

A Search-based Recommender System for Source Code Templates

Tim Molderez

Software Languages Lab

Vrije Universiteit Brussel, Belgium

tmoldere@vub.ac.be

Coen De Roover

Software Languages Lab

Vrije Universiteit Brussel, Belgium

cderoove@vub.ac.be

Abstract—Source code templates are a convenient means to search and transform source code. However, it may not always be evident to write a template that produces only those matches that are desired. To assist users of the EKEKO/X program transformation tool in specifying templates, they can mark which matches are either desired or undesired. The user’s current template is then given to our genetic search algorithm, which is able to suggest an improved template by applying a sequence of mutation operations.

I. INTRODUCTION

Software continually evolves to fix bugs and to add new features. It often occurs that fixing bugs and adding features requires multiple similar changes to the source code. To avoid accidental errors in performing these changes, source code templates can be used to concisely specify all code snippets that need to be transformed.

Our existing EKEKO/X program transformation tool [1] is an Eclipse plug-in that enables its users to search and transform Java source code by means of such code templates.¹ Matching a template results in a number of corresponding source code snippets of interest, e.g. instances of a bug, snippets that should be refactored or transformed, instances of a design pattern, etc. The use of code templates lowers the learning curve of using program transformation tools, as searches and transformations are specified in terms of concrete snippets of Java code, augmented with the use of wildcards, metavariables and different kinds of annotations to alter the template’s semantics. Despite this lowered barrier to entry, it is not always easy to produce a template that matches only with the desired snippets. A template may either be too general and it produces false positives, or too specific and it produces false negatives.

To assist users in specifying EKEKO/X templates, we provide a recommendation system that suggests a series of modifications to templates such that they will only match with a set of desired source code snippets. Note that we currently focus on templates that perform source code searches; supporting transformations is considered future work.

The approach to making template modification recommendations is search-based [3]; it uses a single-objective genetic search algorithm. To give a brief overview of this approach: If

the user’s current template does not quite produce the desired set of matches, he or she can indicate which matches are still missing, as well as which of the current matches are undesired. Given this set of target matches, the current template is used to initiate the genetic search algorithm, which keeps trying out different sequences of modifications with the aim of increasing the number of desired matches. To test this algorithm, we have applied to the edge case where the user has only created a new template from one concrete code snippet, without making any modifications to it, and to use this template as input for the algorithm.

II. THE EKEKO/X PROGRAM TRANSFORMATION TOOL

The EKEKO/X program transformation tool is built on top of the EKEKO [2] meta-programming library, which provides a logic API to perform code searches and transformations at the level of Java ASTs. As mentioned before, in EKEKO/X such searches and transformations are specified in terms of source code using templates. A template essentially is a code snippet, in which parts (corresponding to AST nodes) can be replaced by wildcards and metavariables, and different kinds of annotations called *directives* can be added.

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

Fig. 1. An example template group

An example of a group with two templates is given in Fig. 1. 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` 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. Next to the `invokedby` and `equals` directives, EKEKO/X offers a whole range of different directives that can either add additional constraints to a template, or relax them.

A screenshot of EKEKO/X’s user interface is given in Fig. 2, in which the template editor for the example of Fig. 1 is shown

¹The EKEKO/X program transformation tool is available for download at <https://github.com/cderoove/damp.ekeko.snippets>

