Reverse-engineering reusable language modules from legacy domain-specific languages
David Méndez-Acuña, José Angel Galindo Duarte, Benoit Combemale, Arnaud Blouin, Benoit Baudry, Gurvan Le Guernic

To cite this version:
David Méndez-Acuña, José Angel Galindo Duarte, Benoit Combemale, Arnaud Blouin, Benoit Baudry, et al.. Reverse-engineering reusable language modules from legacy domain-specific languages. International Conference on Software Reuse, Jun 2016, Limassol, Cyprus. hal-01284816

HAL Id: hal-01284816
https://hal.archives-ouvertes.fr/hal-01284816
Submitted on 8 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Reverse-engineering reusable language modules from legacy domain-specific languages

1David Méndez-Acuña (✉), 1José A. Galindo, 1Benoit Combemale, 1Arnaud Blouin, 1Benoit Baudry, and 2Gurvan Le Guernic

1 INRIA and University of Rennes 1, France
2 DGA Maîtrise de l’Information, France
{david.mendez-acuna,jagalindo,benoit.combemale, arnaud.blouin,benoit.baudry}@inria.fr
gurvan.le-guernic@intradef.gouv.fr

Abstract. The use of domain-specific languages (DSLs) has become a successful technique in the development of complex systems. Nevertheless, 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 through the definition of language modules which can be later put together to build up new DSLs. Still, the identification and definition of language modules are complex and error-prone activities, 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 legacy DSLs; and ii) capitalize such potential reuse by extracting a set of reusable language modules with well defined interfaces that facilitate their assembly. We validate our approach by using realistic DSLs coming out from industrial case studies and obtained from public GitHub repositories.

1 Introduction

A domain-specific language (DSL) is a software language whose expressiveness is limited to a well-defined domain. A DSL offers the abstractions (a.k.a., language constructs) needed to describe an aspect of a system under construction. For example, we find DSLs to build graphical user interfaces [22] and to specify security policies [15]. The use of DSLs has become a successful technique to achieve separation of concerns in the development of complex systems [7].

Naturally, the adoption of such a language-oriented vision relies on the availability of the DSLs necessary to describe all the aspects of the system [3]. This implies the development of many DSLs, which is a challenging task due the specialized knowledge it demands. 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. In particular, there are
approaches that take ideas from Component-Based Software Engineering [4] in the construction of DSLs (e.g., [17,24]). Language constructs are grouped into interdependent language modules that can be later integrated as part of the specification of future DSLs. Current approaches for the modular development of DSLs are focused on providing foundations and tooling that allow language designers to specify dependencies among language modules as well as to provide the composition operators needed during the subsequent assembly process.

In practice, however, reuse not necessarily achieved through monolithic processes where language designers define language modules while trying to predict that they will be useful in future DSLs. Contrariwise, 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. Those functionalities can be extracted in reusable language modules. For example, many DSLs offer expression languages with simple imperative instructions, variables management, and mathematical operators. Xbase [1] is a successful experiment that shows that, using compatible tooling, such replicated functionality can be encapsulated and (re)used in different DSLs.

A major complexity of this reuse process is that both, the identification of replicated functionalities and the extraction of the corresponding language modules are manually-performed activities. Language designers must compare DSL specifications to identify replicated language constructs, and then, to perform a refactoring process to extract those replications in language modules. Due the large number of language constructs defined within a DSL, and the dependencies among them, this process is tedious and error-prone [9]. As a result, modularization approaches are often discarded, and non-systematic reuse practices such as simple copy&paste are still quite popular in DSLs development processes. This type of solutions produce many code clones within DSLs’ specifications thus replicating bugs and increasing maintenance costs [26].

In this paper, we propose the use of reverse-engineering techniques to automatically extract reusable language modules from a given set of legacy DSLs. To this end, we define some comparison operators that allow the identification of replicated language constructs. These operators take into account not only the names of the constructs but also the inter-constructs relationships and the semantics. Then, we extract replicated constructs as interdependent language modules whose dependencies are expressed through well-defined interfaces. Those language modules can be later assembled among them to build up new DSLs. The approach presented in this paper is implemented in a language workbench on top of the Eclipse Modeling Framework.

The validation of our approach is twofold. Firstly, we apply the reverse-engineering strategy to a case study, deeply explained by Crane et al. [8], and composed of a set of DSLs for finite state machines. Secondly, we explore public GitHub repositories in search of insights that indicate how common is the phenomenon of specification clones in DSLs development process.

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 a concrete development scenario. Section 4 describes the proposed approach. Section 5 presents the validation. Section 6 discusses the related work. 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 the rest of the paper.

