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. In such context, a common practice for enabling reuse is the definition of languages modules that can be later put togueter in order to build up new DSLs. However, the identification and reuse of those modules in a set of existing languages is currently relying on manual and costly tasks thus, hindering the reuse exploitation when developing DSLs. In this paper, we propose a solution to i) detect and identify possible reuse in a set of DSLs; and ii) break down those languages into modules that can be later exploited to minimize costs when constructing new languages and when maintaining the existing ones. Finally, we validate our approach by using a realistic DSLs coming out from industrial projects 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 [8]. A DSL is a software language in which expressiveness is scoped into a well-defined domain that offers a set of abstractions (a.k.a., *language constructs*) needed to describe certain aspect of the system [6]. For example, in the literature we can find DSLs for prototyping graphical user interfaces [27], specifying security policies [20], or performing data analysis [11].

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 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...) [16].

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 [29]. These reuse mechanisms are based on the premise that “software languages are software too” [12] so it is possible to use software engineering techniques to facilitate their construction [17]. For instance, there are approaches that take ideas from Component-Based Software Engineering (CBSE) [5] and Software Product Lines Engineering (SPLE) [31] 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., [28,22,18]) 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 during the construction of individual DSLs in the form of replicated functionalities that could be extracted as reusable language modules. 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 demonstrate 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.

Inspired in the work of Caldiera and Basili [3], in this paper we propose a computer-aided mechanism 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, the extraction of the reusable language module is based on some basic principles from set theory. It is worth mentioning that our approach is tool-supported; we provide an Eclipse-based IDE that implements the ideas presented in this paper. Besides, our approach considers not only the syntax of the DSLs but also their semantics.

The validation of our approach is twofold. On one hand, we evaluate *correctness* by demonstrating that our ideas and tool support can be actually applied in a real case study. In particular, we a case study composed of three different languages for modeling state machines that share certain commonalities [9]. We show that those commonalities can be extracted in reusable language modules. On the other hand, we evaluate the *relevance* of our approach. To this end, we explore public GitHub repositories and download a big set of DSLs where we apply our analysis. We found that there is large potential reuse in the wild.

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 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

Specification. Like general purpose languages (GPLs), DSLs are defined in terms of syntax and semantics [15]. Hence, the specification of a DSL is a tuple $< syn, sem, M_{syn \leftarrow sem} >$ [7]. 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 [23]. Language designers can, for example, choose between using context-free grammars or metamodels as specification formalism for syntax. Similarly, there are at least three technological spaces for expressing semantics: operationally, denotationally, and axiomatically [24].

In this paper we are interested on executable DSLs (xDSLs) which syntax is specified by means of metamodels, written in the Ecore language, and semantics is specified operationally as methods (a.k.a, *domain-specific actions* [7]) in Xtend[1]. 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. The mapping between metaclasses and domain-specific actions is specified by using the notion of aspect introduced by the Kermeta 3[2] and Melange as explained in [10].

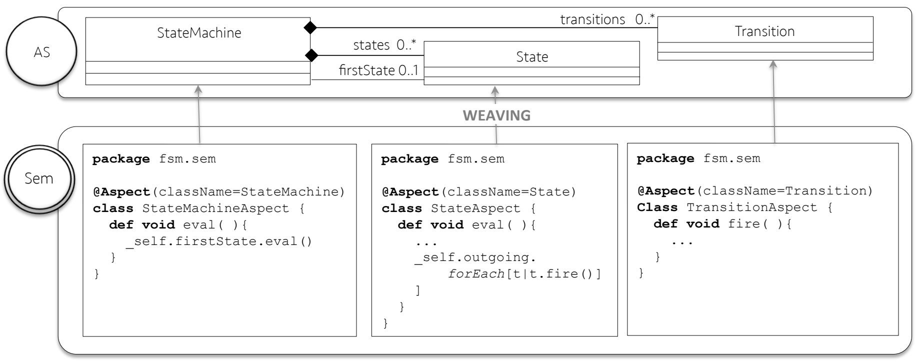
Example: A DSL for finite state machines. Let us illustrate executable DSLs by means of a simple example. Consider the metamodel for a simple language for finite state machines introduced at the top of Figure 1. It contains

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

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

thee classes `StateMachine`, `State`, and `Transition`; The class `StateMachine` contains both states and transitions which is represented with containment references.

