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. The use of domain-specific languages (DSLs) is 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 technique to achieve separation of concerns in the development of complex systems [8]. 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 [6]. For example, in the literature we can find DSLs for prototyping graphical user interfaces [22], specifying security policies [17], or performing data analysis [10].

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. Besides, typically every software project is unique and requires the construction of tailor-made DSLs for dealing with all its particular aspects [4]. 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...) [13].

To improve cost-benefit when using DSLs, the research community in software languages engineering has proposed mechanisms to increase reuse during the DSLs construction process. The idea is to leverage previous engineering efforts and minimize implementation from scratch [24]. These reuse mechanisms are based on the premise that "software languages are software too" [11] so it is possible to use software engineering techniques to facilitate their construction [14]. 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. Thus, ideas such as Components-Based DSLs Development[5] and Language Product Lines [26] have started to appear.

A classical way for adopting the aforementioned reuse mechanisms is to construct DSLs as building blocks (a.k.a, language modules) that can be later extended and/or imported as part of the specifications of future DSLs. Indeed, there are approaches that support definition of interdependent language modules that can be later composed among them [23,19,15]. However, the definition of language modules that can be useful in future DSLs is not easy. In part, this is due to the fact that, as the same as in the general case of software development processes, a language module is always designed to work in a particular context that, in general, depends on the current project.

A more pragmatical approach to leverage reuse in the construction of DSLs is to focus on legacy DSLs [9]. That is, to exploit reuse in existing DSLs that have been developed independently and without being designed to be reused but that share some commonalities (i.e., they provide similar language constructs). Using this strategy, language modules can be extracted from the commonalities of the DSLs by means of reverse-engineering methods. This type of *a posteriori* reuse permits not only to reduce maintenance cost but also to facilitate the development of new DSLs that can be built from the composition of the resulting language modules.

As the reader may guess, the very first step towards the application of this strategy is to identify potential reuse in a set of existing DSLs. Language designers need to detect commonalities between the DSLs under study and objectively evaluate whether those commonalities represent potential reuse enough to justify the effort associated to the reverse-engineering process. This first step is quite important because reverse-engineering is an expensive process. Language designers must be sure that applying it will actually improve cost-benefit.

In this paper, we present an approach 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 in order to detect commonalities. Besides, our approach 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.

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 since they show that there is a large amount of sets of DSLs that share language constructs and where reuse opportunities are evident. 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: Section 2 introduces a set of preliminary definitions/assumptions that we use all along the paper. Section **??** presents a motivating scenario that illustrates both the problem and the solution tacked in this paper. Section **??** introduces the foundations of our approach. Section **??** 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 Preliminary Definitions

2.1 Domain-Specific Languages in a nutshell

Specification: Like general purpose languages (GPLs), DSLs should be defined in terms of syntax and semantics [12]. Hence, the specification of a DSL is a tuple $< syn, sem >$ where syn (the ***syn****tax*) refers to the structure of the DSL and specifies each language construct in terms of its name and the relationships it has with other language constructs. In turn, sem (the ***sem****antics*) refers to the meaning of the language constructs. This meaning corresponds to the dynamic behavior of each language construct and defines the manner in which they are manipulated at runtime.

Technological space: Currently, there are diverse techniques available for the implementation of syntax and semantics of DSLs [20]. Language designers can choose between using context-free grammars or metamodels as specification formalism for syntax. Similarly, there are at least three methods for expressing semantics: operationally, denotationally, and axiomatically [21]. In this paper we are interested on DSLs which syntax is specified by means of metamodels and semantics is specified operationally as a set methods (a.k.a, *domain-specific actions* [7]). Each language construct is specified by means a metaclass and the relationship between language constructs are specified as references between metaclasses. In turn, each metaclass contains a set of domain-specific actions that correspond to the behavior at runtime.

Implementation: In order to implement a DSL, language designer need a set of tools offering the capabilities to specify the DSL according to the selected technological space. Those tools are provided by language workbenches that offer a set of meta-languages for expressing syntax and semantics. The ideas presented in this paper are implemented in an Eclipse-based language workbench. In particular, metamodels are specified in the Ecore language whereas domain-specific actions are specified as methods in Xtend programming language[1]. The mapping between metaclasses and domain-specific actions is specified by using the notion of aspect introduced by the Kermeta 3 and Melange as explained in [9].

