Puzzle Metrics: A Tool for Analyzing Potential Reuse in Domain-Specific Languages

David Méndez-Acuña, José A. Galindo, Benoit Combemale,
Arnaud Blouin, and Benoit Baudry

University of Rennes 1, INRIA/IRISA. France
{david.mendez-acuna,jagalindo,benoit.combemale,arnaud.blouin,benoit.baudry}@inria.fr

Abstract. The use of domain-specific languages (DSLs) has become a successful technique in the implementation of complex systems. However, the construction of this type of languages is time-consuming and requires highly-specialized knowledge and skills. Hence, researchers are currently seeking approaches to leverage reuse during the DSLs development in order to minimize implementation from scratch. An important step towards achieving this objective is to identify commonalities among existing DSLs. These commonalities constitute potential reuse that can be exploited by using reverse-engineering methods. In this paper, we present an approach intended to identify sets of DSLs with potential reuse. We also provide a mechanism that allows language designers to measure such potential reuse in order to objectively evaluate whether it is enough to justify the applicability of a given reverse-engineering process. We validate our approach by evaluating a large amount of DSLs we take from public `GitHub` repositories.

1 Introduction

The use of domain-specific languages (DSLs) has become a successful technique to achieve separation of concerns in the development of complex systems [6]. A DSL is a software language in which expressiveness is scoped into a well-defined domain that offers a set of abstractions (a.k.a., language constructs) needed to describe certain aspect of the system [5]. For example, in the literature we can find DSLs for prototyping graphical user interfaces [15], specifying security policies [13], or performing data analysis [8].

Naturally, the adoption of such language-oriented vision relies on the availability of the DSLs needed for expressing all the aspects of the system under construction [3]. This fact carries the development of many DSLs which is a challenging task due the specialized knowledge it demands. A language designer must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, and interpreters. As a mater of fact, the ultimate value of DSLs has been severely limited by the cost of the associated tooling (i.e., editors, parsers, etc...) [10].

To improve cost-benefit when using DSLs, the research community in software languages engineering has proposed mechanisms to increase reuse during the construction of DSLs. The idea is to leverage previous engineering efforts and minimize implementation from scratch [17]. These reuse mechanisms are based on the premise that "software languages are software too" [9] so it is possible to use software engineering techniques to facilitate their construction [11]. In particular, there are approaches that take ideas from Component-Based Software Engineering (CBSE) [4] and Software Product Lines Engineering (SPLE) [19] during the construction of new DSLs.

A classical way for adopting the aforementioned reuse mechanisms is to group language constructs into interdependent language modules that can be later extended and/or imported as part of the specifications of future DSLs. This type of solution has ultimately gained momentum and, nowadays, there are a diversity of approaches that facilitate such modular DSLs design [16,14,12]. However, the definition of language modules that can be actually useful in future DSLs is not easy. In part, this is due to the fact that the reuse of a language module implies the reuse of all the constructs it offers and language designers do not always have the information that allow them to predict the correct combination of constructs that go well together.

A more pragmatical approach to leverage reuse in the construction of DSLs is to focus on legacy DSLs [7]. That is, to exploit reuse in existing DSLs that are were not necessarily built for being reused but that share some commonalities (i.e., they provide similar language constructs). Using this strategy, language designers can obtain valuable reuse information from real DSLs. For example, they can identify groups of constructs that are frequently used together. Then, a catalog of language modules can be extracted from the commonalities of the DSLs by means of reverse-engineering methods. As the reader may guess, the very first step towards the application of such *a posteriori* reuse strategy is to identify commonalities within a set of existing DSLs. Ideally, this process should be automatic because comparing constructs manually can be time consuming and error prone. In addition, language designers should have mechanisms to objectively evaluate whether those commonalities represent potential reuse enough to justify the effort associated to a reverse-engineering process.