The code snippets at the bottom of Figure 1 introduce some operational semantics to this metamodel by using K3. Note that the main feature of Kermeta 3 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 DSL for finite state machines

3 Motivation

3.1 Illustrating scenario

Consider a team of language designers working on the construction of the DSLs for state machines presented in section 2. During that process, language designers implement the language constructs typically required for expressing finite state machines: states, transitions, events, and so on. In addition, the DSL is intended to provide a constraints language that allows final users to express guards on the transitions. Moreover, the DSL also provides an expressions language that allows to specify actions on the states of the state machine. This expressions language offers classical capabilities such as arithmetic operations, variables management, and imperative procedures.

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 [25]. 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. However, Logo also requires an expressions language in order to specify complex movements. For example, final users may write instructions such as: `forward (x + 2)` where `x` is a variable.

At this point, language designers face the question of how to reuse the expressions language they already defined for the state machines language. As illustrated in Figure 2, the typical solution to this type of situations is to replicate the code in a second DSL. Language designers usually copy/paste the segment of the specification that they can reuse. As a result, we have many clones that are expensive to maintain. Naturally, this situation is repeated each time that there is a new DSL to build. This fact is illustrated in the Figure 2 by introducing a third DSL for expressing flowcharts. In this case, the new DSL requires both, expressions and constraints languages.

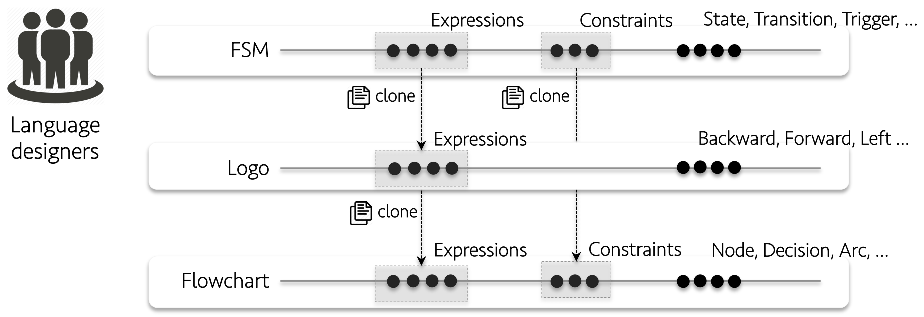


Fig. 2: Cloning in DSLs development process

3.2 Overlapping in DSLs and potential reuse

The aforementioned phenomenon was previously observed by Vöelter et al [30, p. 60-61]. In fact, that study demonstrates 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. Figure 3 illustrates this observation for the case of our illustrating scenario and by using two Venn diagrams to represent 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.

If a set of DSLs have some overlapping and they are specified in the same technological space and using compatible language workbenches, then there is

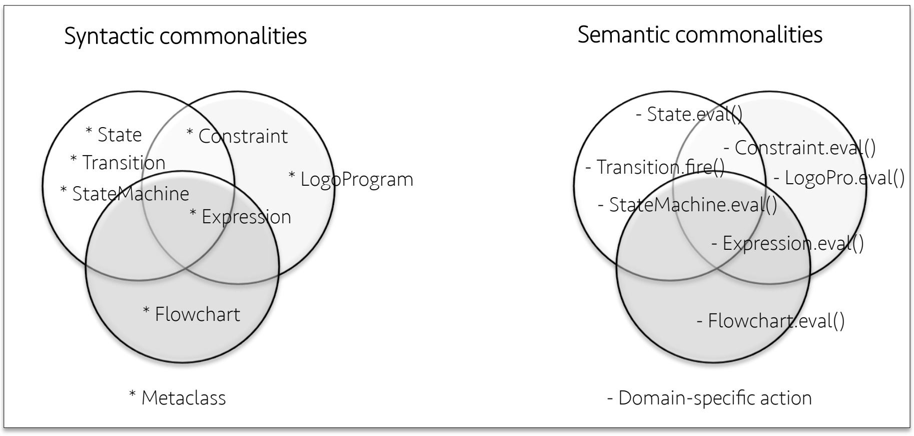


Fig. 3: Visualizing syntactic and semantic commonalities

potential reuse since the specification of those shared constructs can be specified once and reused in the two DSLs [30, p. 60-61]. 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.