**DSLs specification.** Like general purpose languages, domain specific languages are defined regarding three implementation concerns: abstract syntax, concrete syntax, and semantics [11]. The *abstract syntax* refers to the structure of the DSL expressed as the set of concepts that are relevant to the domain and the relationships among them. The *concrete syntax* relates language concepts to a set of symbols that facilitate the usage of the DSL. These representations are usually supported by editors acting as the user interface of the DSL. Finally, the *semantics* of a DSL assigns a precise meaning to each of its language constructs. More precisely, *static semantics* constrains the sets of valid programs while *dynamic semantics* specifies how they are evaluated.

**Technological space.** There are diverse technological spaces available for the implementation of the aforementioned concerns [18]. The abstract syntax can be specified using context-free grammars or metamodels. The concrete syntax can be either textual or graphical. The static semantics can be expressed through diverse constraint languages. Finally, the dynamic semantics can be defined operationally, denotationally, or axiomatically [20].

In this paper, we are interested in executable domain-specific modeling languages (xDsMLs) where the abstract syntax is specified by means of metamodels, and dynamic semantics is specified operationally as a set of *domain-specific actions* [5]. Domain-specific actions are weaved on the metaclasses of a metamodel [12]. The concrete syntax and static semantics are out of the scope of this paper.

**Example: A DSL for finite state machines.** Figure 1 shows a DSL for finite states machines. In that case, the metamodel that implements the abstract syntax contains three metaclasses: *StateMachine, State,* and *Transition.* There are some references among those metaclasses representing the relationships existing among the corresponding language constructs.

The domain-specific actions at the right of the Figure 1 introduce the operational semantics to the DSL. In this example, there is one domain-specific action for each metaclass. In executable metamodeling, the interactions among domain-specific actions can be internally specified in their implementation by means of the *interpreter pattern,* or externalized in a model of computation [5].
Abstract Syntax

StateMachine

State

Transition

Semantics

transitions 0..*
states 0..*
S1
T1
S2
T2
T3
T4
S3

void eval () {...}

Instance: executable model

Fig. 1: A simple DSL for finite state machines

3 Motivating scenario

Suppose a team of language designers working on the construction of the DSL for finite state machines presented in section 2. During that process, language designers implement the constructs typically required for expressing finite state machines: states, transitions, events, and so on. Besides, a constraint language that allows final users to express guards on the transitions should be provided, as well as an expression language for the specification of actions in the states.

After language designers release the DSL for state machines, they are required to build another DSL. The new DSL is intended to manipulate the traditional Logo turtle, which is often used in elementary schools for teaching the first foundations of programming [21]. 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 expression language 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 expression language they already defined for the state machine DSL. Because this expression language was not implemented separately from the DSL for state machines, the typical approach is to copy & paste its corresponding specification segment in the second DSL. In doing so, language designers introduce specification clones all along the project. This practice is repeated in the construction of each new DSL where some reuse is needed. For example, if our language designers team is required to build a third DSL such as a flowchart language that uses not only expressions but also constraints, they will (again) copy & paste the corresponding specification segments. After some iterations, we obtain a set of DSLs with many specification clones, which is quite expensive to maintain.
4 Proposed approach

We propose the use of reverse-engineering techniques to deal with the problem illustrated above. Our proposal, summarized in Figure 2, starts from a classical language development process where a team of language designers develops a set DSLs (a.k.a., the DSLs portfolio) introducing specification clones by copy&paste repeated constructs. This portfolio is the input of a reverse-engineering strategy to extract a set of reusable language modules. Those modules are useful for two purposes. First, they can be assembled to build a new version of the portfolio that does not contain specification clones, thus reducing maintenance costs. Second, they can be used in the construction of future DSLs. In that case, language designers might have to build new language modules.

4.1 Principles of reverse-engineering for language reuse

Our reverse-engineering strategy is based on five principles that will be introduced in this section. Then, we explain how we use those principles to extract a catalog of reusable language modules.

Principle 1: DSL specifications are comparable. Hence, specification clones can be automatically detected. Two DSL specifications can be compared each other. This comparison can be either coarse-grained indicating if the two specifications are equal regarding both syntax and semantics, or fine-grained detecting segments of the specifications that match. The latter approach permits to identify specification clones between two DSLs and supposes the comparison of each specification element. In the case of the technological space discussed in this paper, specification elements for the abstract syntax are metaclasses whereas specification elements for the semantics are domain-specific actions.

