Identifying Reusable Language Modules from Existing 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. Indeed, we can find several approaches that support modular definition of DSLs. Still, the design of language modules that can be actually reusable is a difficult task. With little information, language designers have to define language modules intended to be reused in future contexts that, of course, they do not know in advance. In this paper, we explore a different approach by presenting a strategy to identify reusable language modules from existing DSLs. Our idea is to detect sets of language constructs that are tipically used together and extract them in interdependent language modules. We validate our approach by using realistic DSLs that we obtain 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 [7]. 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 [21], specifying security policies [16], or performing data analysis [9].

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 matter of fact, the ultimate value of DSLs has been severely limited by the cost of the associated tooling (i.e., editors, parsers, etc...) [12].

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 [23]. These reuse mechanisms are based on the premise that "software languages are software too" [10] so it is possible to use software engineering techniques to facilitate their construction [13]. For instance, there are approaches that take ideas from Component-Based Software Engineering (CBSE) [4] and Software Product Lines Engineering (SPLE) [25] during the construction of new DSLs.

The basic principle underlying the aforementioned reuse mechanisms is that language constructs are grouped into interdependent *language modules* that can be later integrated as part of the specification of future DSLs. Current approaches for modular development of DSLs (e.g., [22,18,14]) are focused on providing foundations and tooling that allow language designers to explicitly specify dependencies among language modules as well as to provide the composition operators needed during the subsequent assembly process.

Despite such important advances, the definition of language modules that can be actually useful in future DSLs is still a difficult task. This difficulty is due to several factors. For example, the reuse of a language module implies the reuse of all the constructs it offers; in many cases, some of those constructs are not necessary and, worst, they might be even conflictive. Language modules rarely can be reused 'as-is' and without any adaptation. In this context, our research question is: *How to define language modules that can be actually reusable in the construction of future DSLs?*

Inspired in the work of Caldiera and Basili [2], in this paper we propose a tool-supported approach that analyses a set of existing DSLs to extract a catalog of reusable language modules. The main idea is to perform static analysis on a given set of DSLs that are implemented in an homogeneous technology in order to detect groups of constructs that are typically used together. Once those groups are identified, we extract them by means of a break-down algorithm. Our approach considers not only syntax but also semantics of the DSLs under study. We validate our approach by applying it in an evaluation scenario with realistic DSLs that we obtain from public GitHub repositories. The results of this validation are quite promising.

The reminder of this paper is organized as follows: Section 2 introduces a set of preliminary definitions/assumptions as well as an illustrating scenario that we use all along the paper. Section 3 describes our approach that is evaluated in Section 4. Section **??** presents the related work and, finally, Section **??** concludes the paper.

2 Preliminary Definitions

2.1 Domain-Specific Languages in a nutshell

Specification: Like general purpose languages (GPLs), DSLs are defined in terms of syntax and semantics [11]. Hence, the specification of a DSL is a tuple $< syn, sem, M_{syn \leftarrow sem} >$ [6]. The parameter *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, the parameter *sem* (the ***sem**antics*) refers to the meaning of the language constructs. This meaning corresponds to the dynamic behavior that establishes the manner in which language constructs are manipulated at runtime. Finally, the parameter $M_{syn \leftarrow sem}$ refers to the mapping between the language constructs and the semantics.

Technological space: Currently, there are diverse techniques available for the implementation of syntax and semantics of DSLs [19]. Language designers can, for example, 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 [20]. 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* [6]). Each language construct is specified by means a metaclass and the relationship between language constructs are specified as references between metaclasses. In turn, domain-specific actions are specified as java-like methods that are allocated in each metaclass.

Implementation: In order to implement a DSL, language designers need a tool set that offer capabilities to specify a DSL according to the selected technological space. This kind of tool sets are provided by language workbenches (such as Eclipse Modeling Framework or MetaEdit+) that provide meta-languages for where syntax and semantics can be expressed. 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[2] and Melange[3] as explained in [8].

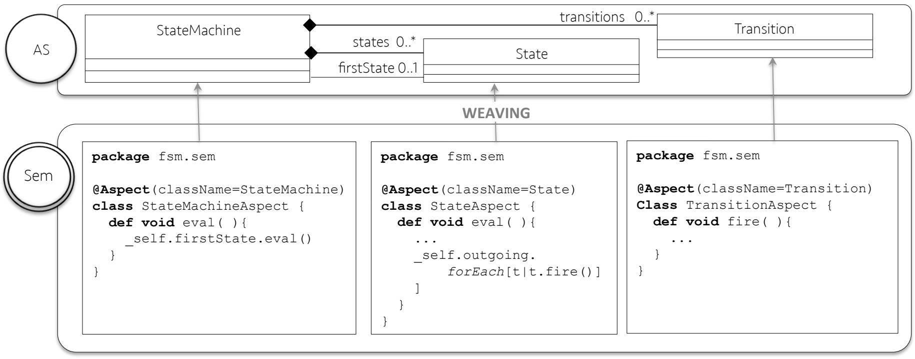
A simple DSL: Let us illustrate this idea by using a simple example. Consider the metamodel introduced at the top of Figure 1. It is a metamodel for a simple language for finite state machines. It contains thee classes `StateMachine`, `State`, and `Transition`; The class `StateMachine` contains both states and transitions which is represented with containment references. In turn, the code snippets at