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 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

Given a set of existing DSLs our approach is intended to identify a catalog of reusable language modules from a given set of DSLs (that we refer to as the *input set*). To this end, our approach is threefold (see Figure 4). **First**, we identify commonalities existing in a given set of DSLs. Those commonalities are structured as the form of Venn diagrams containing syntax and semantic overlapping. **Second**, we break-down the Venn diagrams by separating all the intersections; for each different intersection, we create a language module by considering not only syntax but also semantics. **Third**, we encapsulate those language modules and we establish the dependencies among them by means of interfaces. The reminder of this section is dedicated to deeply explain our approach.

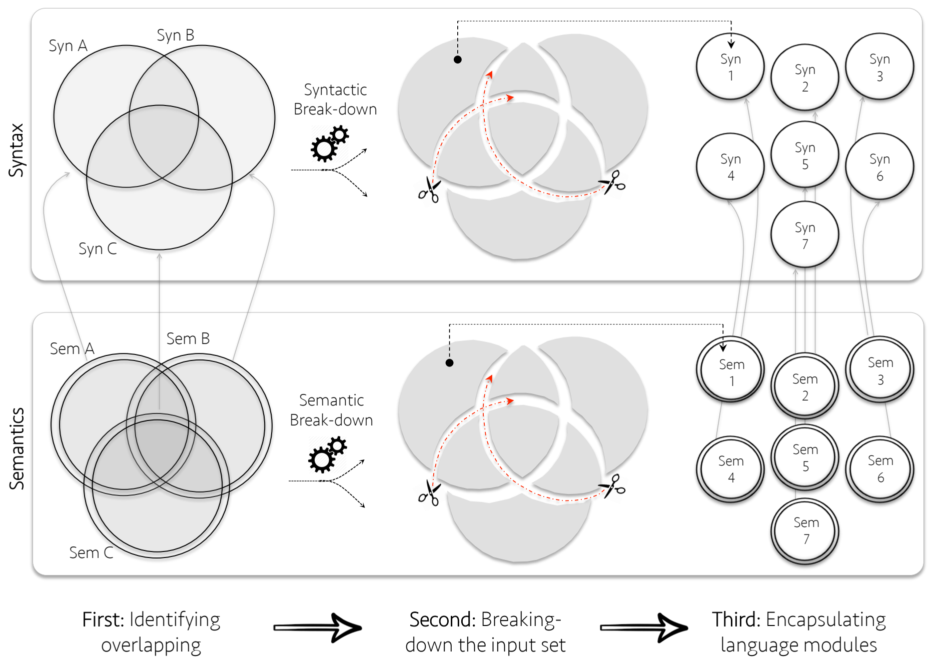


Fig. 4: Breaking down the input set by separating overlapping

4.1 Identifying overlapping

Our algorithm for identifying **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 (i.e., metaclasses that are *equal*) 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 (i.e., domain-specific actions that are *equal*) among these aspects.

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. Certainly, this approach results quite useful and it is quite probable that, we can find potential reuse. Unfortunately, comparison of metaclasses by using only their names might have some problems. There are cases in which two meta-classes with the same name are not exactly the same since they do not represent the same domain concept or because there are domains that use similar vocabulary. In such cases, an approach that certainly helps is to compare meta-classes not only by their names but also by their attributes and references. Hence, we define a comparison operator for metaclasses i.e., $\doteqdot$ defined as follows.

$$\begin{aligned} MC_A \doteqdot MC_B = true \implies & \\ & MC_A.name = MC_B.name \land \\ & \forall a_1 \in MC_A.attr \mid (\exists a_2 \in MC_B.attr \mid a_1 = a_2) \land \\ & \forall r_1 \in MC_A.refs \mid (\exists r_2 \in MC_B.refs \mid r_1 = r_2) \end{aligned} \tag{1}$$