[1] `http://www.eclipse.org/xtend/`

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

By definition, DSLs are scoped to specific domains so they provide restricted set of language constructs. As a result, there is a proliferation of many DSLs in the literature each of which is useful in certain application contexts [20]. Although many of those existing DSLs are completely different and tackle independent domains; there are related DSLs with overlapping domains [25, p. 60-61]. That is, they share certain language constructs i.e., they have **commonalities** between them. If two DSLs have commonalities and they are specified in the same technological space, then there is **potential reuse** since the specification of those shared constructs can be specified once and reused in the two DSLs [25, p. 60-61].

Naturally, commonalities can be found not only at the level of the syntax but also at the level of the semantics. For the technological space discussed in this paper, syntactic commonalities appear where DSLs share some metaclasses and semantic commonalities appear where DSLs share some domain-specific actions. Figure 1 illustrates this situation. There are three DSLs (i.e., A, B, and C) with both syntactic and semantic commonalities. Syntactic commonalities (at the left of the figure) correspond to the metaclasses MC_Z and MC_Q. MC_Z is shared by DSLs A and C whereas MC_Q is shared by DSLs A, B, and C. Semantic commonalities (at the right of the figure) correspond to the domain-specific action DSA_3 which is shared by all the DSLs.

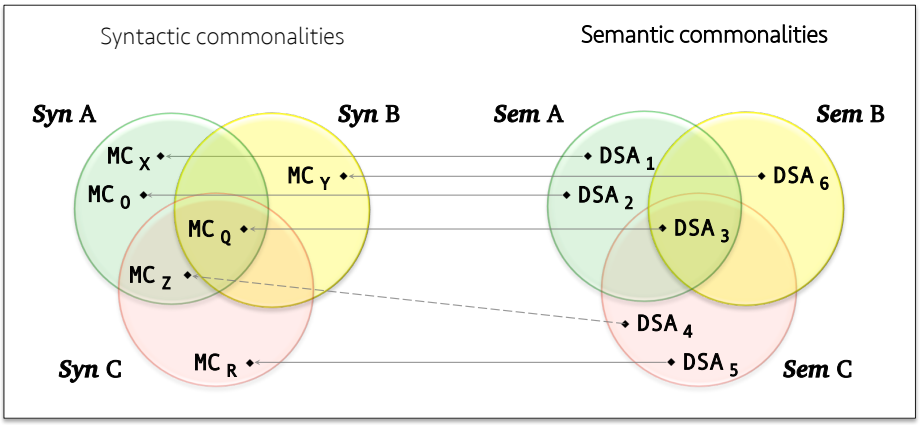

Fig. 1: Commonalities between domains and potential reuse

Commonalities can be found between two ore more DSLs of the input set. That is, we can find metaclasses and domain specific actions that are shared by more than two DSLs. Hence, intersections should be searched among all the possible combinations of the DSLs in the input set. Once those functions are defined and implemented, the second phase is to use them in order to find the intersections among the DSLs of the input set.

Semantical variability: Note that the fact that two metaclasses are shared does not imply that all their domain specific actions are the same. In that figure, we see this in the case of MMz. This metaclass is shared by the DSLs A and C but, in each DSL the semantics is different since each of them implement a different DSA for the metaclass. We refer to that phenomenon as *semantical variability*. There are two constructs that share the syntax but that differ in their semantics. In such case, there is potential reuse at the level of the syntax since the metaclass can be defined once and reused in the DSLs but the semantics should be defined differently for each DSLs.

3 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 evaluate those commonalities in order to know if the input set is a good candidate to a reverse engineering method that permits to exploit the existing potential reuse. The reminder of this section explain how we tackle this problem.

3.1 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 such as the presented in the previous section (Figure 1). We consider these diagrams as a useful mechanism that allows language designers to easily visually identify commonalities. 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. We do the proper for the case of the domain-specific actions.

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. As mentioned before, 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.

Comparison operators: A syntactic intersection is a set of metaclasses that are equal in two or more DSLs. Similarly, a semantic intersection is a set of domain-specific actions that are equal in two or more DSLs. At this point we need to clearly define the notion of equality between metaclasses and domain-specific actions. That is, we need to establish the criteria under we consider that two metaclasses/domain-specific actions are equal.