In this paper, we present a tool that takes as input a set of DSLs and identifies the commonalities existing among them. To do so, we perform static analysis on the artifacts where the DSLs are specified and compare language constructs at the level of the syntax and semantics. Besides, our tool computes a set of metrics that permit to objectively evaluate those commonalities. This second part of our approach is based on some reuse metrics already proposed in the literature for the general case of software development [1,2] that we adapt them to the specific case of DSLs development.

2 Proposed approach

Given a set of existing DSLs (that we term as the *input set*) our approach is intended to identify commonalities –and so, potential reuse–. Then, we provide some metrics that permit to objectively evaluate those commonalities. The reminder of this section explain how we tackle this problem.

2.1 Identifying commonalities

In the first part of our approach, we perform static analysis in syntax and semantics of a given set of DSLs in order to build a pair of Venn diagrams that allows language designers to easily visually identify commonalities among DSLs under study. The idea is to visualize each DSL as a pair of two sets: The first one is a set of metaclasses representing the syntax, and the second one is a set of domain-specific actions representing the semantics. Syntactic and semantic commonalities are represented as intersections between the corresponding sets. To this end, we designed an algorithm that is able to compute the all intersections among the syntax of the DSLs in the input set.

Our algorithm for detecting **syntactic intersections** can be described as by the function that receives a set of metamodels (one for each DSL of the input set) and returns a set of tuples containing all the intersections among these metamodels. Note that there can be intersections among any of the combinations of the input set. Hence, in the result there is a tuple for each of the possible combinations of the input metamodels (i.e., the power set). Similarly, our algorithm for detecting **semantic intersections** can be described as a function that receives a set of aspects (one for each DSL of the input set) and returns a set of tuples containing all the intersections among these aspects. Figure 1 shows the Venn Diagram for the case of our motivating scenario. In that figure we can see that the family is an overlapping family in terms of the abstract syntax.

2.2 Measuring potential reuse

As a second part of our analysis, we propose a quantitative evaluation of the potential reuse represented as commonalities in a set of DSLs. Such evaluation is based on the metrics introduced in [1] which are originally defined for software products in general and that we adapted for the case where the software products are domain-specific languages. We choose these metrics because they perfectly fit in the idea of conceiving potential reuse as intersections of Venn diagrams and because they were already evaluated in an industrial case study [2].

Figure 2 shows the reuse metrics graphically. They are intended to normalize the size of the commonalities existing among the DSLs of the input set. To do so, the idea is to see the intersections in terms of percentages. Hence, they answer questions such as: what is the percentage of language constructs of a given DSL that are also defined in another DSL in the input set? In this section we present these metrics in terms of the formulas we used to compute them. It is