Our comparison operator for metaclasses is quite restrictive. It identifies two metaclasses as a commonality if and only if their specifications are exactly the same. Hence, potential reuse is guaranteed in the sense that we are sure that there are not additional elements to consider. However, we are aware that there are other approaches to compute commonalities within metaclasses. So, at the implementation level our approach is flexible and additional comparison operators such as the surveyed in [19] 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. 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. 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 actions we will have to deal with the semantic equivalence problem that, indeed, is known as be undecidable [21]. 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 [2].

$$
\begin{aligned}
&DSA_A \mathrel{\hat{=}} DSA_B = true \implies \\
&\qquad DSA_A.name = DSA_B.name \wedge \\
&\qquad DSA_A.returnType = DSA_B.returnType \wedge \\
&\qquad DSA_A.visibility = DSA_B.visibility \wedge \\
&\qquad \forall p_1 \in DSA_A.params \mid (\exists p_2 \in DSA_B.params \mid p_1 = p_2) \wedge \\
&\qquad DSA_A \mathrel{\hat{=}} DSA_B \wedge \\
&\qquad DSA_A.AST = DSA_B.AST
\end{aligned} \tag{2}
$$

4.2 Breaking down the input set

After being identifying 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 [30, p. 60-61]; we break-down the overlapping by creating one language module for each different intersection as illustrated in Figure 4. 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.

4.3 Encapsulating language modules

Once we have identified the reusable language modules, we encapsulate them in such a way that each language module contains a metamodel and a set of domain-specific actions. Besides, the dependencies among the language modules are expressed.

The dependencies among language modules are expressed as follows: 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 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.

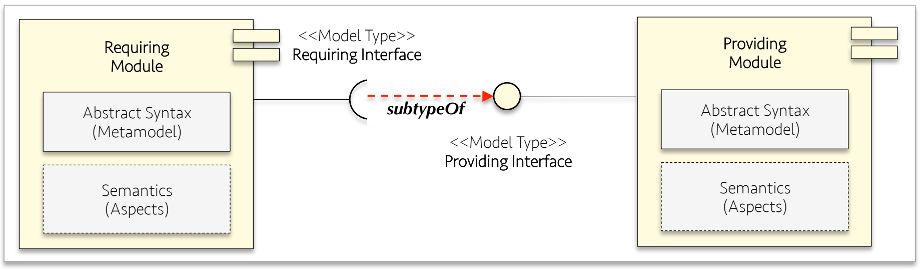

Requiring Module
Abstract Syntax (Metamodel)
Semantics (Aspects)
<<Model Type>> Requiring Interface
subtypeOf
<<Model Type>> Providing Interface
Providing Module
Abstract Syntax (Metamodel)
Semantics (Aspects)

Fig. 5: Interfaces for language modules

5 Evaluation

In this section we present the validation of our approach. As aforementioned, this validation is twofold. On one hand, we demonstrate that our approach is correct. To do so, we take use a case study that is well documented in [9] and where we exactly know the commonalities existing among the input set. We execute our approach and we compare the results while expecting that the input of our tool matches with the commonalities we know. It is important to mention that this case study corresponds to a real problematic in the industry. Concretely, it is one of the motivations of the VaryMDE[3] project which is a collaboration between Thales Research & Technology, and INRIA.

The second part of the evaluation is intended to demonstrate that our approach is relevant. We demonstrate, with empirical data, that the phenomenon of syntactic and semantic commonalities is currently appearing in real DSLs and, so, there is an important amount of potential reuse in the wild and our approach can be actually useful.

5.1 Evaluating *correctness*: The state machines case study

The case study is composed of three different DSLs for expressing state machines: UML state diagrams [26], Rhapsody [13], and Harel's state charts [14]. Because the three DSLs are intended to express the same formalism, they have commonalities. For example, all of them provide basic concepts such as `StateMachine`, `State`, `Transition`, or `Trigger`. However, not all those DSLs are equal. In fact, they have some syntactic and semantic differences which are well documented in the Crane's article [9]. To validate our approach we implemented these three DSLs while strictly following that documentation. Thus, we obtained a set of

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

DSLs to test our tool and we know in advance the results that the tool should provide. We use that as an oracle to test our approach.