– **Comparison of metaclasses:** The name of a metaclass usually corresponds to a word that evokes the domain concept the metaclass represents. Thus, intuitively one can think that a first approach to compare meta-classes is by comparing their names. As we will see later in this paper, this approach results quite useful and it is quite probable that, we can find potential reuse.

$$MC_A \doteq MC_B = true \implies MC_A.name = MC_B.name \tag{1}$$

Unfortunately, comparison of metaclasses 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 since they do not represent the same domain concept or because there are domains that use similar vocabulary. 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. Hence, we define a second comparison operator for metaclasses i.e., $\doteqdot$.

$$\begin{aligned} MC_A \doteqdot MC_B = true \implies & \\ & MC_A \doteq= MC_B \wedge \\ & \forall a_1 \in MC_A.attr \mid (\exists a_2 \in MC_B.attr \mid a_1 = a_2) \wedge \\ & \forall r_1 \in MC_A.refs \mid (\exists r_2 \in MC_B.refs \mid r_1 = r_2) \end{aligned} \tag{2}$$

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. At the implementation we provide support for the two comparison approaches explained above. However, additional comparison operators such as the surveyed in [16] can be easily incorporated.

– **Comparing domain-specific actions:** Like methods in Java, domain-specific actions have a signature that specifies its contract (i.e., return type, visibility, parameters, name, and so on), and a body where the behavior is actually implemented. In that sense, 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.

$$\begin{aligned} DSA_A \mathbin{\hat{=}} DSA_B = true \implies & \\ & DSA_A.name = DSA_B.name \wedge \\ & DSA_A.returnType = DSA_B.returnType \wedge \\ & DSA_A.visibility = DSA_B.visibility \wedge \\ & \forall p_1 \in DSA_A.params \mid (\exists p_2 \in DSA_B.params \mid p_1 = p_2) \end{aligned} \tag{3}$$

However, as the reader might imagine, there are cases in which signatures comparison is not enough. Two domain-specific actions defined in different DSLs can perform different computations even if they have the same signatures. As a result, a second approach relies in the comparison of the bodies of the domain-specific actions. Note that such comparison can be arbitrary difficult. Indeed, if we try to compare the behavior of the actions we will have to deal with the semantic equivalence problem that, indeed, is known as be undecidable [18]. In this case, we a conservative approach is to compare only the structure (abstract syntax tree) body of the domain-specific action. To this end, we use the API for java code comparison proposed in [3].

$$\begin{aligned} DSA_A \triangleq DSA_B = true \implies \\ DSA_A \triangleq DSA_B \wedge \\ DSA_A.AST = DSA_B.AST \end{aligned} \tag{4}$$

Visualizing the results: Figure 2 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.

3.2 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 [1,2] and we adapt them for the case of domain-specific languages. Those metrics (graphically shown in Figure 3) are intended to measure the size of the commonalities existing among the DSLs of the input set. In this section we present these metrics in terms of the formulas we used to compute them. It is important to remember that the results provided by those metrics also depend on the comparison operator.

- **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.

