Identifying and Evaluating Potential Reuse in the Development of 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. Domain-specific languages (DSLs) are becoming 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 objectively evaluate whether potential reuse 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 approach to achieve separation of concerns in the development of complex systems [5]. A DSL is a software language in which expressiveness is scoped into a well-defined domain that offers the abstractions needed to describe certain aspect of the system [3]. For example, in the literature we can find DSLs for prototyping graphical user interfaces [15], specifying security policies [13], or performing data analysis [6].

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. This fact carries the development of these DSLs which is a challenging task also due to the specialized knowledge it requires. 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 matter of fact, the ultimate value of DSLs has been severely limited by the cost of the associated tooling (i.e., editors, parsers, compilers, etc...) [10].

To deal with such complexity, the research community in Software Languages Engineering (SLE) has proposed mechanisms to increase reuse within the construction of DSLs. The idea is to leverage previous engineering efforts and minimize implementation from scratch. These reuse mechanisms are based on the premise that "software languages are software too" [7] 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) and Software Product Lines Engineering (SPLE) during the construction of new DSLs [12,18,20].

The aforementioned reuse mechanisms can be adopted by means of two different approaches: *top-down* and *bottom-up*. The top-down approach proposes the design and implementation from scratch of reusable language modules that can be employed in the construction of several DSLs. Differently, the bottom-up approach proposes the use or reverse-engineering processes to extract those language modules from existing DSLs that share some commonalities which can be properly encapsulated. Whereas the major complexity in top-down approaches is that language designers should design language modules by trying to predict their potential reuse; bottom-up approaches do not have to deal with that problem. Rather, the extraction of the reusable language modules is based on the detection of commonalities in existing DSLs. Consequently, bottom-up approaches can be considered as promising approach and, indeed, there are already in the literature some works (e.g., [17]) advancing towards that direction.

It is worth noting that the very first step towards a reverse engineering process for DSLs is the identification of potential reuse. In other words, before applying reverse engineering, language designers need: (1) to identify sets of DSLs with potential reuse and; (2) be sure that the potential reuse is enough to justify the associated effort. In this paper, we present a proposal to automatically tackle this issue. Concretely, we introduce an approach that takes as input a set of DSLs and identifies which of them share commonalities so they have potential reuse. To do so, we perform static analysis on the artifacts where the DSLs are specified (i.e., metamodels and semantic definitions). Besides, our approach computes a set of metrics on the DSLs that permit to objectively evaluate if the potential reuse justifies the effort required by the reverse-engineering process. 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.

We validate our approach by taking as input an important amount of languages available on GitHub public repositories. The results of this validation are quite promising. All the ideas presented in this paper are implemented in an Eclipse-based tool that can be downloaded and installed as well as the validation scenarios.

This paper is organized as follows: After this introduction, Section 2 introduces a set of basic definitions we use along the paper. Section 3 presents a motivating scenario that illustrates both the problem and the solution tacked in this paper. Section **??** introduces the foundations of our approach. Section 6 validates the approach on DSLs we take from GitHub. Section X discusses the threads to validity. Section X presents the related work and, finally, Section X concludes the paper.

2 Background

2.1 Domain-Specific Languages

Like general purpose languages, domain specific languages are specified in terms of three interdependent dimensions: abstract syntax, concrete syntax, and semantics [9]. The *abstract syntax* refers to the structure of the DSL expressed as the set of concepts that are relevant to the domain and the relationships among them. The *concrete syntax* refers to the association of the language concepts to a set of symbols (either graphical or textual) that facilitate the usage of the DSL. These representations are usually supported by editors that allow users to write programs using the symbols defined by the concrete syntax acting as the graphical user interface of the DSL. Finally, the *semantics* of a DSL refers to the meaning of its domain concepts. This meaning is expressed through static constraints and dynamic behavior specification. The static constraints usually correspond to the type system of the DSL. In turn, the dynamic behavior defines the manner in which domain concepts are manipulated at runtime, typically through the definition of an interpreter or compiler.

In this paper we are interested in executable DSLs which abstract syntax is specified by means of metamodels and semantics is specified operationally as a set actions (or methods) on the metaclasses of the metamodel. We adopt the vocabulary presented in [4] and we term these actions as *domain-specific actions* (DSAs). The concrete syntax is out of the scope of this paper.

