking the matches of other (incomplete) templates. In case too many snippets were added to the list, they can simply be removed.

Once the list of desired matches is specified, the algorithm can be started. In our example run, the following solution was produced after 29 generations (with S=8,M=22,R=7,W_1=0.6,W_2=0.4):

```
public abstract class ... extends ... {
    [public void ...(...) {
        [[...]@[invokes ?v527655792];]@[child*]]
    [...]@[(equals ?v54716550)]]@[match|set]}

public class ?PrimVal23539979 extends ... {
    [...(...) {...}]@[(overrides ?v995827533)]@[match|set]}
```

As each new generation is produced, the results view (see [3]) is updated, showing the best template group of the new generation, its fitness, F_1 and *partial* values. A fitness chart is updated as well, shown in Fig. 3. This chart shows the interplay between the two fitness components. The *partial* score gradually pushes the templates towards producing more true positives, but does not take into account false positives,



Input template: initial.ekt

?TypeDeclaration12151	?TypeDeclaration12239
AbstractFigure	StandardDrawing
AbstractFigure	LineConnection
CompositeFigure	StandardDrawing

Gen	Best Fitness	Best F1	Best Partial
8	0.36221264367...	0.34482758620...	0.38829022988...
9	0.36906455862...	0.30303030303...	0.46811594202...
10	0.36906455862...	0.30303030303...	0.46811594202...
11	0.39414292853...	0.34482758620...	0.46811594202...
12	0.39689655172...	0.34482758620...	0.475

```
[public]@[protect] [abstract]@[protect] cla
[[  ... void ...(...) {
[[    [?MethodInvocation34607773]@[(invokes
]@[protect child*]]@[match|set]  }
]@[protect]
[...]@[(equals ?MethodDeclaration30549026)]
```

Figure 2: Overview of the automated generalization and refinement system

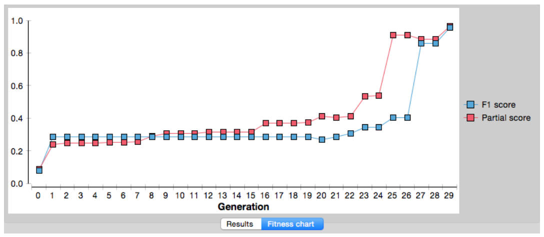
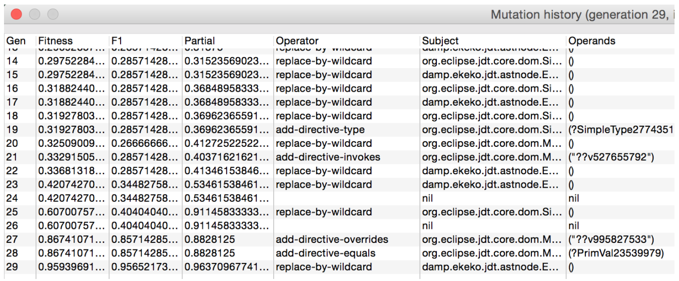


Figure 3: Fitness component chart

whereas F_1 does. This is why *partial* can increase while F_1 drops, as seen in generations 9 and 20.

When the solution template is almost found, there is a jump in both components (around generation 25). If we are about to produce an additional desired match, the template may be sufficiently generalized that it will actually describe several additional desired matches at once.

Aside from inspecting the best fitness values per generation, each template group that was generated in the search process can be inspected in detail. It can be opened in a template editor, if the user wants to resume the manual editing process. A template group's mutation history can also be inspected, showing all mutations that were used to arrive at the selected template group, starting from the initial template group. The history of our solution template is given in Fig. 4. What can be deduced from this figure is that, early on, the algorithm added several wildcards. While in this stage, most other directives would either reduce the fitness value or keep it unchanged,



