Tool-automation for supporting the DSL learning process

Electronic Communications of the EASST
Volume X (2014)

Proceedings of the Second International Workshop on Open and Original Problems in Software Language Engineering (OOPSLE 2014)

Tool-automation for supporting the DSL learning process

Federico Tomassetti, Cristhian Figueroa, Daniel Ratiu

5 pages

Guest Editors: Anya Helene Bagge, Vadim Zaytsev
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Tool-automation for supporting the DSL learning process

Federico Tomassetti[1], **Cristhian Figueroa**[1], **Daniel Ratiu**[2]

[1]Politecnico di Torino,[2]fortiss gGmbH

Abstract: Recent technologies advances reduced significantly the effort needed to develop Domain Specific Languages (DSLs), enabling the transition to language oriented software development. In this scenario new DSLs are developed and evolve at fast-pace, to be used by a small user-base. This impose a large effort on users to learn the DSLs, while DSL designers can use little feedback to guide successive evolutions, usually just based on anecdotal considerations.

We advocate that a central challenge with the proliferation of DSLs is to help users to learn the DSL and providing useful analyses to the language designers, to understand what is working and what is not in the developed DSL.

In this position paper we sketch possible directions for tool-automation to support the learning processes associated with DSL adoption and to permit faster evolution cycles of the DSLs.

Keywords: Domain Specific Languages, DSL Learning, AST, Graph similarity, Language Oriented Programming

1 Motivation

Domain Specific Languages (DSL) are a very efficient mean to express business domain level functionality: the more specific they are, the more tailored and therefore efficient they will be to develop narrow aspects of the business domain.

The language oriented programming paradigm [Dmi04] is made today possible by several technical advances. Firstly, using Language Workbenches[1] like Xtext, MetaEdit+ or Jetbrains MPS, building an entire DSL with its environment, i.e., syntax, code generator or a model interpreter, and advanced editors support can now be done in a matter of days. Secondly, in case of language workbenches with advanced support for language modularization, it is easy to realize and adopt new extensions for existing languages. Developers can program their system in a host language and use domain specific language extensions whenever they are needed – e.g. in the mbeddr project, the host language is C implemented in JetBrains MPS, different domain specific extensions were built to support variability, requirements [VRT13] or analyses [RSVK12]. Thirdly, DSLs can evolve as the domain is better understood or new requirements arise [HRW10], therefore new constructs have to be learned, and new idioms emerge as the codebase of the DSL grows.

As consequence of the reduced cost of building, evolving and adapting DSLs, company-specific or even project-specific DSLs are now being developed in an iterative and more agile manner. *As entire DSL or single extensions are rapidly implemented, evolved and adapted, the*

[1] See http://www.languageworkbenches.net/

ability of practitioners to learn them becomes the bottleneck. The learning process involve both DSL users (developers writing code with the DSL) and DSL designers (developers designing and implementing the DSL). Recent work shows that often not all of the constructs of a DSL are used [TC13]. One of the reasons of this phenomenon is that users are not aware of their existence, hence they do not know completely the language. Another reason could be that DSL users consider some constructs not useful, hence the DSL designer did not address their real necessities.

Modern Language Workbenches emerged only recently, therefore DSL design is a practice not yet established. To help DSL design to be attested as a mature practice appropriate guidelines and metrics are needed. We suggest they could be derived from the analysis of the learning process.

The learning curve sums up with an already existing problem: the reluctance of practitioners on investing in learning new DSLs; or the frustration of having to do with different new constructs, or even having them used in a wrong manner. *Adequate support to reduce the investment required from developers to learn new DSLs is essential for the proliferation of DSLs.*

This paper aims to foresee a possible approach to support the DSL learning process. We propose to use code recommenders to support DSL learning and to provide feedback to DSL designers.

2 Proposal

Our goal is to collect the knowledge and wisdom about the DSL usage from available sources, in order to provide useful insights, to different actors, as they need them.

First, we want to guide beginner users to write higher quality DSL statements. It could be done through auto-completion and suggestions of complete examples:

- **auto-completion** would be based not only on syntax rules but also on semantic constraints. Semantic constraints are validation rules expressed in the language definition. Among the allowed constructs, we would suggest the most typical patterns of usage.
- **complete examples** and associated documentation would be shown when relevant to the current context.

Second, we plan to provide feedback to the DSL designers about the real **usage** of the DSL. As part of the feedback, statistical information about the most used and unused constructs could help the designer to identify parts of the DSL which are considered useless or confusing. Additionally, more elaborate processing of the usage information could lead to identify **emerging patterns**, which can be the hint indicating a missing abstraction for which explicit support could be added to the language.