2.2 On the notions of *commonalities* and *potential reuse* in DSLs

As aforementioned, a DSL is intended to focus on a specific aspect of the system under construction. Due to the complexity of current systems, there is a large amount of concerns to deal with. As a result, there is a proliferation of DSLs in the literature [14]. Although many of those existing DSLs are completely different and with independent domains; it is also true that we can find related DSLs with overlapping domains. Moreover, there are set of DSLs for which the domains can be hierarchically organized [19, p. 60-61].

Figure 1 illustrates the situation explained above. At the left of the figure there are two DSLs that are totally independent. That means that they do not share any of their domain abstractions, and consequently, there is not potential reuse between them. Differently, the two DSLs shown in the center of the figure have overlapping domains. That means that there are a subset of domain abstractions that are **"equal"** in both languages. In such case, we say that the domains abstractions are shared by the two DSLs and that the domains overlap. Note that because domain abstractions are defined in the specification of the two DSLs, it is natural to thing that the definitions are equal. That means that those definitions are replicated in the specifications of the DSLs. Thus, there is potential reuse. The case presented at the right of the figure is even more interesting. In that case there is a DSL that contains all the domain abstractions of another DSL. In the vocabulary introduced in [19, p. 60-61] that means that there is a domain hierarchy i.e., the domain of the DSL 1 is a subdomain of the domain of the DSL 2. In that case, there is potential reuse because in the sense that the DSL 2 could import the DSL 1 as part of its specification without having to redefine all those domain abstractions.

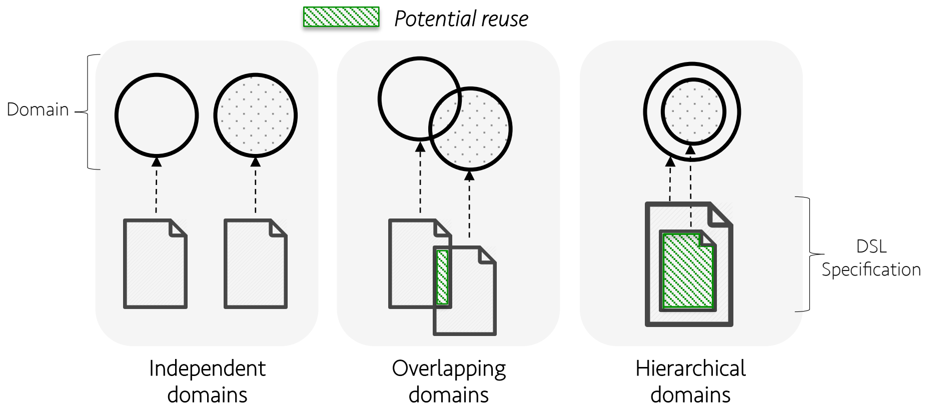

Fig. 1: Commonalities between domains and potential reuse

2.3 Equivalence between language constructs

So far, we have based the notion of potential reuse in DSLs on the commonalities existing in the specification of a set of DSLs. Nevertheless, this supposes that we are able to compare two language constructs in order to know whether their specifications are equivalent. The comparison of two language constructs relies on two dimensions: (1) comparison of the meta-classes in the abstract syntax; and (2) comparison of the domain-specific actions in the semantics. The reminder of this section is dedicated to explain this notion of equality for both meta-classes and DSAs.

Comparing meta-classes. A first approach to compare meta-classes is by comparing their name. Two meta-classes are equal if they have the same name. This approach results quite useful if we consider that, in most of cases, the name of the meta-class refers to a domain concept and it is quite probable that, we can find potential reuse. Unfortunately, comparison of meta-classes by using only their names might have some problems. There are cases in which two meta-classes with the same name are not exactly the same. This may occur either because they not refer to the same language construct or because one meta-class is more specialized than the other one. In such cases, an approach that certainly helps is to compare meta-classes not only by their names but also by their attributes and references. Although this second approach might be too restrictive, it implies that the specification of the two meta-classes are exactly the same so potential reuse is guaranteed.