Oracle. Figure 6 shows a table with the constructs contained by each DSL in the case study. Note that not all the DSLs have exactly the same constructs. The main differences are in the support for types of triggers. Whereas Rhapsody only supports simple triggers. Harel's state charts and UML provide support composing triggers. In particular, 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 only the `AND` operator. Another difference between the DSL of our case study corresponds to the different support for pseudostates. Whereas there are pseudostates that are supported by all the DSLs (`Fork`, `Join`, `ShallowHistory`, and `Junction`); there are certain psueudostates such as `DeepHistory`, `Conditional`, or `Choice` that are not supported in all of the DSLs.

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
UML	●	●	●	●	●	●	-	●	-	●	●	●	●	●	●	●	-	●	●	●	●	●	●
Rhapsody	●	●	●	●	●	●	-	-	-	●	●	●	●	-	●	●	●	-	●	●	●	●	●
Harel	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	-	●	●	●	●	●

Fig. 6: Oracle for evaluation of correctness

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 1 construct shared by Harel's state charts and Rhapsody.

Results. Figure 7 shows the results of the first part of the analysis. It presents the Venn diagram produced for the case study of the state machines. Note that the numbers correspond to our oracle demonstrating that our approach is correct. Due to lack of space, we do not present the semantic results. However, in a later section we present a tool demonstration that shows the complete functionality of our tool.

Figure 8 shows the results for the second and third part of our approach: identifying and extracting reusable language modules. Note that there is a language module (core) that contains all the language constructs that are shared by the three DSLs. Then, the other language modules encapsulate pseudostates and triggers separately since they represent the differences among the DSLs. Note that in order to obtain the Harel's state charts language, we need to compose

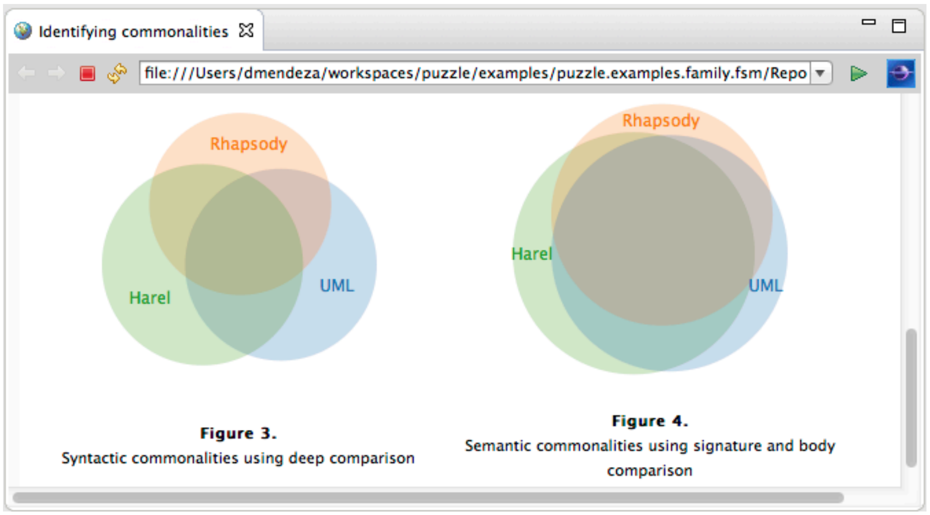

Identifying commonalities
file:///Users/dmendeza/workspaces/puzzle/examples/puzzle.examples.family.fsm/Repo
Rhapsody
Harel
UML
Figure 3.
Syntactic commonalities using deep comparison
Rhapsody
Harel
UML
Figure 4.
Semantic commonalities using signature and body comparison

Fig. 7: Results for the state machines case study: identifying overlapping

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.

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

In order to identify potential reuse in the wild, we explored the public `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 in Xtend as domain-specific actions. The objective was to build a data set composed of DSLs developed by diverse development teams.

As a result of this search, we found 2800 Ecore 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 only found some few DSLs all of them developed in our research team (which is the team that developed Kermeta 3). We decided to conduct analysis only in the metamodels. We consider that such analysis a good insight to know if there is potential reuse.

The questions. Our analysis to evaluate potential reuse in the wild was guided by two questions: 1) What is the probability that a metamodel share some commonalities with another one? and 2) How big is the average commonality between metamodels?

