Identifying Reusable Language Modules from Existing Domain-Specific Languages

No Author Given

No Institute Given

Abstract. The use of domain-specific languages (DSLs) has become a successful technique in the implementation of complex systems. Still, the construction of this type of languages is time-consuming and requires highly-specialized knowledge and skills. An emerging practice to facilitate this task is to enable reuse by means of the definition of language modules that can be later put together to build up new DSLs. However, the identification and definition of those modules is complex and error-prone thus, hindering the reuse exploitation when developing DSLs. In this paper, we propose a computer-aided approach to i) identify potential reuse in a set of DSLs; and ii) break down those DSLs into language modules that can be later exploited to minimize costs when constructing new languages and when maintaining the existing ones. We validate our approach by using a realistic DSLs coming out from industrial case studies and obtained 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, and that offers the abstractions (a.k.a., *language constructs*) needed to describe certain aspect of the system. For example, we find DSLs to build graphical user interfaces [23], and to specify security policies [17].

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 [4]. This implies the development of many DSLs which is a challenging task due the specialized knowledge it demands. 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...) [13].

To improve cost-benefit when using DSLs, the research community in software languages engineering has proposed mechanisms to increase *reuse* during the language development process. The idea is to leverage previous engineering efforts and minimize implementation from scratch [25]. These reuse mechanisms are based on the premise that "software languages are software too" [10] thus, it is possible to use software engineering techniques to facilitate their construction [14]. For instance, there are approaches that take ideas from Component-Based Software Engineering (CBSE) [5] and Software Product Lines Engineering (SPLE) [28] 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., [19,15,26]) 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.

In practice, however, reuse is rarely achieved as a result of monolithic processes where language designers define language modules while trying to predict that they will be useful in future DSLs. Rather, the exploitation of reuse is often an iterative process where reuse opportunities are discovered, in the form of replicated functionalities, during the construction of individual DSLs. For example, many DSLs offer expression languages with simple imperative instructions (e.g., `if`, `for`), variables management, and mathematical operators. Xbase [1] is a successful experiment that demonstrates that, using compatible tooling, such replicated functionality can be encapsulated and used in different DSLs.

The major complexity of this reuse process is that both, the identification of reuse opportunities and the extraction of the corresponding languages modules are manually-performed activities. Due the large number of language constructs within a DSL, and the dependencies among them, this process is tedious and error prone. Language designers must compare DSLs in order to identify commonalities and, then, to perform a refactoring process to extract those commonalities on separated and interdependent language modules.

In this paper we propose a computer-aided approach to automatically identify reuse opportunities within a given set of DSLs, and to extract reusable language modules from those commonalities. To this end, we define comparison operators that we use to detect commonalities in a given set of DSLs during a static analysis process. In turn, we use principles from set and graph theories for the extraction of the reusable language modules. Our approach considers not only the syntax of the DSLs but also their semantics.

The validation of our approach is threefold. First, we implement our ideas in an Eclipse-based IDE in order to demonstrate that they are feasible with current capabilities of the existing language workbenches. Second, we evaluate the *correctness* of our approach by demonstrating that it can be properly applied in a real case study. In particular, we use a case study that has a direct application in the industry, and composed of three different languages for modeling state machines that share certain commonalities [8]. Third, we evaluate the *relevance* of our approach. To this end, we explore public GitHub repositories and download about 2400 DSLs where we apply an analysis in search of potential reuse. We found that, in the 50% of the cases, there are reuse opportunities that can be exploited with our approach.

The reminder of this paper is organized as follows: Section 2 introduces a set of preliminary definitions/assumptions that we use all along the paper. Section 3 presents a motivation to the problem by introducing an illustrating scenario.

Section 4 describes our approach that is evaluated in Section 5. Section 6 presents the related work and, finally, Section 7 concludes the paper.

2 Background: Domain-specific languages in a nutshell

We use this section to introduce some basic definitions intended to establish a unified vocabulary that facilitates the comprehension of the ideas presented in this paper.