Comparing domain-specific actions. Intuitively, the comparison of two domain-specific actions can be performed by checking if their signatures are equal. This approach is practical and also reflects potential reuse; one might think that the probability that two domain-specific actions with the same signatures are the same is elevated. However, as the reader might imagine, there are cases in which signatures comparison is not enough. Two domain-specific actions can perform different computations even if they have the same signatures. As a result, a second approach for comparing DSAs relies in the comparison of the body of the operations. Note that comparing the body of the operations can be arbitrary complex task. Indeed, if we try to compare that the actions are semantically equivalent we rely on the semantic equivalence problem that, indeed, is known as be undecidable. In this case, we prefer a more conservative approach and we only compare the structure of the DSA. We use the approach proposed in X. Basically, it accepts operations with the same structure and permits certain differences. Concretely, names of variables and parameters can change.

Semantical variability. A necessary condition for deciding whether two language constructs are equivalent is that both, the meta-class and the domain-specific actions are equivalent. This guarantees that the specification is the same not only at the level of the abstract syntax but also at the level of the semantics. However, there is a phenomenon in the literature that corresponds to semantical variability. There is semantical variability when there there are two constructs that having the same abstract syntax (i.e., their meta-classes are equal) differ in the domain-specific actions. This case is of interest for us because even in the presence of semantical variability we can have some potential reuse. If the meta-classes of two constructs are the same we can reuse them even if their domain-specific actions are different.

3 Motivating scenario

In this section we present a motivating scenario that we use to illustrate the preliminary definitions introduced in the previous section and that we will use to present our approach. This scenario is a set three DSLs.The first one is the classical Logo that has been historically used it in education. The second one is a DSL for expressing flow charts. Finally, the third one is a DSL for expressing simple state machines.

Figure 2 shows the metamodels corresponding to each of the DSLs. Although each language is independent, it is clear that there are certain commonalities that represent potential reuse. The most important commonality is highlighted in yellow and corresponds to the constructs for expressing simple arithmetic expressions. The three DSLs of the scenario use the same constructs.

4 Automatically detecting potential reuse

Our approach for identifying potential reuse in a set of DSLs is based on static analysis of the metamodels and the domain specific actions and the later the visualization of the set of DSLs in the form of Venn diagrams. This type of visualization allows the language designer the visualization of the overlapping existing among the given set of DSLs.

Figure 3 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. In the case of the semantics the results are quite interesting. Note that depending on the type of comparison operator we have different results. When the comparison operator is the naming, we have the same overlapping shape that in the case of the abstract syntax. However, when the operators become more restrictive the overlapping region is reduced.

5 Objectively evaluating the potential reuse

The second part of our analysis corresponds to a quantitative evaluation of potential reuse. To do so, we include the reuse metrics presented in X and we adapt them for the case of families of DSLs. In this section we present these metrics in terms of the formulas we used to compute them based on the basic definitions presented in section X. Besides, we show the output that our tool provides for our motivating scenario.

- **Size of Commonality (SoC):** This metric shows the size of the core with respect to the rest of the family. It is calculated as the percentage of constructs/methods that are included in the core with respect to the union of the constructs/methods of all the DSLs of the family.

  On the other hand, the larger the core the smaller the variability. So the amount of required decomposition is reduced and the variability model is simpler. So, one may think that a family where the core is big is a family where the variability is easier to manage.

- **Product-Related Reusability (PRR_i):** This metric shows the percentage of reuse of each DSL with respect the core. Concretely, it shows for each product the amount of constructs/methods that are included in the core.

  This metric is important because we can detect the product more related to the core. This identification will be helpful at the moment of defining that core.