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

[2] `https://github.com/diverse-project/k3/wiki/Defining-aspects-in-Kermeta-3`

[3] `http://melange-lang.org/`

the bottom of Figure 1 introduce some operational semantics to this metamodel by using K3. Note that the main feature of K3 is the notion of aspect that permits to weave the operational semantics defined in a Xtend class to a metamodel defined in Ecore. In our example, the metaclass `StateMachine` is enriched with the operation `eval()` that contains a loop that sequentially invokes the operation defined for the class `State`. This operation is also defined by using one aspect. The metaclass `Transition` is enriched with the operation `fire()`.



```
package fsm.sem

@Aspect(className=StateMachine)
class StateMachineAspect {
  def void eval( ){
    _self.firstState.eval()
  }
}
```

```
package fsm.sem

@Aspect(className=State)
class StateAspect {
  def void eval( ){
    ...
    _self.outgoing.
        forEach[t|t.fire()]
    ]
  }
}
```

```
package fsm.sem

@Aspect(className=Transition)
Class TransitionAspect {
  def void fire( ){
    ...
  }
}
```

Fig. 1: A simple FSM language

As said above, we use Melange to integrate and execute the definitions of the abstract syntax and semantics of a DSL. Melange is a language for modeling in the large that facilitates the integration of the different artifacts that compose the specification of a DSL. Listing 1.1 illustrates the use of Melange. At the left we have an abstract representation of a language that is composed of a metamodel and three aspects implementing the semantics. At the right of the figure we have the corresponding Melange script.

```
language FSM {
    syntax "fsm.mm/models/fsm.ecore"

    with fsm.sem.StateMachineAspect
    with fsm.sem.StateAspect
    with fsm.sem.TransitionAspect
}
```

Listing 1.1: Melange script for a simple FSM language

2.2 Overlapping in DSLs

In [24, p. 60-61], Vöelter et al that observes that although many of the existing DSLs are completely different and tackle independent domains; there are related

DSLs with overlapping domains. 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 and using compatible language workbenches, then there is **potential reuse** since the specification of those shared constructs can be specified once and reused in the two DSLs [24, 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.

2.3 Illustrating Scenario

Let us now introduce a toy example to illustrate the concepts introduced so far. Consider the following a set of three DSLs that contains the language for finite state machines introduced before, and two additional ones; a DSL for Logo and another one for Flowchart. Logo is a DSL for expressing movements of the classical Logo turtle used in elementary schools for teaching the first foundations of programming. This DSL offers the constructs for moving a turtle forward and backward, as well for rotating the turtle at the left or at the right. In addition, the DSL offers simple arithmetic expressions for indicating the distance/angles the must should move/rotate. In turn, Flowchart is a DSL for expressing simple flow diagrams. Each flow is a sequence of nodes and arcs between them. An arc can be either an action or a decision. An action is a set of instructions that modify some variables in the execution context. A decision is a bifurcation point where depending on a given condition the flow goes from a direction or another.

As a matter of fact, these DSLs are essentially different. Each of them is focus on a particular domain and offers different language constructs. However, there are syntactic and semantic commonalities that are illustrated in Figure 2. All the thee DSLs offer some expressions for modifying variables. In the case of FSM these actions are needed the specification of the actions in the states; in the case of Logo expressions are needed to specity the movement and rotation parameters; and in the case of flowchart expressions are needed to specify the body of actions. In addition, both FSM and Flowchart rely on a constraints language. The former for expressing guards in the transitions and the later for expressing guards of the decisions.

Note that each DSL is specified in terms of a set of metaclasses (top of the figure), and a set of aspects (bottom of the figure) that weave some domain-specific actions to the metaclasses. In the case of this example, the semantics of the metaclasses expression and constraints are also shared. That means that the semantics are the same.