- **DSLs specification** → Like general purpose languages (GPLs), DSLs are defined in terms of syntax and semantics [12]. 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 technological spaces available for the implementation of syntax and semantics of DSLs [20]. Language designers can, for example, choose between context-free grammars or and metamodels to specify syntax. Similarly, there are at least three manners to express semantics: operationally, denotationally, and axiomatically [21].

  In this paper we are interested on executable DSLs (xDSLs) which syntax is specified by means of *metamodels*, and semantics is specified operationally by means of *domain-specific actions* [6]. Each language construct is specified in a metaclass. The relationships between language constructs are specified as references between metaclasses. In turn, domain-specific actions are specified as java-like methods that are allocated in the metaclasses.

Example: A DSL for finite state machines. Figure 1 shows a example DSL for finite states machines. It contains three language constructs that are specified in metaclasses: `StateMachine`, `State`, and `Transition`. A state machine contains states and transitions. Those relationships are represented as containment references between the corresponding metaclasses.

The code snippets at the bottom of Figure 1 introduce the operational semantics to the DSL. In particular, the metaclass `StateMachine` is enriched with the operation `eval()` that contains a loop that sequentially invokes the `eval()` operation defined for the metaclass `State`. The metaclass `Transition` is enriched with the operation `fire()`.

3 Motivation

Consider a team of language designers working on the construction of the DSL for state machines presented in section 2. During that process, language design-

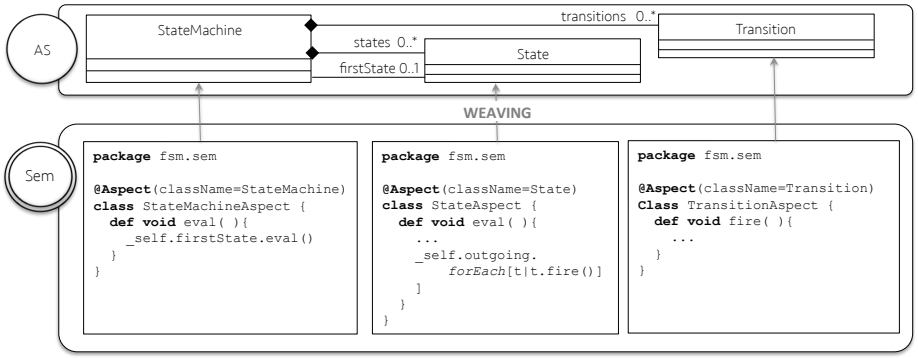

Fig. 1: A simple DSL for finite state machines

ers implement the language constructs typically required for expressing finite state machines: states, transitions, events, and so on. In addition, a constraints language that allows final users to express guards on the transitions should be provided, as well as an expressions language that allows to specify actions on the states of the state machine.

After being released their DSL for state machines, the language development team is required again. This time the objective is to build a DSL for manipulating the traditional Logo turtle, which is often used in elementary schools for teaching the first foundations of programming [22]. Certainly, the new DSL is essentially different from the DSL for state machines. Instead of states and transitions, Logo offers some primitives (such as `Forward`, `Backward`, `Left`, and `Right`) to move a character (i.e., the turtle) within a bounded canvas. Still, Logo also requires an expressions language in order to specify complex movements. For example, final users may write instructions such as: `forward (x + 2)`.

At this point, language designers face the problem of reusing the expressions language they already defined for the state machines DSL. The typical solution to this type of situations is to clone the existing language constructs in the second DSL. Language designers usually copy/paste the segment of the specification that they want to reuse. This situation is repeated each time that there is a new DSL to build. For example, if our language designers team is required to build a third DSL (such as a flowcharts language) that uses not only expressions but also constraints, they will have (again) to copy/paste these specification segments. As a result, after the construction of some DSLs, there are clones that are expensive to maintain.

Overlapping in DSLs and potential reuse. Certainly, the aforementioned phenomenon was previously observed in the literature. In the "DSLs Engineering" book, Vöelter et al [27, p. 60-61] show that, although many of the existing DSLs are completely different and tackle independent domains, there are also DSLs with overlapping domains; they share some language constructs. Figure 2 illustrates this observation for the case of our illustrating scenario. We

use two Venn diagrams to represent both syntax and semantic overlapping. For the technological space discussed in this paper, syntactic overlapping corresponds to intersections between sets of metaclasses. In turn, semantic overlapping corresponds to intersections between sets of domain-specific actions.