The experiments. In order to answer the first question, we conducted an experiment intended to compute the number of metamodels that share at least one construct with another metamodel. In other words, we consider our data

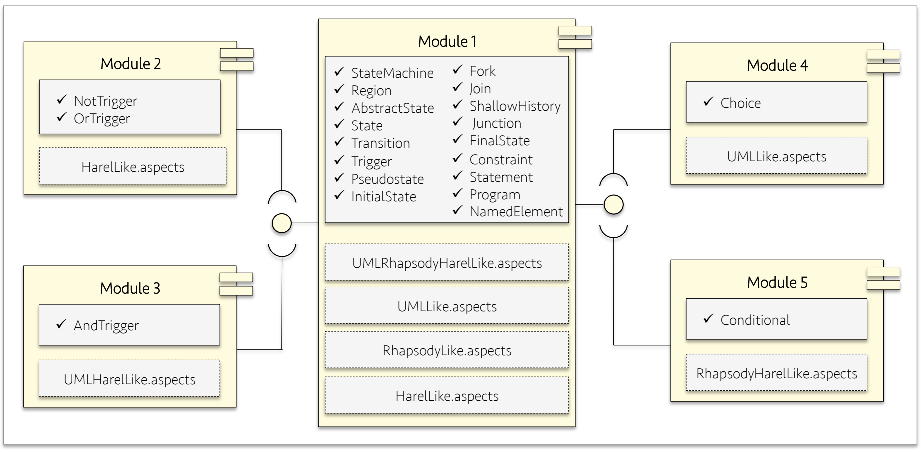


Fig. 8: Results for the state machines case study: extracting language modules

set as a data sample in order to predict the probability that a metamodel share commonalities with another one. To do that, we

one was intended to know what is the average of metamodels to which a metamodel share something. To do so, we execute a pair-wise analysis between all the metamodels and compute a matrix.

The answers. The experiments were conducted using a version of Puzzle implemented in Java. Further, Puzzle was installed in the Grid5000 Cloud, which is a cluster with more than 5000 cores from were we took XX dual-CPU Dell Blades with Intel Xeon X3470 CPUs running at 2.93GHz, with 16 threads per CPU, and CentOS v6. Each dual-CPU Dell Blade has 36GB of RAM.

5.3 Tool demonstration

The tooling supporting the approach presented in this paper is implemented on top of the Eclipse Modeling Framework. We provide a tool demonstration[4] that illustrates the use of our tool in both: the example presented in section 3 and the case study of the state machines presented in section 5.

6 Related work

7 Conclusions and Perspectives

The use of DSLs has demonstrated advantages in the software development process. However, building a DSL is a costly task that requires important engineering efforts. In this paper, we provide an approach for exploiting reuse during the

[4] Tool demonstration: `www.tooldemo.com`

construction of DSLs. We demonstrate that it is possible to partially automate the reuse process by identifying commonalities 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.

As future work, we plan to propose approaches to automatically build language product lines i.e., software product lines where the products are DSLs. The intention is to follow with the idea of automating the reuse process. This time, using ideas that facilitate the management of the variability existing among DSLs.

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; the VaryMDE project, a collaboration between Thales and INRIA; and the Cybersecurity project involving DGA (French department of defense) and INRIA.

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, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, June 2014.
7. B. Combemale, 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.

8. S. Cook. Separating concerns with domain specific languages. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 1–3. Springer Berlin Heidelberg, 2006.
9. 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.
10. 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.
11. 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.
12. 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.
13. D. Harel and H. Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In H. Ehrig, W. Damm, J. Desel, M. Grofle-Rhode, W. Reif, E. Schnieder, and E. Westk‰mper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.
14. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, Oct. 1996.
15. D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of “semantics"? *Computer*, 37(10):64–72, Oct 2004.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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.
22. M. Mernik. An object-oriented approach to language compositions for software language engineering. *J. Syst. Softw.*, 86(9):2451–2464, Sept. 2013.
23. 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.

24. 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.
25. A. Olson, T. Kieren, and S. Ludwig. Linking logo, levels and language in mathematics. *Educational Studies in Mathematics*, 18(4):359–370, 1987.
26. O. M. G. (OMG). Uml 2.4.1 superstructure specification, 2011.
27. 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.
28. E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.
29. 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.
30. 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.
31. 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.