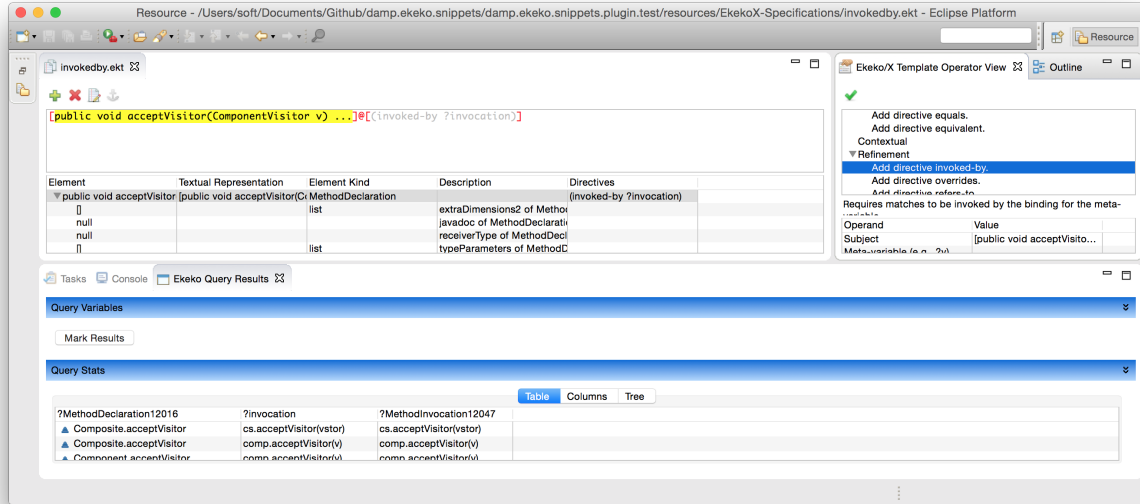


Fig. 2. EKEKO/X user interface

in the top-left. In the bottom view, the list of match results is given, listing all `acceptVisitor` declaration-call pairs found in all EKEKO/X-enabled Java projects. Finally, and perhaps most importantly, the top-right view provides access to all operators that can be used to modify a template. There are operators to add, remove and replace parts (AST nodes) of a template. There are operators to add and remove directives. Finally, there also are "composite" operators that can affect multiple parts of a template, e.g. to abstract away all occurrences of a particular variable with a metavariable.

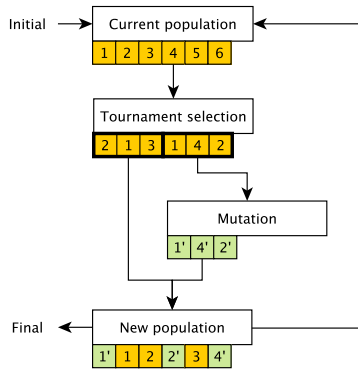


Fig. 3. Overview of the genetic algorithm

III. RECOMMENDING SOURCE CODE TEMPLATES

The example template group that was given in Fig. 1 still is reasonably straightforward to specify. However, this example could be extended to something more complex than looking for `acceptVisitor` methods. For example, it could be extended to detect all instances of the visitor design pattern. In such a case, the user may want some assistance from

our genetic search algorithm in designing a suitable group of templates.

An informal overview of this algorithm is given in Fig. 3. As input, the algorithm is given one template group, as well as a set of all desired matches. An initial population of template groups is created by making a number of copies of the input template group. (In Fig. 3, the algorithm is illustrated with a population size of 6, where each template group is labeled with its own number.) This initial population is then used as the current population in the genetic algorithm. Next, tournament selection [4] is performed by making 6 (in this case) random selections in the current population. Note that the same template group may be selected multiple times. Depending on the number of "tournament rounds", tournament selection is also more biased towards picking the "best" template group of the current population.

Measuring "how good" a template group is, is determined by its fitness value. The fitness of a template group consists of two components: The main component is an F-score determined by how many desired and undesired matches a template group produces. As this is a fairly coarse-grained measure of fitness, a second fitness component is needed. This second component is the "partial matching" component. This component is based on the idea that a template group that *almost* produces an additional desired match is better than one that does not. As such, during the template matching process, we keep track of what percentage of AST nodes, belonging to a desired match, were actually matched by the template.

Once the tournament selection phase has finished, the population is split into two partitions: one partition of template groups is carried over directly to the new population; the template groups in the other partition need to go through a mutation phase. To mutate a template group, we randomly

pick a node in one of the group's templates, and apply a random mutation operator onto it. Note that the set of available mutation operators actually is the very same set of operators available to the user when manually editing a template. In case an operator requires a number of operands, its values are chosen at random as well. The operand type of most operators typically is a metavariable. Picking a random operand value will then either choose among the existing metavariables occurring in a template group, or generate a new metavariable.

Once the template groups have been mutated, the two partitions are combined into a new population. In case one of the template groups inside the new population is a solution that produces exactly the desired matches, the algorithm can stop here. If not, the algorithm starts over again, where the population we just produced becomes the current population. This loop will continue producing new generations of populations until a solution is found. The user may of course also stop the algorithm at any time, and use the results in the current population to resume editing templates by hand.

IV. CONCLUSION

We have presented a brief overview of our search-based approach to recommend improvements to EKEKO/X source code templates, such that they produce a desired set of matches. To test the effectiveness of the algorithm itself, we are currently using it to produce template groups that describe all instances of a design pattern within a project, given one instance as input. We are also in the process of testing to what extent this approach scales to more complex templates, as well as larger Java projects.

REFERENCES

- [1] 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.
- [2] C. De Roover and R. Stevens. Building development tools interactively using the EKEKO meta-programming library. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 429–433, February 2014.
- [3] 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.
- [4] Brad L Miller and David E Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995.