# Reverse Engineering Language Product Lines

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

University of Rennes 1, INRIA/IRISA. France
{david.mendez-acuna,jagalindo,benoit.combemale,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 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 [4]. 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 [3]. For example, in the literature we can find DSLs for prototyping graphical user interfaces [13], specifying security policies [11], 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 [1]. 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...) [8].

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 [15]. 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 [9]. In particular, there are approaches that take ideas from Component-Based Software Engineering (CBSE) [2] and Software Product Lines Engineering (SPLE) [16] 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 [14,12,10]. 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. What is the correct level of granularity? Are there constructs that should be always together? Are there constructs that should be always separated? In addition, the construction of a variability model is quite challenging as well. It requires domain knowledge.

A more pragmatical approach to leverage reuse in the construction of DSLs is to focus on legacy DSLs [5]. 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.

In this paper, we present an approach to reverse engineering a language product line from a given set of DSLs. To to so, we first identify and extract reusable language modules. Then, we infer a variability model that represents the variability existing in the set of DSLs. This variability model can be used for configuring new DSLs.

This paper is organized as follows: Section **??** introduces a set of preliminary definitions/assumptions as well as some motivating examples that we use all along the paper. Section **??** describes our approach. Section **??** presents the empirical study that validates the approach. Section **??** discusses the threads to validity. Section **??** presents the related work and, finally, Section **??** concludes the paper.

## 2 Problem statement

## 3 Proposed approach

### 3.1 Overview

### 3.2 Breaking down the DSLs

The objective of this step is to divide the DSL in several language modules that can be latter composed. To do so, we need to deal with two problems. The former is to find the way in which the we will split the language constructs. The second one is to define the language modules themselves with their corresponding required and provided interfaces.

**Identifying constructs distribution** The input is the set of DSLs. We first identify the language constructs for each one. Then we perform a match according to a given comparison operator. Then, we merge. Finally, we execute a graph partitioning algorithm.

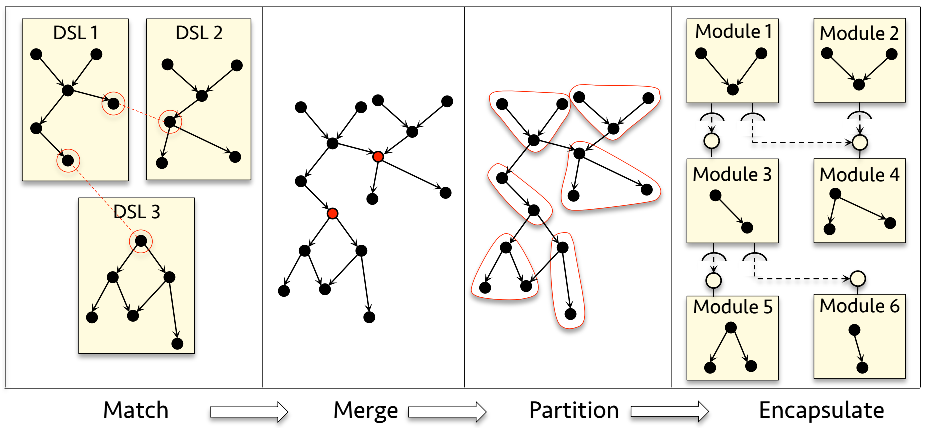

Fig. 1: Process for breaking down a set of DSLs

**Specifying language modules** Note that both DSLs and modules are, at the end, set of language constructs. Hence, one may think that a module is also a DSL. Although this is technically true, there is a substantial difference between DSLs and modules. A DSL is a closed set of constructs that materializes a complete DSL specification that is ready to be used. Contrariwise, a module is set of constructs whose specification may depend on other constructs defined in other modules. A module can be used (and considered itself as a DSL) as long as their dependencies are fulfilled.

Accordingly, the main requirement for supporting separation of features in DSLs relies on the capability of expressing dependencies between language modules. In this context, we have identified two types of dependencies: *aggregation* and *extension*. In the following, we explain each of them and we present the corresponding tool support.

**Language modules *aggregation*:** In aggregation, there is a *requiring module* that uses some constructs provided by a *providing module*. The requiring module has a dependency relationship towards the providing one that, in the small, is materialized by the fact that some of the classes of the requiring module have references (simple references or containment references) to some constructs of the providing one.

In order to avoid direct references between modules, we introduce the notion of interfaces for dealing with modules' dependencies. In the case of aggregation, the requiring language has a *required interface* whereas the providing one has the *provided interface*. A required interface contains the set of constructs required by the requiring module which are supposed to be replaced by actual construct provided by other module(s).

It is important to highlight that we use *model types* [**?**] to express both required and provided interfaces. As illustrated on top of Figure 2, the relationship between a module and its required interface is *referencing*. A module can have some references to the constructs declared in its required interface. In turn, the relationship between a module and its provided interface is *implements* (deeply explained in [**?**]). A module implements the functionality exposed in its model type. If the required interface is a subtype of the provided interface, then the provided interface fulfills the requirements declared in a required interface. Note that the partial sub-typing relationship defined in [**?**], permits a required interface being partially fulfilled by a provided interface. The result of the composition will be a module with a new required interface that contains only those elements that were not provided.

**Language modules *extension*:** In this case, there is an extension module that (naturally) extends the functionality provided by *base module*. The extension module has a dependency to the base module. Moreover, the extension module has little sense by itself without the existence of a base module [**?**]. Note that this is a conceptual difference with respect to modules aggregation where the required make sense by itself but requires some external services in order to work correctly.