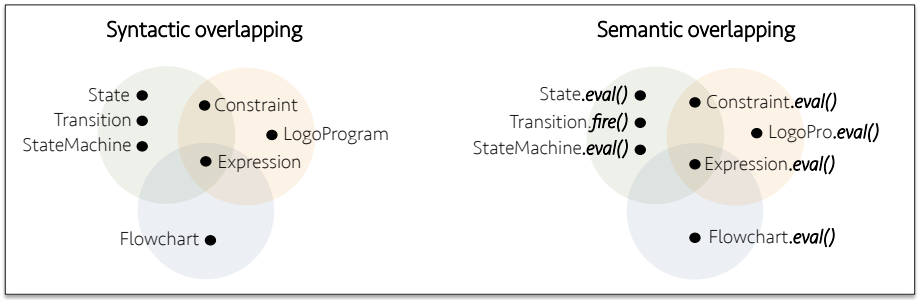

Fig. 2: Visualizing syntactic and semantic commonalities

Note that if a set of DSLs have some overlapping, and in addition they specified in the same technological space, then there is **potential reuse**. The specification of those shared constructs can be specified once and reused in the two DSLs [27, p. 60-61].

It is worth to mention that the fact that two metaclasses are identical does not imply that all their domain-specific actions are equal as well. 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.

4 Proposed approach

Our objective is to extract a catalog of reusable language modules from a given set of DSLs (that we refer to as the *input set*). To this end, we propose an approach based on the aforementioned notions of overlapping and potential reuse. Concretely, in our approach we first identify syntactic and semantic overlapping among the DSLs of the input set. Then, we cut such overlapping in order to break-down the DSLs in reusable language modules. Those language modules are encapsulated in such a way that they can be later composed among them to obtain complete DSLs. The overall strategy is illustrated in Figure 3. This section is dedicated to explain it in detail.

4.1 Identifying overlapping: *match* and *merge*

Our strategy to identify syntactic overlapping is based on the fact that a metamodel can be viewed as a directed graph $G =< V, A >$ where the vertexes V

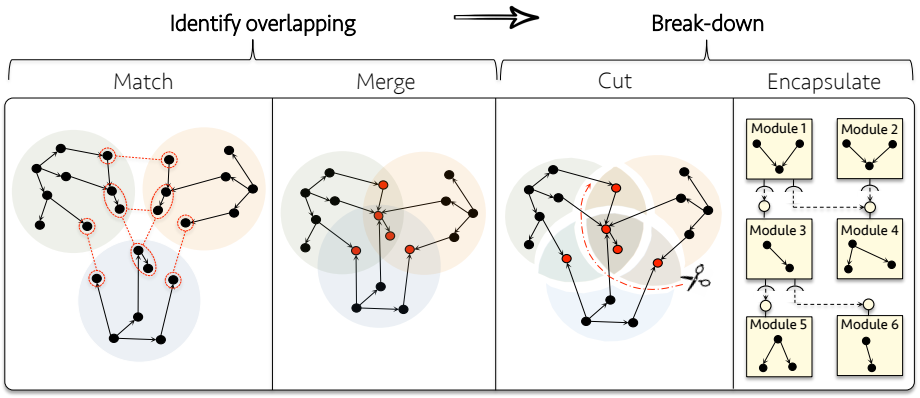

Fig. 3: Breaking down the input set by cutting overlapping

represent metaclasses and the arcs A represent references between metaclasses. Each DSL of the input set has a metamodel that represents its syntax. Syntactic overlapping is detected by identifying replicated vertex among all those graphs. Then, we organize the merged graphs in the form of a Venn diagram as illustrated in the two first steps shown in Figure 3.

More concretely, our algorithm to identify syntactic overlapping is twofold. First, we perform a matching process that receives a set of graphs (one for each metamodel) and returns the collection of vertexes referencing metaclasses that are *identical*. Second, we merge the matched vertexes, thus removing replications. In order to create the intersections, we store the information about what are the DSLs where the corresponding metaclass is defined.