As shown in Figure 1, we plan to reuse the grammar and the constraints defined by the language designer while realizing the DSL and the related tooling. We would also reuse examples provided for documentation purposes. Later, the users themselves could contribute by simply writing DSL code: repetitive, consolidate patterns would be extracted and used together with examples to guide users. Through the analysis of the actual usages feedback would be provided to the DSL designer who could iteratively evolve the language. New evolutions of the language

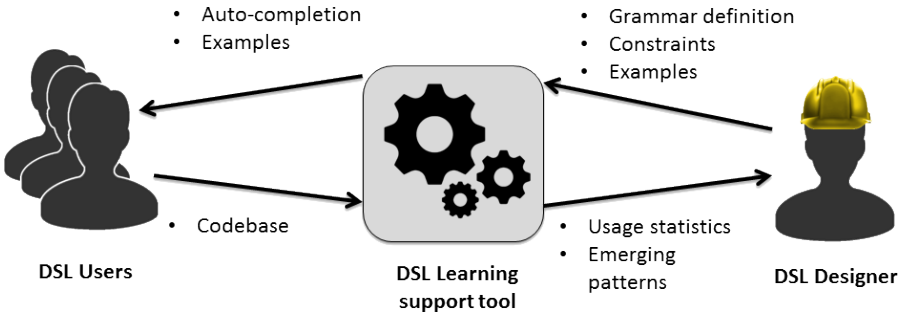

Figure 1: Schema of the DSL learning cycle

would be suggested to DSL users, who could keep in pace with the DSL evolution without consulting continuously the documentation.

3 Foreseeable design

To provide **autocompletion** we would transform the code being currently written by the DSL user, derive the corresponding AST and build possible extensions of that AST on the base of examples and common usages. Serializing those extended AST we would build a list of terms to be proposed as auto-completion items. This process is represented in Figure 2. The extended AST would have to satisfy the syntactic and semantic rules specified by the language.

To identify the appropriate **examples** we would compare the AST of the code being currently written with the AST of the examples, with graph similarities techniques.

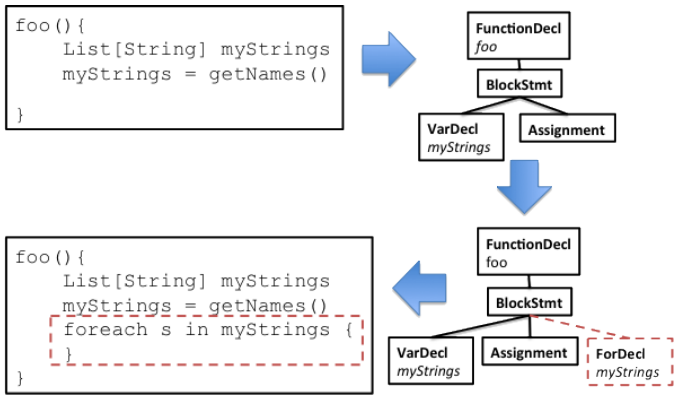

Figure 2: How we could implement autocompletion

Usage statistics could be easily calculated from the ASTs derived from the codebase written by DSL users.

To identify **emerging patterns** we should look for most frequent structures which appear in the codebase and containing redundancy.

4 Discussion

Through this automatic tooling we plan to decrease the cost of learning DSLs by supporting users and help them to achieve their immediate goals, reducing their reluctance in learning the language. Moreover DSL designers could benefit from the experience of the whole userbase, in a more systematic way, rather than relying on anecdotal indications, at best.

We think DSL design and deployment is a collaborative effort which involve both users and designers. The learning process, which goes through a number of iterations, need to be supported as a whole: the definitive goal is to base DSL design on facts and to help DSL users in learning how to learn languages.

Future work should be done in different directions: first we should study existing literature to learn from the experience matured on API recommenders and the general lesson learned on language learning. As second point we need to study technological solutions, reusing existing graph similarities algorithms.

There are still many open questions about future steps. We currently do not know how we could validate our approach: how we could measure the effort spent on learning and the reduction hopefully granted by our approach? How we could measure the understanding of a whole language by its users? Should we ask the opinions of users, measure their productivity, calculate how much time is required to each extension to be absorbed? Which are the measures most useful to language designers? How exactly should we identify emerging patterns?

Bibliography

[Dmi04] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard* 1(2), 2004.

[HRW10] M. Herrmannsdoerfer, D. Ratiu, G. Wachsmuth. Language Evolution in Practice: The History of GMF. In Brand et al. (eds.), *Software Language Engineering*. LNCS 5969, pp. 3–22. Springer, 2010.

[RSVK12] D. Ratiu, B. Schaetz, M. Voelter, B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*. Pp. 9–15. 2012.

[TC13] R. Tairas, J. Cabot. Corpus-based analysis of domain-specific languages. *Software and Systems Modeling*, pp. 1–16, 2013.

[VRT13] M. Voelter, D. Ratiu, F. Tomassetti. Requirements as First-Class Citizens: Integrating Requirements closely with Implementation Artifacts. In *6th Int. Workshop on Model Based Architecting and Construction of Embedded Systems*. 2013.