# 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 [21], 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. 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 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 deal with such complexity, the research community in Software Languages Engineering (SLE) has proposed mechanisms to increase reuse during 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" [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 [25] 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. For example, there are approaches that exploit the notion of genericity in DSLs [22], as well as approaches that support definition and composition of interdependent language modules [23,19,15]. The success of this strategy relies on a set of design decisions that favor extensibility and/or genericity thus increasing the probabilities that the language modules are useful in the future. In this case, the major complexity comes from the fact that language designers do not know *a priori* the needs of future DSLs. Consequently, in practice many of these building blocks are not reusable *as is* and rather they require some previous adaptation.

An alternative strategy is to focus on legacy DSLs. That is, to exploit reuse in existing DSLs that have been developed independently and without being designed to be reused [9]. This strategy is quite useful when there are DSLs that share some commonalities (i.e., they provide similar language constructs) that can be encapsulated in independent language modules 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. It is worth highlighting, however, that the very first step towards the application of this strategy is to identify potential reuse. In other words, language designers need to detect sets of DSLs that share commonalities and they have to be sure that these commonalities are enough to justify the effort associated to the reverse-engineering process.

In this paper, we present 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 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 on the DSLs that permit to objectively evaluate if the existing 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 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

As aforementioned, a DSL is a set of language constructs each of which represents certain abstraction in a particular domain. In the general case, a DSL offers an editor that provides the capabilities needed to write programs (textually or graphically) by using the language constructs. Once a the program is written, it can be used as part of the implementation artifacts of a system under construction.

Like general purpose languages (GPLs), DSLs are specified in terms of three interdependent dimensions: abstract syntax, concrete syntax, and semantics [12]. The *abstract syntax* 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 of the DSL. The *concrete syntax* refers to the association of the language constructs to the set of symbols (either graphical or textual) offered by the editor. Finally, the *semantics* of a DSL refers to the meaning of the language constructs and it is expressed through static constraints and dynamic behavior. The static constraints usually correspond to the type system of the DSL. In turn, the dynamic behavior defines the manner in which language constructs are manipulated at runtime, typically through the definition of an interpreter or compiler.

**Scope:** In this paper we are interested on DSLs which abstract syntax is specified by means of metamodels and semantics is specified operationally as a set methods (a.k.a, *domain-specific actions* [7]) on the metaclasses of the metamodel. In other words, each language construct is specified as a metaclass and the relationship between language constructs are specified as references between metaclasses. In turn, each metaclass contains a set of methods that correspond to the behavior in runtime. The concrete syntax is out of the scope of this paper.

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

Due to the complexity of current systems, there is a large amount of concerns to deal with. In addition, by definition the domain of a DSL is scoped to a specific aspect of a system. As a result, there is a proliferation of many DSLs in the literature [20]. Although many of those existing DSLs are completely different and tackle independent domains; there are related DSLs with overlapping domains. That is, they share certain language constructs. When two DSLs share language constructs, we say that there is **potential reuse** since the specification of those shared constructs can be reused in the two DSLs [24, p. 60-61].

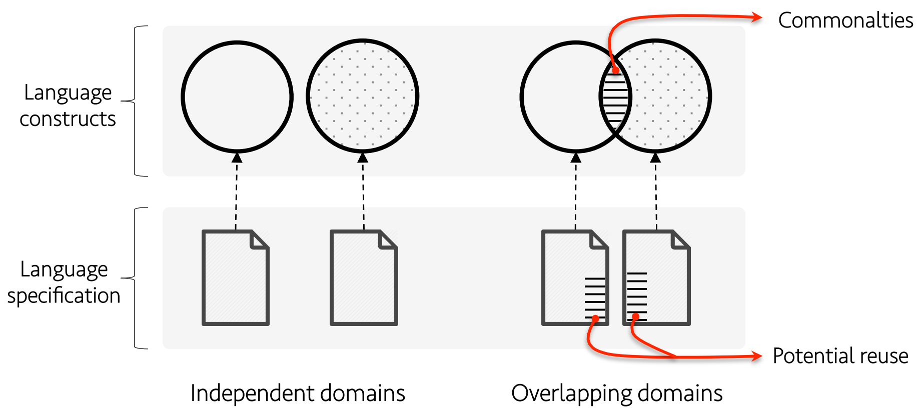

Fig. 1: Commonalities between domains and potential reuse

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 language constructs, and consequently, there is not potential reuse between them. Differently, the two DSLs shown at the right of the figure have overlapping domains. That means that there are a subset of language that are **"equal"** in both DSLs. Note that if two language constructs are the same, we can assume that their specifications are equal and can be reused instead of being replicated.

### 2.3 Equivalence between language constructs

So far, we have based the notion of potential reuse in DSLs on the commonalities existing in a set of DSLs. Nevertheless, this assumption supposes that we are able to compare two language constructs in order to know if they are equivalent. So, now we need to define this *equivalence* relationship. In particular, 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.

**Comparing meta-classes.** 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 name. As we will see later in this paper, this approach results quite useful and it is quite probable that, we can find potential reuse.

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. 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.

In the approach present in this paper, 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 where the contract is specified (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.

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 complex task. 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 onlythe structure (abstract syntax tree) body of the domain-specific action.

In our approach we support both comparison operators: the one based on the signature and the one based on the signature and the body. To the later, we use the API for java code comparison proposed in [3].

**Semantical variability.** A necessary condition to decide whether two language constructs are equivalent is that both, the metaclass and the associated domain-specific actions are equivalent. This condition 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 [4]. There is semantical variability when there there are two constructs that have the same abstract syntax (i.e., their metaclasses are equal) but that 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 metaclasses of two constructs are the same we can reuse them even if their domain-specific actions are different.

## 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. M. V. Cengarle, H. Grönniger, and B. Rumpe. Variability within modeling language definitions. In A. Sch¸rr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 670–684. Springer Berlin Heidelberg, 2009.
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. 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.
22. L. Rose, E. Guerra, J. de Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software & Systems Modeling*, 12(1):201–219, 2013.
23. E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.
24. 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.
25. 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.