Once matched vertexes are merged, we analyze the domain-specific actions associated to the corresponding metaclasses. Particularly, for each set of identical metaclasses we check if the domain-specific actions are equal as well. If so, they can considered as replicated code and we have semantic overlapping. Those domain-specific actions are merged.

In the case in which not all the domain-specific actions associated to two matched metaclasses are the same, we create different clusters of domain-specific actions, thus establishing a semantic variation point.

It is worth noting that detection of both syntactic and semantic overlapping relies on comparison of metaclasses and domain-specific actions respectively. At this point we need to clearly define such notion of equality that, as a matter of fact, is quite important to avoid alterations on the DSLs after the extraction of reusable language modules.

Comparison of metaclasses: An operator for metaclasses comparison can be specified as follows:

$$\doteq \ : MC \times MC \rightarrow bool \tag{1}$$

To implement such operator, one can intuitively consider as a first approach the comparison of metaclasses names matching; two metaclasses are considered equal if their names are equal. Certainly, this approach results quite useful and it is quite probable that, we can find potential reuse. For example, one can expect that in a set of DSLs for finite state machines DSL, the construct `Transition` can be considered as a commonality.

Unfortunately, comparison of metaclasses by using only their names might have some problems. There are cases in which two metaclasses with the same name are not exactly the same because they do not represent the same domain concept, or because there are domains that use similar vocabulary. For instance, whereas in many cases the transitions of a state machine are only specified in terms of triggers and constraints, there are certain DSLs for state machines that allow to annotate transitions are annotated with time flags [11].

In such cases, feasibility of potential reuse is not clear because the involved metaclasses offer different functionalities. Hence, a more restrictive operator should be considered. An approach that certainly helps is to compare metaclasses not only by their names but also by their attributes and references. Although this strategy can be quite restrictive, it guarantees that the detected reuse opportunities correspond to exact code clones so they can be extracted as reusable modules without any risk of altering the behavior of the DSLs.

In our approach we use the later strategy. Nevertheless, we consider that certain flexibility might be to define those operators. We provide an extensible approach where other operators (such as the surveyed in [16]) can be easily incorporated.

Comparison of domain-specific actions: In turn, the operator for comparison of domain-specific actions can be specified as follows:

$$\stackrel{\circ}{=} \; : DSA \times DSA \rightarrow bool \tag{2}$$

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 implementation of a comparison operator for 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, during the conduction of this research we realized that 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. For example, we can found semantic variation points in the implementation of DSLs for state machines where the domain-specific actions are implemented differently although their signatures are the same. As a result, we only guarantee potential reuse where we compare also 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 domain-specific actions we will have to deal with

the semantic equivalence problem that, indeed, is known to be undecidable [18]. To deal with this issue, we compare the body of domain-specific actions by in terms of its abstract syntax tree as proposed by Biegel et al [2]. This strategy guarantees that semantic overlapping correspond to exact clones. Again, our approach can be extended to support other comparison operators for domain-specific actions.

4.2 Breaking down the input set: *cut* and *encapsulate*

After being identified overlapping among the DSLs in the input set, we extract a set of reusable language modules. To this end, we adopt the idea presented by Vöelter et al [27, p. 60-61]: we break-down the overlapping by creating one language module for each different intersection as illustrated in the third step of Figure 3. The reasoning to this solution is quite simple: by definition, an intersection is a set of language constructs that are shared by two or more DSLs. If we extract those language constructs in separated language modules, the language module can be defined once and reused by all the DSLs that require it. So, we can consider that the extracted language module is reusable.

In our approach, we implemented this separation of overlapping as a graph partitioning algorithm. The algorithm receives the graph(s) obtained from the merging process and returns a set of vertex clusters: one cluster for each intersection of the Venn diagram. Arcs defined between vertexes in different clusters can be considered as cross-cutting dependencies between clusters. Then, we encapsulate each vertex cluster in the form of language modules. Each module contains a metamodel, a set of domain-specific actions, and a set of dependences towards other language modules.