There are to different mechanisms for extending a base module: *constructs specialization* and *open classes* [**?**]. In constructs specialization, the constructs of the base module can be extended by adding new subclasses. The extension module contains the new subclasses that reference (by means of the inheritance relationship) the constructs of the base module that are being extended. In this case, the base module remains intact in after the composition but there are additional constructs. In open classes, constructs of the base module can be reopened and modified by the extension module. For example, for adding a new attribute to a given construct without creating a sub-class, or for overriding a given segment of the semantics. In this case, the extension module is altered after the composition phase.

Similarly to aggregation, dependencies between the base and the extension modules are specified through interfaces. The base module exposes an *extension point interface* with the constructs that can be extended. In turn, the *extension interface* declares the constructs of the base module that are being extended.

Like in aggregation, and as illustrated at the bottom of Figure 2, we use model types of expressing these interfaces. Although the approach is quite similar, there is one fundamental difference between the interfaces in aggregation and the interfaces in extension: the relationship between an extension module and its extension interface is *usage* more than just referencing. That means that the module can to reference the elements declared in the required interface and also modify them by adding new elements. This capability is introduced to support extension by the open-classes mechanism.

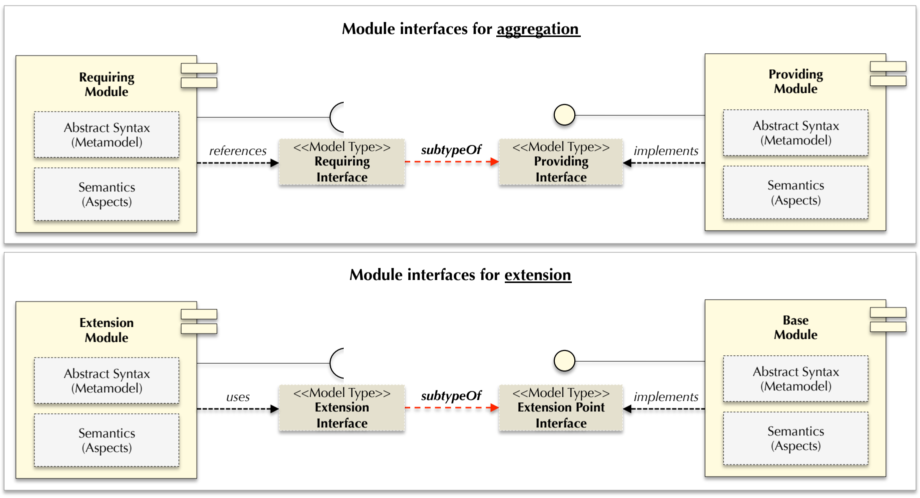
Module interfaces for aggregation
Requiring Module
Abstract Syntax (Metamodel)
Semantics (Aspects)
references
<<Model Type>> Requiring Interface
subtypeOf
<<Model Type>> Providing Interface
implements
Providing Module
Abstract Syntax (Metamodel)
Semantics (Aspects)
Module interfaces for extension
Extension Module
Abstract Syntax (Metamodel)
Semantics (Aspects)
uses
<<Model Type>> Extension Interface
subtypeOf
<<Model Type>> Extension Point Interface
implements
Base Module
Abstract Syntax (Metamodel)
Semantics (Aspects)

Fig. 2: Interfaces for modularization of DSLs

## 3.3 Inferring the variability model

After having a set of language modules with their corresponding references among them, we need to automatically infer a variability model that represents the existing variability. To do so, we use as input an algorithm that, based on the dependencies graph of the modules, infer a simple variability model.

## 3.4 Deriving a DSL

Once the variability of the language product line is correctly specified, the next step is to configure DSLs by using the variability model. Since the variability model is expressed in CVL, the configuration of the language product corresponds to the specification of a realization model that captures the decisions made by the language designers that are configuring the DSL. Our approach uses the realization model to produce the corresponding Melange script.

As an example, consider the configuration presented in the variability model of the Figure **??**. The corresponding Melange script is presented in the following listing code snippet:

```
language ModuleA {
    ecore MetamodelA.ecore
    with package.A.Semantics6
}

language ModuleB {
    ecore MetamodelB.ecore
    with package.B.Semantics4
}

language MyDSL {
    aggregation(ModuleB, ModuleA)
}
```

Note that the Melange script only contains the language elements that correspond to a given configuration. For example, the ModuleA contains only the Semantics6 because it was the choice made at configuration time. Similarly, ModuleB only contains Semantics4. Note also that there is a third language appearing in the script: MyDSL. This language represents the composition of the language modules and can be understood as the root of the script. In this case, this statement of Melange indicates that the modules A and B are composed by aggregation. The first element in the operation corresponds to the requiring module and the second element corresponds to the providing module.

Once the configuration process produces a Melange script that captures the choices made by the language designers for a particular DSL, it is necessary to compose the declared language modules and produce the DSL. The composition of a set of modules requires a previous phase of compatibility checking. Not all language modules are compatible and in that case composition cannot be performed. In our approach, check the compatibility of two language modules is to verify the sub-typing relationship between the required and provided interface (for the case of aggregation), and the extension and extension point interfaces (for the case of extension).

Once this compatibility checking is correctly verified, language modules are composed. In particular, their specifications are be merged to generate a complete language specification. This merging basically replaces the elements of the required interface by its corresponding implementation in the provided component. A similar process is performed in the case of extension.

## 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. 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.
2. 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.
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. 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.
5. 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.
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. 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.
9. 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.
10. 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.
11. 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.
12. M. Mernik. An object-oriented approach to language compositions for software language engineering. *J. Syst. Softw.*, 86(9):2451–2464, Sept. 2013.
13. 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.

14. E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.
15. 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.
16. 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.