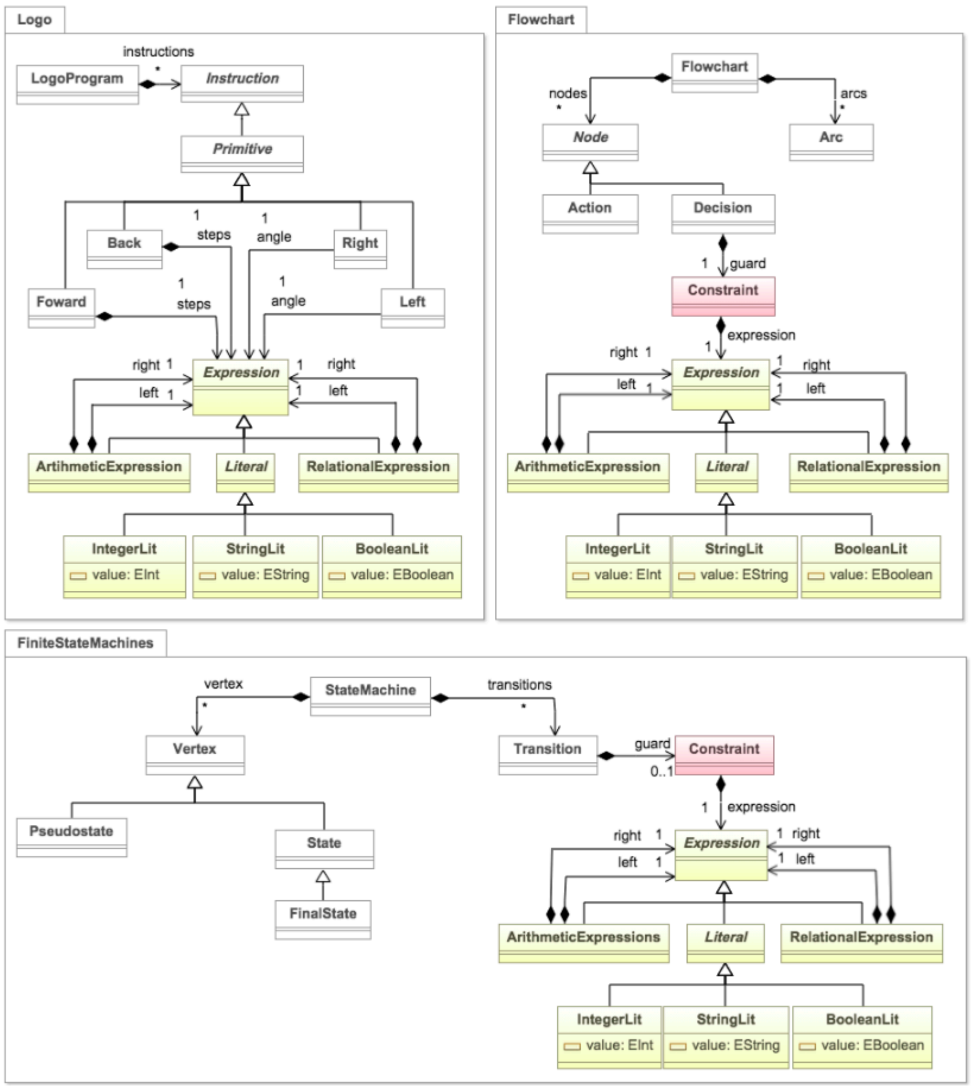

Fig. 2: Different relationships between DSL domains

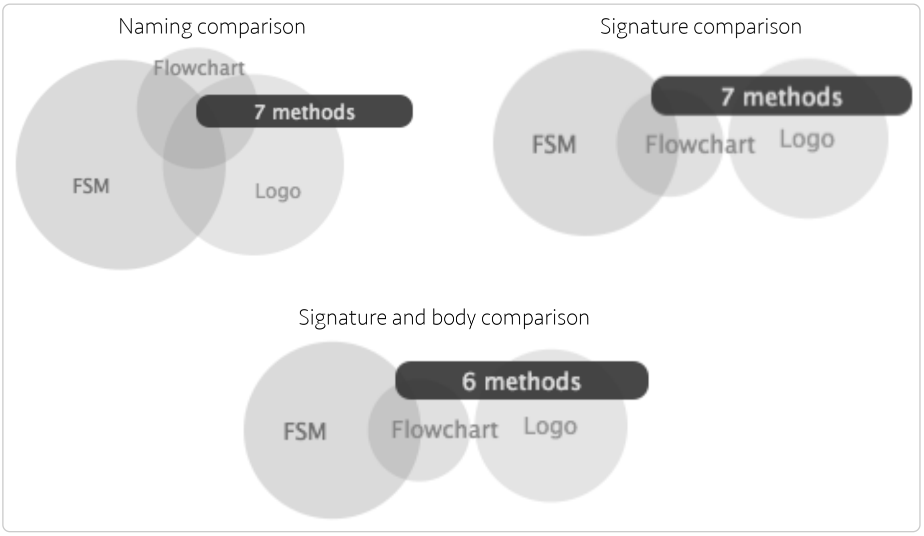

Fig. 3: Visualizing family's shape according to the selected comparison operator

- **Individualization Ratio (IR_i):** This metric shows the percentage of reuse of each DSL with respect the rest of the family. Concretely, it shows for each product the amount of constructs/methods that are included in at least another DSL that is member of the family.