Dependencies between language modules are supported by means of the classical required and provided roles in components-based software development. 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, containment references, or inheritances) 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. 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) (see Figure 4).

We use *model types* [24] to express both required and provided interfaces. 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 [9]). 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.

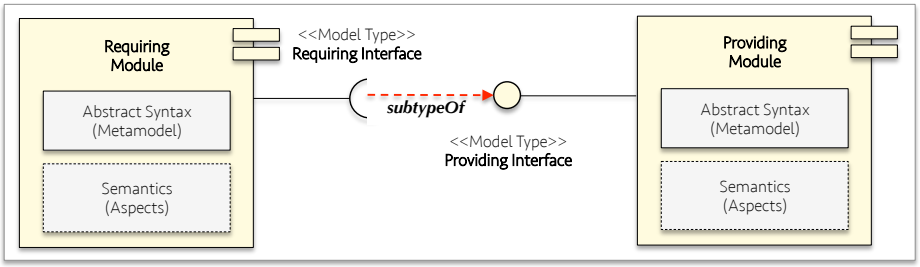

Fig. 4: Interfaces for language modules

5 Evaluation

The evaluation of our approach is threefold. First, we implement the ideas on top of an existing language workbench. Second, we evaluate the *correctness* of the approach by means of a test oracle that consists of a well-documented case study where we exactly know the existing overlapping among the involved DSLs. We execute our approach on the case study and we check that the input matches the expected overlapping. Third, we evaluate *relevance*. More concretely, we use empirical data to demonstrate that the phenomenon of syntactic and semantic overlapping is actually appearing in realistic DSLs that we obtain from public `GitHub` repositories. Thus, we show that there is room in real projects to the applicability of our approach.

5.1 Implementation: The Puzzle tool-suite

The approach presented within this paper is implemented in the **Puzzle** tool-suite. Puzzle is developed on top of the Eclipse Modeling Framework (EMF). In that context, 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 the Melange [9]. The implementation details, examples, and some tool demonstrations can be downloaded from the project's website[3].

5.2 Evaluating *correctness*: The state machines case study

The case study we use to evaluate the correctness of our approach was introduced by Crane et al [8], and it is composed of three different DSLs for expressing state machines: UML state diagrams, Rhapsody, and Harel's state charts. This case study describes a realistic problematic that has an impact on the industry. As a

[1] `http://www.eclipse.org/xtend/`
[2] `https://github.com/diverse-project/k3/wiki/Defining-aspects-in-Kermeta-3`
[3] `http://damende.github.io/puzzle/`

matter of fact, it is one of the motivations of the VaryMDE[4] project which is a collaboration between Thales Research & Technology, and INRIA.

Test oracle → Naturally, because the three DSLs are intended to express the same formalism, they have commonalities, i.e., overlapping. For example, all of them provide basic concepts such as `StateMachine`, `State`, and `Transition`. However, not all those DSLs are exactly equal. They have both syntactic and semantic differences which are well documented in the Crane's article [8].

Syntactic differences are materialized by the fact that not all the DSLs provide exactly the same constructs. There are differences in the support for transition's triggers and pseudostates. Whereas Rhapsody only supports simple triggers. Harel's state charts and UML provide support composed triggers. More concretely, in Harel's state charts triggers can be composed by using `AND`, `OR`, and `NOT` operators. In turn, in UML triggers can be composed by using the `AND` operator.

In addition, whereas there are pseudostates that are supported by all the DSLs (`Fork`, `Join`, `ShallowHistory`, and `Junction`); there are two psueudostates i.e., `DeepHistory` and `Choice` that are only supported by UML. The `Conditional` pseudostate is only provided by Harel's state charts.

Figure 5 shows a table with the language constructs provided by each DSL. In summary, there are: **17** language constructs that are shared by all the DSLs; **1** construct that is exclusive of UML; **2** constructs that are exclusive of Harel's state charts; **2** constructs shared by UML and Harel's state charts; and, finally, **1** construct shared by Harel's state charts and Rhapsody.