It is worth to mention that the fact that two metaclasses are shared does not imply that all their domain specific actions are the same. 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

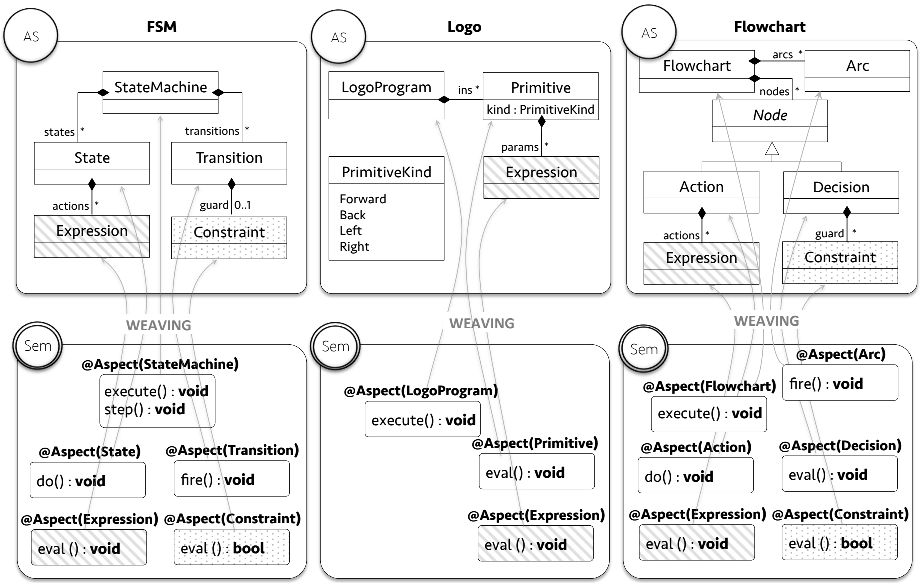


Fig. 2: Commonalities between domains and potential reuse

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

3.1 Identifying overlapping

Given a set of existing DSLs (that we term as the *input set*) our approach is intended to identify a catalog of reusable language modules. To that end, we detect groups of language constructs that usually appear together. Our hypothesis is that if a set of constructs is usually used together is because there can be some strong domain relationships between the concepts they represent.

To detect language modules, we first perform analysis analysis on the input DSLs and we find overlapping among them. Our analysis can be illustrated by means of a Venn Diagram such as the one presented in Figure 3 that uses our illustrating scenario and includes both syntax and semantic overlapping. Syntactic and semantic overlapping is represented as intersections between the corresponding sets. To this end, we designed an algorithm that is able to compute the all overlapping among the syntax of the DSLs in the input set.

Our algorithm for detecting **syntactic overlapping** 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 overlapping among these metamodels. Note that there can be overlapping 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 overlapping** 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 overlapping among these aspects.

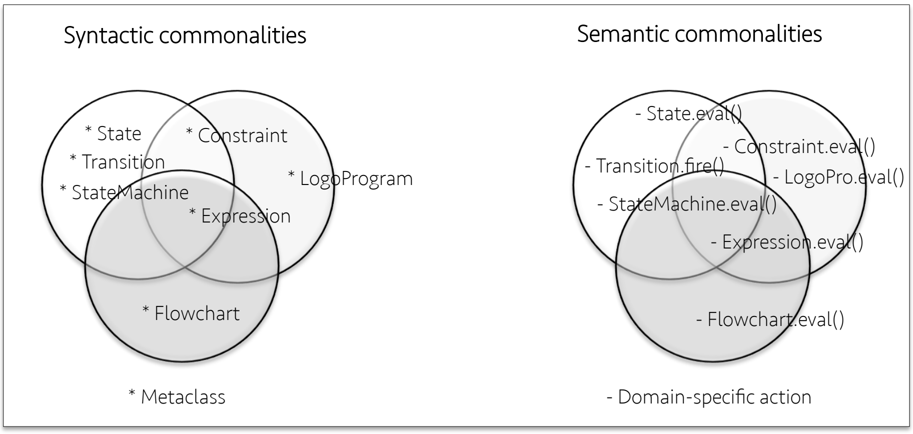


Fig. 3: Visualizing syntactic and semantic commonalities

Comparison operators: A syntactic overlapping is a set of metaclasses that are equal in two or more DSLs. Similarly, a semantic overlapping 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.

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

  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 [15] 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.

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

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 [17]. 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 [1].

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

3.2 Extracting reusable language modules

Breaking down the input set: After being identifying commonalities among the input set, we extract a set of reusable language modules. To do so, we focus the attention on analyzing overlapping among the sets. Each overlapping contains a set of metaclasses/domain specific actions that are used together in a given set of DSLs. In the illustrating scenario we can see that the overlapping among Logo, FSM, and Flowchart is a set of language constructs that permit to express expressions. They go well together and it makes sense to group them. Similarly, the overlapping between Flowchart and FSM is a set of constructs about constraints. Note that this example show us that it is not appropriated to put constraints and expressions in one language module. In that case, Logo would include constraints that is not necessary.

From that observation, we propose to create one module for each overlapping as presented in Figure 4. For each different overlapping, we create a language module. Of course, there are certain dependencies among modules that we need to express. Besides, breaking down DSLs in language modules only makes sense if we can later compose those modules to obtain complete DSLs. Then, a composition strategy is needed.