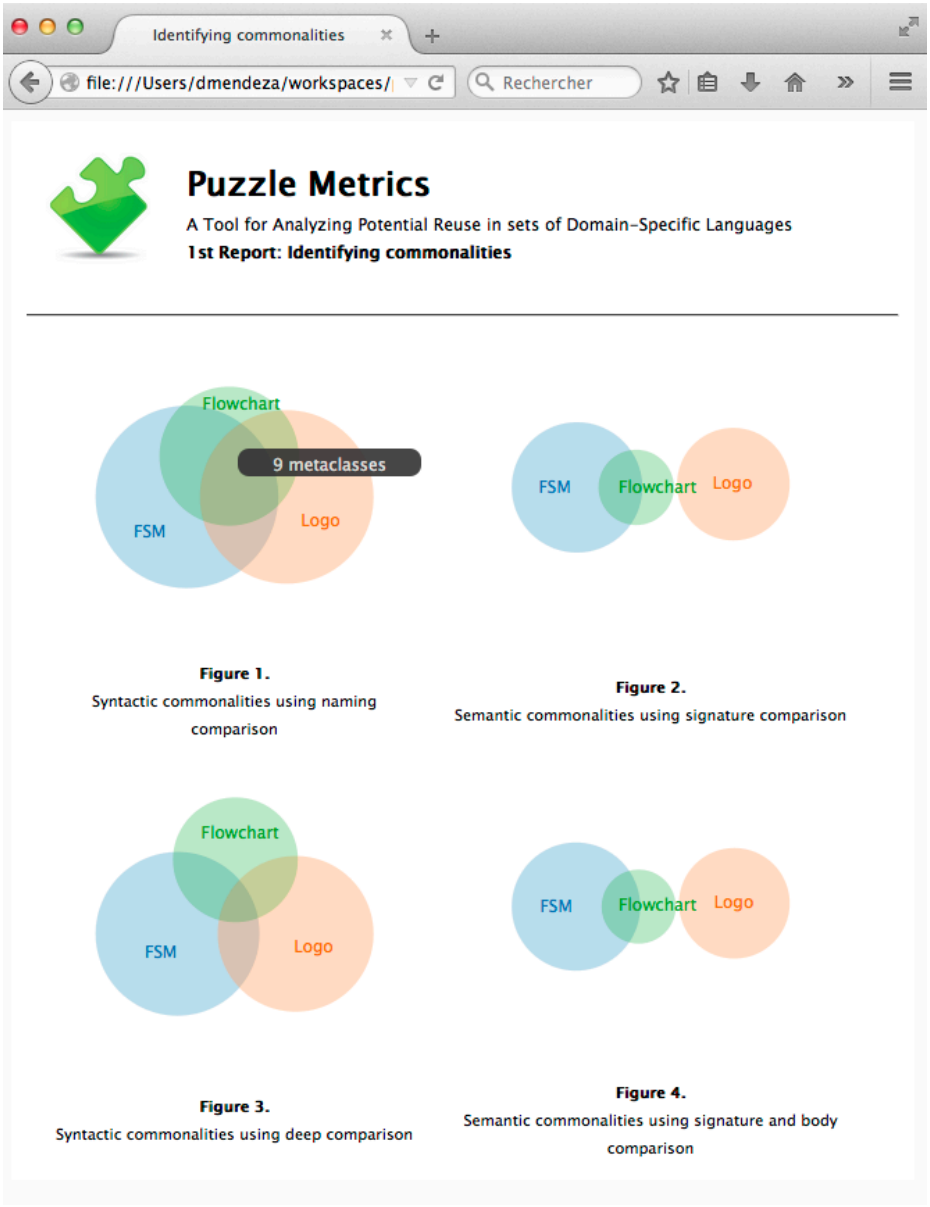

Fig. 1: Visualizing syntactic and semantic commonalities

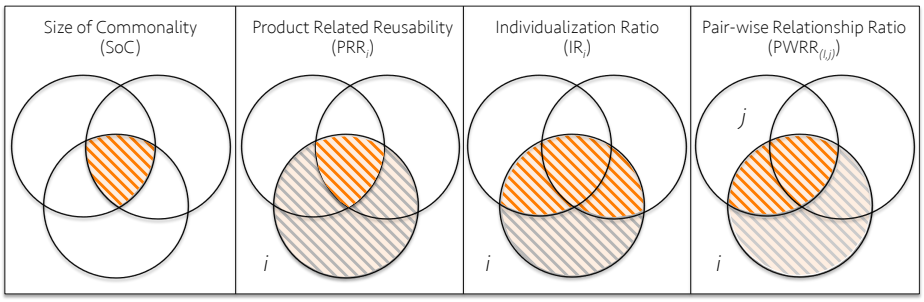

Fig. 2: Metrics for evaluation of potential reuse

important to remember that the results provided by those metrics also depend on the comparison operator.

- **Size of Commonality (SoC):** The size of commonality metric is intended to measure the amount of metaclasses and domain-specific actions that are shared by all the DSLs in the input set. It is defined as the size of the intersection of all the DSLs in the input set.