Language vs. Construct	StateMachine	Region	AbstractState	State	Transition	Trigger	NotTrigger	AndTrigger	OrTrigger	Pseudostate	InitialState	Fork	Join	DeepHistory	ShallowHistory	Junction	Conditional	Choice	FinalState	Constraint	Statement	Program	NamedElement	Total of constructs
UML	●	●	●	●	●	●	·	●	·	●	●	●	●	●	●	●	·	●	●	●	●	●	●	20
Rhapsody	●	●	●	●	●	●	·	·	·	●	●	●	●	·	●	●	●	·	●	●	●	●	●	18
Harel	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	·	●	●	●	●	●	22

Fig. 5: Oracle for evaluation of correctness

Semantic differences are materialized by the fact that not all the DSLs have the same behavior at execution time. For example, whereas in Harel's state charts simultaneous events are attended in parallel, both UML and Rhapsody follow the run to completion principle so simultaneous events are attended sequentially.

As a direct consequence of those semantic differences, not all the domain-specific actions are the same. For instance, due to the semantic difference in the events treatment policy explained above, the methods `eval()` and `step()` in the `StateMachine` metaclass are different in each DSL.

[4] `http://varymde.gforge.inria.fr/`

Results → Figure 6 shows the results of the first part of the analysis. It presents the Venn diagram that shows syntactic and semantic overlapping. Note that, in the case of the syntactic overlapping, the cardinalities of the intersections in the Venn diagram match the test oracle thus, demonstrating that our approach to comparison of metaclasses is correct. In turn, the methods `eval()` and `step()` of the `StateMachine` metaclass are correctly identified as different in each DSL.

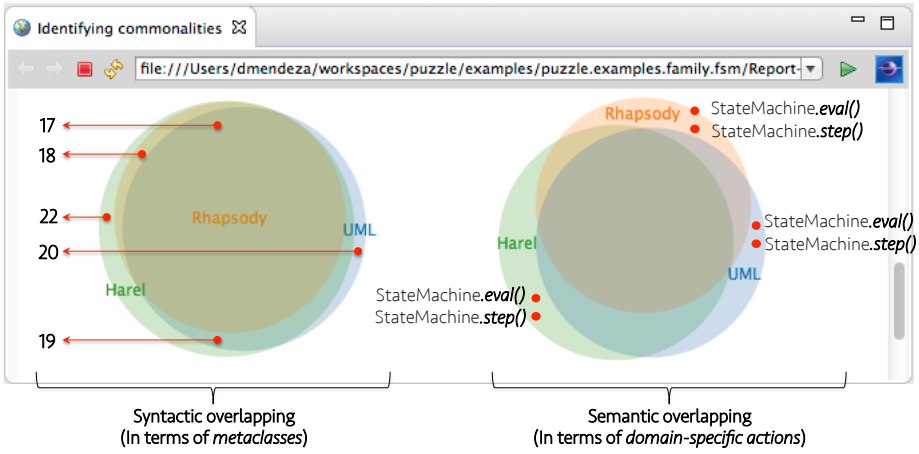

Fig. 6: Overlapping detected using the Puzzle toolsuite in the state machines case study.

In turn, Figure 7 shows the results for the second and third steps of our approach: identifying and extracting reusable language modules. There is a language module that contains all the language constructs shared by the three DSLs. This language module can be considered as a basic DSL for state machines that supports the basic constructs. It can be used in the construction of new DSLs for state machines. In addition, there are other language modules encapsulate pseudostates and triggers separately. Note that in order to obtain the Harel's state charts language, we need to compose the modules 1, 2, and 5. In turn, to obtain UML we need to compose modules 1, 3, and 4. Finally, to obtain Rhapsody we need to compose modules 1 and 5.

This first experiment can be replicated by following the instructions presented in a public website[5].