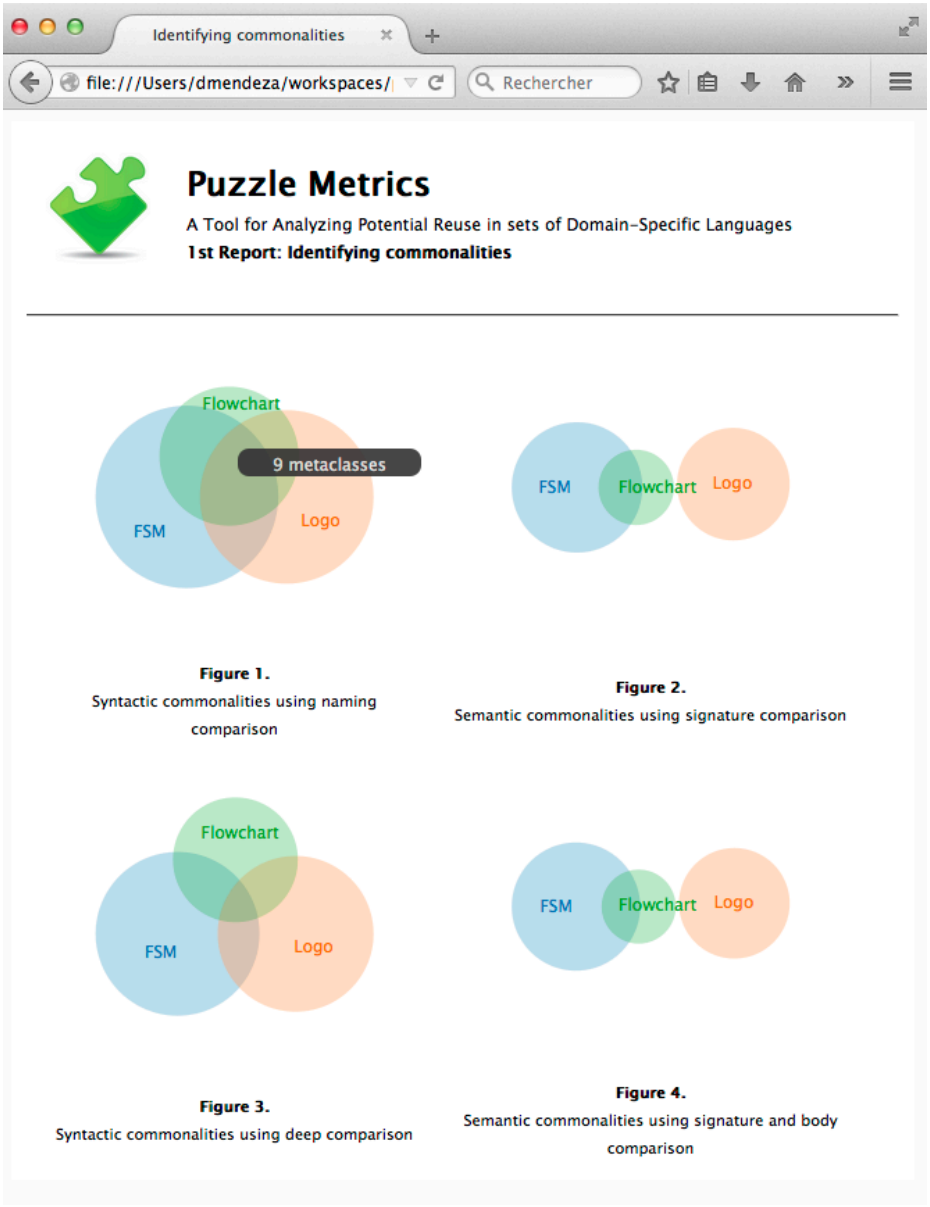

Fig. 2: Visualizing family's shape according to the selected comparison operator

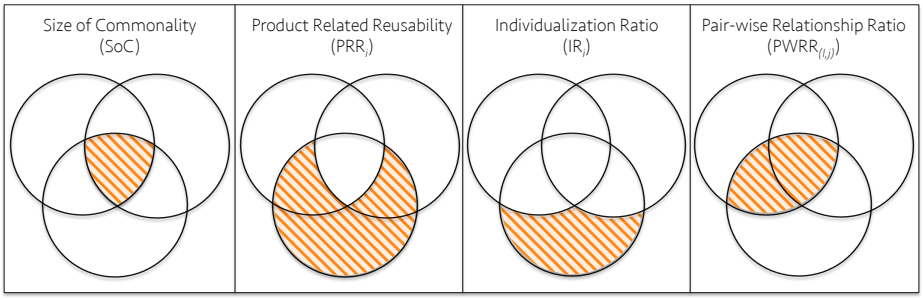

Fig. 3: Metrics for evaluation of potential reuse

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.

- **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.

Tool support. Figure x shows the output of our tool. In this case, it receives a set of DSLs and it computes all the metrics explained above.

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. Biegel and S. Diehl. Jccd: A flexible and extensible api for implementing custom code clone detectors. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 167–168, New York, NY, USA, 2010. ACM.
4. 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.
5. 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.
6. B. Combemale, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, June 2014.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct 2004.
13. 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.
14. 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.
15. 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.
16. L. Lafi, S. Hammoudi, and J. Feki. Metamodel matching techniques in mda: Challenge, issues and comparison. In L. Bellatreche and F. Mota Pinto, editors, *Model and Data Engineering*, volume 6918 of *Lecture Notes in Computer Science*, pages 278–286. Springer Berlin Heidelberg, 2011.
17. 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.

18. D. Lucanu and V. Rusu. Program equivalence by circular reasoning. In E. Johnsen and L. Petre, editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 362–377. Springer Berlin Heidelberg, 2013.
19. M. Mernik. An object-oriented approach to language compositions for software language engineering. *J. Syst. Softw.*, 86(9):2451–2464, Sept. 2013.
20. 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.
21. P. D. Mosses. The varieties of programming language semantics and their uses. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 165–190. Springer Berlin Heidelberg, 2001.
22. 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.
23. E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.
24. 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.
25. 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.
26. 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.