  The usefulness of this metric relies on the identification of a common *core* among all the DSLs. This is quite relevant because there are certain reuse approaches for DSLs (such as the one presented in [18]) where the existence of a core is a prerequisite.
- **Product-Related Reusability (PRR_i):** The product-related reusability is a metric that, for each DSL of the input set, measures the percentage of the metaclasses and domain-specific actions that are defined in the core detected by the SoC metric. It permits to see how related is each DSL with the core of the input set.
- **Individualization Ratio (IR_i):** The individualization ratio is a metric that, for each DSL in the input set, measures the percentage of the metaclasses and domain-specific actions that are common with at least another DSL. This metric shows how particular is each language. That is, how many metaclasses and domain-specific actions are tailor-made for the DSL.
- **Pairwise Relationship Ratio ($PWRR_{(i,j)}$):** The pair-wise relationship ration is a metric that measures the reusability between each possible pair of DSLs in the input set. This metric can be seen as a pair-wise similarity that indicates how different is a DSL for each of the other DSLs in the input set.

3 Downloading and using PUZZLEMETRICS

PuzzleMetrics can be downloaded from the following URL The download corresponds to a virtual machine where there is an Eclipse distribution that already

contains all the dependencies. Besides, the image contains a workspace that contains all the families used in this paper. That is, the motivating example presented in section X and the case studies presented in section Y.

4 Conclusions and Perspectives

References

1. C. Berger, H. Rendel, and B. Rumpe. Measuring the ability to form a product line from existing products. volume abs/1409.6583. 2014.
2. C. Berger, H. Rendel, B. Rumpe, C. Busse, T. Jablonski, and F. Wolf. Product Line Metrics for Legacy Software in Practice. In *SPLC 2010 : Proceedings of the 14th International Software Product Line Conference*, pages 247–250. Univ., Lancaster, 2010.
3. T. Clark and B. Barn. Domain engineering for software tools. In I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, editors, *Domain Engineering*, pages 187–209. Springer Berlin Heidelberg, 2013.
4. T. Cleenewerck. Component-based dsl development. In F. Pfenning and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 245–264. Springer Berlin Heidelberg, 2003.
5. B. Combemale, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, June 2014.
6. S. Cook. Separating concerns with domain specific languages. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 1–3. Springer Berlin Heidelberg, 2006.
7. T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, Oct. 2015.
8. J. Eberius, M. Thiele, and W. Lehner. A domain-specific language for do-it-yourself analytical mashups. In A. Harth and N. Koch, editors, *Current Trends in Web Engineering*, volume 7059 of *Lecture Notes in Computer Science*, pages 337–341. Springer Berlin Heidelberg, 2012.
9. J.-M. Favre, D. Gasevic, R. L‰ommel, and E. Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, volume 6563 of *LNCS*, pages 316–326. Springer, 2011.
10. J.-M. Jézéquel, D. Méndez-Acuña, T. Degueule, B. Combemale, and O. Barais. When Systems Engineering Meets Software Language Engineering. In *CSD&M'14 - Complex Systems Design & Management*, Paris, France, Nov. 2014. Springer.
11. A. Kleppe. The field of software language engineering. In D. Ga?evi?, R. L‰ommel, and E. Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 1–7. Springer Berlin Heidelberg, 2009.
12. H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.
13. T. Lodderstedt, D. Basin, and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer Berlin Heidelberg, 2002.

14. M. Mernik. An object-oriented approach to language compositions for software language engineering. *J. Syst. Softw.*, 86(9):2451–2464, Sept. 2013.
15. S. Oney, B. Myers, and J. Brandt. Constraintjs: Programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 229–238, New York, NY, USA, 2012. ACM.
16. E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.
17. T. van der Storm, W. Cook, and A. Loh. Object grammars. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 4–23. Springer Berlin Heidelberg, 2013.
18. S. Zschaler, D. S. Kolovos, N. Drivalos, R. F. Paige, and A. Rashid. Domain-specific metamodelling languages for software language engineering. In M. van den Brand, D. Ga?evi?, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 334–353. Springer Berlin Heidelberg, 2010.
19. S. Zschaler, P. Sánchez, J. a. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. a. Araújo, and U. Kulesza. Vml* a family of languages for variability management in software product lines. In M. van den Brand, D. Ga?evi?, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer Berlin Heidelberg, 2010.