Encapsulating language modules: Let us now, to explain how we support the capability of encapsulating subsets of language constructs in separated and interdependent language modules. Different modules specify different constructs; a complete DSL is obtained by composing a set of modules. Accordingly, the main requirement for supporting separation of features in DSLs relies on the capability of expressing dependencies between language modules. In the following, we explain each of them and we present the corresponding tool support.

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 5, the relationship

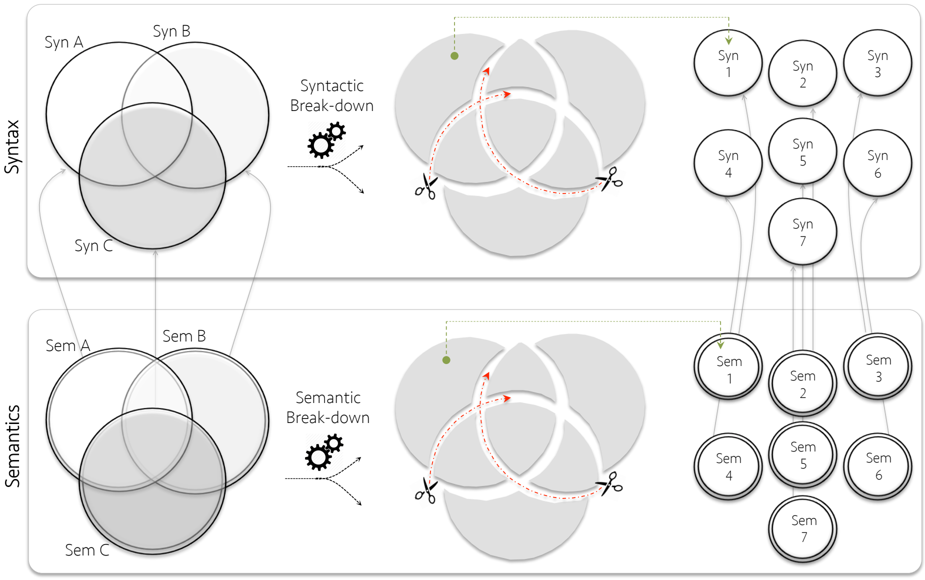


Fig. 4: Breaking down the input set by separating overlapping

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.

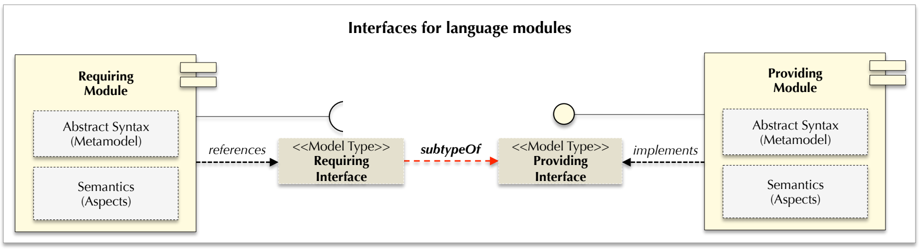

Interfaces for language modules
Requiring Module
Abstract Syntax (Metamodel)
Semantics (Aspects)
references
<<Model Type>> Requiring Interface
subtypeOf
<<Model Type>> Providing Interface
implements
Providing Module
Abstract Syntax (Metamodel)
Semantics (Aspects)

Fig. 5: Interfaces for language modules

The tool support we provide for specifying interfaces on language modules is based on annotations at the level of the abstract syntax (i.e., the metamodel). That means that the required constructs of a requiring module are declared as meta-classes in the metamodel as the same as the actual constructs of the module. However, the required constructs are annotated with `@Required` to distinguish them from the actual constructs.

4 Evaluation

In this section we evaluate our approach by using two different evaluation scenarios. The former corresponds to a case study of a set of DSLs for expressing state machines. The idea is to test our approach in a set of language where we know that there are commonalities that we know in advance. The second case study aims to test our approach in a more realistic scenario. We take some DSLs from GitHub public repositories.

4.1 Case study 1: State Machines

4.2 Case study 2: Seeking GitHub repositories

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. 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.
2. G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, Feb. 1991.
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. B. Combemale, C. Hardebolle, C. Jacquet, F. Boulanger, and B. Baudry. Bridging the chasm between executable metamodeling and models of computation. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 184–203. Springer Berlin Heidelberg, 2013.
7. 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.

8. 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.
9. 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.
10. 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.
11. D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct 2004.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. M. Mernik. An object-oriented approach to language compositions for software language engineering. *J. Syst. Softw.*, 86(9):2451–2464, Sept. 2013.
19. 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.
20. 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.
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. E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.
23. 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.

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.