5.3 Evaluating *relevance*: Identifying potential reuse in the wild

In order to evaluate the relevance of our approach, we conducted an study on empirical data intended to answer two questions: 1) What is the probability

[5] Website for experiment 1: `http://damende.github.io/puzzle/extractingmodules`

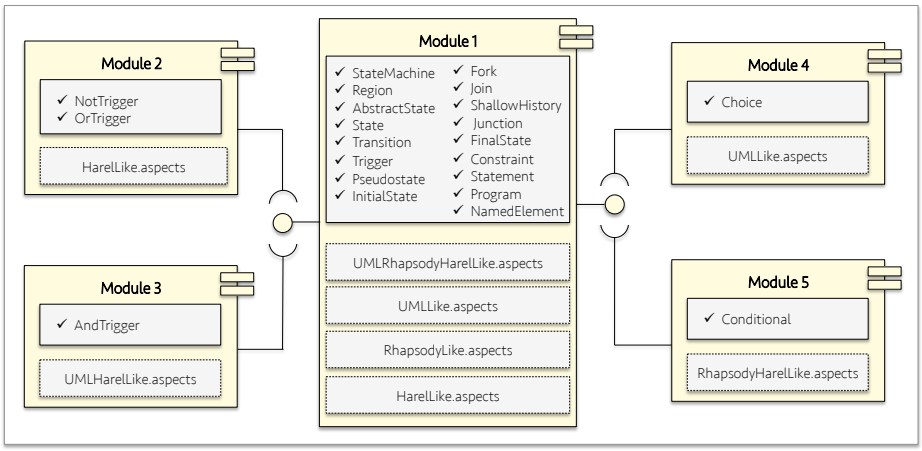

Fig. 7: Results for the state machines case study: extracting language modules

that a DSL has some overlapping with another DSL?; and 2) How big is the average overlapping shared by the existing DSLs? The reminder of this section is dedicated to explain how we obtain the empirical data we use and how we use that data to answer the questions. In this experiment, we use the comparison operators defined in section 4.1. All the data and tooling needed to replicate these experiments can be downloaded in the experiment website [6].

Collecting empirical data → To collect empirical data, we explored `GitHub` repositories in search of DSLs that are built on the same technological space that we used in our approach. Namely, metamodels written in Ecore with operational semantics defined as domain-specific actions in Xtend. The objective was to build a realistic data set composed of DSLs developed by diverse development teams.

As a result of this search, we found **2423** metamodels (after discarding metamodels with errors). Contrariwise, due to the fact that Kermeta 3 and its implementation in Xtend is a quite recent idea, we found very few data for the semantics part. We decided to conduct analysis only in the metamodels thus, the syntactic part of the languages. We consider that such analysis a good insight to know if there is potential reuse. In the following, we refer to our empirical data as S which is a set of metamodels.

What is the probability that a DSL has some overlapping with another DSL? → To answer this first question, we compute the relative frequency of the event `E`: *"the metamodel has some overlapping with, at least, another metamodel"* in the set S. Formally:

$$\begin{aligned} P(E) \approx \ & RelativeFrequency(E) = \frac{|S_E|}{|S|} \\ & where\ S_E = \{x \in S \mid (\exists y \in S \mid (x \neq y \ \wedge x \cap y \neq \emptyset))\} \end{aligned} \tag{3}$$

[6] Website for experiment 2: `http://damende.github.io/puzzle/reusewild`

Note that computing such probability corresponds to scan the set S in search of occurrences of the event E. After doing so, we obtained that that the relative frequency is $\mathbf{1215/2423 = 0.50}$.

From a pragmatic point of view, this result means that there are **50%** of probabilities that, during the construction of a new DSL, a language designer is replicating at least one metaclass defined in at least one legacy DSL that is available in `GitHub`. This probability is quite elevated if we take into account that our comparison operator for metaclases is quite restrictive.

Although this result is quite encouraging, we still wonder how important is the potential reuse. This motivate the following question:

How big is the average overlapping existing among DSLs? → To compute the average of overlapping among the metamodels in the set S, we performed a pair-wise comparison between the metamodels to obtain the size of the overlapping (i.e., amount of shared metaclasses) between all the possible pairs. Then, we compute the average of overlapping that each metamodel has with the other metamodels of the set. To compute that average, we consider only the metamodels where the overlapping is at least 1 metaclass. Finally, we compute the average of these averages.

