Generation of Families of Similar Programs by Analogy

Ruben Heradio, Jose Antonio Cerrada, Ismael Abad, Carlos Cerrada
Dpto. de Ingenieria de Software y Sistemas Informaticos
Universidad Nacional de Educacion a Distancia
Juan del Rosal 16, E–28040, Madrid, Spain
(rheradio|jcerrada|iabad|ccerrada)@issi.uned.es

Code Generation is an increasing popular technique for implementing families of similar software products, where the code is automatically synthesized from abstract specifications written in *Domain Specific Languages (DSLs)* [2]. DSL compilers usually work as any general purpose language compiler: analyzing the source code and (a) synthesizing the final code from scratch or (b) transforming the source code into the final code. Hence, when a DSL compiler is developed the next paradox usually comes up. A DSL is a specialized, problem-oriented language. From the point of view of the DSL user, it is interesting that the DSL is as abstract as possible (supporting the domain terminology and removing the low-level implementation details). On the other hand, from the point of view of the compiler developer, the DSL abstraction makes harder to build the compiler. That is, the further DSL specifications are from the final code, the more difficult is to transform them into final code.

We propose to solve this paradox by taking advantage of a common property to DSL compilers: the big amount of *software clones* shared by the products[1]. When similar products are written in an unsystematic way, programmers reuse pieces of code by copying and pasting, i.e., selecting exemplars of code that are manually adapted and composed to create the new code. In order to reduce the development costs of DSL compilers, we propose to systematize and automatize the code generation by copying and pasting. Thus, instead of synthesizing the final code from scratch or transforming a distant input specification, we suggest to obtain the final products adapting a previously developed domain product to satisfy the input DSL specifications. We will refer to this initial product as the domain *exemplar*.

Template languages use implicitly this approach, since a text template can be viewed as a piece of an exemplar with "holes". The exemplar code that is common to all the domain products is maintained in the template, whereas the variable code is replaced by holes, that are filled with *metacode* which specifies how code must change. Unfortunately, code and metacode are strongly coupled in templates. Indeed, as argued in [3], some domain variability should be implemented as *crosscutting concerns*. When a template engine does not support *Aspect Oriented Programming*, templates may suffer *metacode tangling* (multiple variable concerns implemented simultaneously in a template) or *metacode scattering* (a variable concern implemented in multiple templates).

To overcome the templates coupling problem, the metacode should be kept out of the exemplar code. In this case, the exemplar might be processed at:

- lexical level, using *regular expressions*. Unfortunately, though regular expressions can manage text in an agile way, they have serious limitations because are internally implemented as state machines without memory and cannot manage nested or balanced constructs.
- syntactical level, using a *metaparser* or a *transformation language*. However, in most cases the simplicity of the exemplar changes does not justify to waste time either defining the exemplar language grammar or working with Abstract Syntax Trees.

We propose an intermediate solution, the *Exemplar Flexibilization Language (EFL)* [1], that provides new operators to overcome the regular expressions limitations. EFL also supports the integration with parsers to manage marginal complex exemplar modifications. In addition, EFL supports the implementation of *crosscutting generators*, that manage variability scattered over the exemplar, and the decomposition and combination of generators.

1. Note that domain product commonalities are the main reason to develop the products jointly as a family, instead of one by one.

References

[1] Heradio Gil, R. *Metodologia de desarrollo de software basada en el paradigma generativo. Realizacion mediante la transformacion de ejemplares*. Ph. D. Thesis, Departamento de Ingenieria de Software y Sistemas Informaticos de la UNED, Spain, April 2007.

[2] Pohl C. et al. *Survey of existing implementation techniques with respect to their support for the requirements identified in M3.2*. AMPLE Consortium, Version 1.2, 7/30/2007. URL: `http://ample.holos.pt`

[3] Voelter, M.; Groher, I. *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*. 11th International Software Product Line Conference (SPLC 2007).