  This metric is important because it allows the identification of the most isolated product as well as the most integrated to the family.
- **Pairwise Relationship Ratio ($PWRR_{(i,j)}$):** This metric shows the percentage of reuse of each DSL with respect the each of the other DSLs that are members of the family. Concretely, it shows, for each product, the amount of constructs/methods that are included each of the other members of the family.

5.1 Visualizing semantic variability

After being evaluated the potential reuse of the family under study we go deeper in the details and we show the variability. In the case of the functional variability, we show a graph that shows us the constructs of all the family, the DSLs they belong to and the relationships they have with other constructs. This is useful since it is a first income to the reverse engineering process. In fact, as state in Y, the first step towards the reverse-engineering process is to break down a language. Besides, the modularization problem can be seen as a graph partitioning problem. Our analysis provides this graph.

In the case of the semantical variability we show the different versions of the DSAs available for each construct.

6 Validation

In this section we present the validation our approach which is performed into two phases. In a first phase, we take a set of DSLs existing in the literature, we implemented it in kermeta, and we perform the corresponding analysis. The idea is to show how our approach behaves in presence of a set of DSLs that we know shares commonalities and where we alredy know the existing semantic variability. A second phase is a more general one. In this case we want to detect potential reuse in existing DSLs that we can find in GitHub repositories. Because kermeta is a quite new benchmark, we there are not many DSLs. However, we can find many metamodels.

6.1 Rhapsody + Harel's Statecharts + UML

An important of this family with respect to the other two is that whereas in the two .. cases the families were built by us by respecting some knowledge of the... this other goes from a different team. The code of this family already exists. This allow us to see the behavior of our tool in a real-life example thus reducing the possible slant that may exist during the construction of the example.

6.2 Identifying potential reuse in DSLs from GitHub

Acknowledgments

The research presented in this paper is supported by the European Union within the FP7 Marie Curie Initial Training Network “RELATE" under grant agreement number 264840 and VaryMDE, a collaboration between Thales and INRIA.

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. B. Combemale, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, June 2014.
4. B. Combemale, J. Deantoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France. Reifying concurrency for executable metamodeling. In M. Erwig, R. F. Paige, and E. Van Wyk, editors, *6th International Conference on Software Language Engineering*, volume 8225 of *SLE*, pages 365–384, Indianapolis, IN, United States, Oct 2013. Springer.
5. 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.

6. 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.
7. J.-M. Favre, D. Gasevic, R. L‰mmel, and E. Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, volume 6563 of *LNCS*, pages 316–326. Springer, 2011.
8. L. Hamann, O. Hofrichter, and M. Gogolla. On integrating structure and behavior modeling with ocl. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 235–251. Springer Berlin Heidelberg, 2012.
9. D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct 2004.
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‰mmel, and E. Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 1–7. Springer Berlin Heidelberg, 2009.
12. J. Liebig, R. Daniel, and S. Apel. Feature-oriented language families: A case study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.
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, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.
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. K.-D. Schewe and J. Zhao. Typed abstract state machines for data-intensive applications. *Knowledge and Information Systems*, 15(3):381–391, 2008.
17. E. Vacchi, W. Cazzola, B. Combemale, and M. Acher. Automating Variability Model Inference for Component-Based Language Implementations. In P. Heymans and J. Rubin, editors, *SPLC'14 - 18th International Software Product Line Conference*, Florence, Italie, Sept. 2014. ACM.
18. E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability support in domain-specific language development. In M. Erwig, R. Paige, and E. Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 76–95. Springer International Publishing, 2013.
19. M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.
20. J. White, J. H. Hill, J. Gray, S. Tambe, A. Gokhale, and D. Schmidt. Improving domain-specific language reuse with software product line techniques. *Software, IEEE*, 26(4):47–53, July 2009.