After executing this experiment, we obtain that the average of overlapping is 5.31. If we analyze the results of questions one and two at the same time, we can conclude that during the construction of the **50%** of the new DSLs language designers are replicating, in average, **5.31** language constructs. If we take into account that, in our empirical data, the average number of constructs of a DSLs is **28.4**, we found that the reuse opportunities are about **18%**.

6 Related work and discussion

Leveraging reuse in the construction of DSLs is an objective that has been previously discussed in the research community on software languages engineering. One of the very first achievements towards this objective was the notion of components-based language development. With the time passing, approaches are becoming more sophisticated, thus supporting more complex modularization scenarios, and being applicable for more diverse technological spaces [19,15,26].

More recent approaches are focused not only on dealing with modularization issues, but also on facilitating the reuse process itself. For instance, Melange [9] is a tool-supported approach that introduces some operators (such as slice, inheritance, and merge) intended to manipulate legacy DSLs in such a way that they can be easily integrated in new developments. Using Melange, a language designer can combine a set of DSLs in different ways to produce a new one.

The main contribution of our approach is that it advances towards the automation of the reuse process. We demonstrate that, with the correct abstractions and assuming some constraints, the process can be automated by means of reverse-engineering techniques. It is important mentioning that our approach uses some of the ideas presented by Caldiera and Basili [3]. That approach proposes reverse-engineering methodologies for extracting reusable modules from

legacy C code. In addition, our work is based on an observation about commonalities and potential reuse provided by Vöelter et al [27, p. 60-61]. We show that the notion of commonalities is quite useful for extracting reusable language modules.

There are, however, some open issues that need further investigation. In particular, during the conduction of this research, and trying to apply the approach in further case studies, we realized that the comparison operators to detect those commonalities can become an Achilles' heel. In some cases, the notion of commonality can be associated to a given *functionality* (abstractly speaking), more than equality in the specification. For example, there are many DSLs that use constraints languages but that use different language constructs to support them. They share the functionality of constraint languages, but the specifications do not match. That does not mean that there is not potential reuse. The detection of this kind of commonalities can become quite difficult due to the ambiguity in that notion of functionality.

As a matter of fact, considering more flexible approaches for the detection of commonalities can have additional advantages. Consider for example two DSL that define different constraints languages where one is better defined (more completely or with more powerful capabilities) than the other. If this situations can be detected, and language designers can chose a preferred language module, our approach can become useful not only to achieve reuse but also to improve quality of existing DSLs. We claim that more complex overlapping identification (probably with human intervention) should be provided.

7 Conclusions

In this paper, we presented an approach to exploit reuse during the construction of DSLs. We show that it is possible to partially automate the reuse process by identifying overlapping among DSLs and automatically extracting reusable language modules that can be later used in the construction of new DSLs.

We evaluated our approach in a real industrial case study and we demonstrate that there is an important amount of potential reuse in DSLs in public repositories. More concretely, based on empirical data, we showed that in about the **50%** of the new DSLs there are reuse opportunities to exploit reuse. This reuse is, in average, about the **18%** of the size of the DSLs.

References

1. L. Bettini, D. Stoll, M. Völter, and S. Colameo. Approaches and tools for implementing type systems in xtext. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 392–412. Springer Berlin Heidelberg, 2013.
2. 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.

3. G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, Feb. 1991.
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, 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. M. Crane and J. Dingel. Uml vs. classical vs. rhapsody statecharts: not all models are created equal. *Software & Systems Modeling*, 6(4):415–435, 2007.
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.-M. Favre, D. Gasevic, R. Lämmel, and E. Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, volume 6563 of *LNCS*, pages 316–326. Springer, 2011.
11. S. Graf and A. Prinz. Time in state machines. *Fundam. Inf.*, 77(1-2):143–174, Jan. 2007.
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. Gasevic, 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. A. Olson, T. Kieren, and S. Ludwig. Linking logo, levels and language in mathematics. *Educational Studies in Mathematics*, 18(4):359–370, 1987.
23. 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.
24. J. Steel and J.-M. Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.
25. 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.
26. M. Vöelter. Language and ide modularization and composition with mps. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 383–430. Springer Berlin Heidelberg, 2013.
27. 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.
28. 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. Gasevic, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer Berlin Heidelberg, 2010.