Mutation history (generation 29,

Gen	Fitness	F1	Partial	Operator	Subject	Operands
14	0.29752284...	0.28571428...	0.31523569023...	replace-by-wildcard	org.eclipse.jdt.core.dom.Si...	()
15	0.29752284...	0.28571428...	0.31523569023...	replace-by-wildcard	damp.ekeko.jdt.astnode.E...	()
16	0.31882440...	0.28571428...	0.36848958333...	replace-by-wildcard	org.eclipse.jdt.core.dom.Si...	()
17	0.31882440...	0.28571428...	0.36848958333...	replace-by-wildcard	damp.ekeko.jdt.astnode.E...	()
18	0.31927803...	0.28571428...	0.36962365591...	replace-by-wildcard	org.eclipse.jdt.core.dom.Si...	()
19	0.31927803...	0.28571428...	0.36962365591...	add-directive-type	org.eclipse.jdt.core.dom.Si...	(?SimpleType2774351
20	0.32509009...	0.26666666...	0.41272522522...	replace-by-wildcard	org.eclipse.jdt.core.dom.M...	()
21	0.33291505...	0.28571428...	0.40371621621...	add-directive-invokes	org.eclipse.jdt.core.dom.M...	("??v527655792")
22	0.33681318...	0.28571428...	0.41346153846...	replace-by-wildcard	damp.ekeko.jdt.astnode.E...	()
23	0.42074270...	0.34482758...	0.53461538461...	replace-by-wildcard	damp.ekeko.jdt.astnode.E...	()
24	0.42074270...	0.34482758...	0.53461538461...		nil	nil
25	0.60700757...	0.40404040...	0.91145833333...	replace-by-wildcard	org.eclipse.jdt.core.dom.Si...	()
26	0.60700757...	0.40404040...	0.91145833333...		nil	nil
27	0.86741071...	0.85714285...	0.8828125	add-directive-overrides	org.eclipse.jdt.core.dom.M...	("??v995827533")
28	0.86741071...	0.85714285...	0.8828125	add-directive-equals	org.eclipse.jdt.core.dom.M...	(?PrimVal23539979)
29	0.95939691...	0.95652173...	0.96370967741...	replace-by-wildcard	damp.ekeko.jdt.astnode.E...	()

Figure 4: Inspecting the mutation history of a template group

wildcards can improve the partial score. When a wildcard replaces a non-leaf node, the total number of nodes in the template drops. This causes the partial score to rise, even if the template group still produces the same matches.

At some point, adding too many wildcards would increase the number of false positives. This is why the algorithm will then tend to choose a refining operator to reduce the false positives again. This is apparent in generations 20 and 27, where an `invokes` and an `overrides` directive are added.

Performance considerations - Most time in the algorithm is spent on computing fitness values, which needs to produce a template group's matches. To reduce matching time, all template groups in a generation are matched in parallel. As EKEKO/X is written in Clojure, designed with concurrency in mind, this was reasonably straightforward to implement.

To further reduce matching time, it also is possible to reduce the amount of code that needs to be searched. There is the option to only search within the classes containing desired matches. While this significantly improves the amount of time needed to match template groups, the algorithm now might produce a solution with false positives. Nonetheless, it is a sound approach to first apply the genetic algorithm against a subset of the code, then test whether the solution that was found also works against the entire codebase.

V. Related work

Several program search and transformation languages exist that are based on code templates [3], [10], [2], [14]. However, the constraints available in these languages are limited to expressing syntactic and structural characteristics, but not semantic ones. When considering languages that focus solely on program searches [12], [8], [5], these languages do support various semantic constraints, but are not template-based.

With regards to our genetic search approach, 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 [7], [1], [11]. These approaches focus on repairing one instance, without looking for similar instances of the same bug. While our system does not perform any program repairs, templates can be used to describe multiple instances of a bug in one template. In this regard, the work of Meng et al. [13] is more closely related, as its goal is to repair similar changes. Based on two instances of the same bug fix, a transformation is generated that should find and fix all instances of the bug. This approach however does not support interprocedural modifications.

VI. Conclusion

In this tool paper we have presented an extension of EKEKO/X, which consists of a suite of mutation operators, and a system that can automatically generalize and refine templates. Current experiments using this system, in which one instance of a design pattern is generalized into a template group that produces all instances, indicate that the system is able to either substantially improve a given template group, or even find a solution that matches only the desired snippets. The main direction of future work is to extend the focus from program searches to program transformations, such that e.g. a transformation that repairs one instance of a bug can be generalized to a transformation that repairs all instances.

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, NY, USA, 2011.

[2] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Guarded Program Transformations using JTL. In *46th International Conference on Objects, Models, Components and Patterns (TOOLS)*, 2008.

[3] James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[4] C. De Roover and K. 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.

[5] 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.

[6] Coen De Roover and Reinout Stevens. Building Development Tools Interactively using the Ekeko Meta-Programming Library. In *IEEE CSMR-WCRE 2014 Software Evolution Week, Tool Demo Track*, 2014.

[7] V. Debroy and W.E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, April 2010.

[8] 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.

[9] 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.

[10] 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)*, 2007.

[11] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, Jan 2012.

[12] 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, NY, USA, 2005. ACM.

[13] 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*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

[14] Romain Robbes and Michele Lanza. Example-based program transformation. In *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2008.