For the case of comparison of metaclasses, we need to take into account that a metaclass is specified by a name, a set of attributes, and a set of references
to other metaclasses. Two metaclasses are considered as equal (and so, they are clones) if all those elements match. Formally, comparison of metaclasses can be specified by the operator \( \equiv \).

\[
MC_A \equiv MC_B = \text{true} \implies
MC_A.\text{name} = MC_B.\text{name} \land
\forall a_1 \in MC_A.\text{attr} | (\exists a_2 \in MC_B.\text{attr} | a_1 = a_2) \land
\forall r_1 \in MC_A.\text{refs} | (\exists r_2 \in MC_B.\text{refs} | r_1 = r_2) \land
|MC_A.\text{attr}| = |MC_B.\text{attr}| \land |MC_A.\text{refs}| = |MC_B.\text{refs}|
\]

In turn, for the case of comparison for domain-specific actions we need to take into account that –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 implemented. Two domain-specific actions are equal if they have the same signature and body.

Whereas comparison of signatures can be performed by syntactic comparison of the signature elements, comparison of bodies can be arbitrary difficult. If we try to compare the behavior of the domain-specific actions, then we will have to address the semantic equivalence problem, which is known to be undecidable [16]. To address this issue, we conceive bodies comparison in terms of its abstract syntax tree as proposed by Biegel et al. [2]. In other words, to compare two bodies, we first parse them to extract their abstract syntax tree, and then we compare those trees. Note that this decision makes sense because we are detecting specification clones more than equivalent behavior. Formally, comparison of domain-specific actions (DSAs) is specified by the operator \( \simeq \).

\[
DSA_A \simeq DSA_B = \text{true} \implies
DSA_A.\text{name} = DSA_B.\text{name} \land
DSA_A.\text{returnType} = DSA_B.\text{returnType} \land
DSA_A.\text{visibility} = DSA_B.\text{visibility} \land
\forall p_1 \in DSA_A.\text{params} | (\exists p_2 \in DSA_B.\text{params} | p_1 = p_2) \land
|DSA_A.\text{params}| = |DSA_B.\text{params}| \land
DSA_A.AST = DSA_B.AST
\]

**Principle 2:** Specification clones are viewed as overlapping. If a DSL specification is viewed as sets of metaclasses and domain-specific actions, then specification clones can be viewed as intersections (a.k.a., overlapping) of those sets. Figure 3 illustrates this observation for the case of the motivation scenario introduced in Section 3. We use two Venn diagrams to represent both syntax and
semantic overlapping. In that case, the fact that the expression language is used in all the DSLs is represented by the intersection in the center of the diagram where the three sets overlap the metaclass Expression (and its domain-specific actions). In turn, the intersection between the state machines DSL and Logo shows that they overlap the metaclass Constraint that belongs to the constraint language. Note that the identification of such overlapping is only possible when there are comparison operators (principle 1) that formalize the notion of equality.

**Principle 3:** Breaking down overlapping produces reusable language modules. According to principle 2, overlapping between two DSLs implies the existence of repeated metaclasses/domain-specific actions (i.e., specification clones). Those repeated elements can be specified once and reused in the two DSLs [25, p. 60-61]. Hence, reusable language modules can be obtained by breaking-down the overlapping existing among DSL specifications as illustrated in Figure 4; each different intersection is encapsulated in a different language module.

**Principle 4:** Abstract syntax first, semantics afterwards. As aforementioned, the abstract syntax of a DSL specifies its structure in terms of metaclasses and relationships among them. Then, the domain-specific actions add
executability to the metaclasses. Hence, the abstract syntax is the backbone of the DSL specification, and so, the process of breaking down overlapping should be performed for the abstract syntax first. Afterwards, we can do the proper for the semantics. In doing so, we need to take into consideration the phenomenon of semantic variability. That is, two cloned metaclasses might have different domain-specific actions. That occurs when two DSLs share some syntax specification but differ in their semantics.

**Principle 5:** Metamodels are directed graphs. Hence, breaking down a metamodel is a graph partitioning problem. The metamodel that specifies the abstract syntax of a DSL can be viewed as a directed graph $G$.

$$G = \langle V, A \rangle$$

where:

- $V$: is the set of vertices each of which represents a metaclass.
- $A$: is the set of arcs each of which represents a relationships between two meta-classes (i.e., references, containments, and inheritances).

This observation is quite useful at the moment of breaking down a metamodel to satisfy the principle 4. Breaking down a metamodel can be viewed as a graph partitioning problem where the result is a finite set of subgraphs. Each subgraph represents the metamodel of a reusable language module.

### 4.2 Reverse-engineering process: The 5 principles in action

The reverse-engineering strategy to produce a catalog of reusable modules is illustrated in Figure 5. It is composed of two steps: identifying overlapping and breaking down.

![Fig. 5: Breaking down the input set by cutting overlapping](image-url)
Identifying overlapping: match and merge. To identify syntactic overlapping in a given set of DSLs, we start by producing a graph for each DSL according to the principle 5. Then, we identify specification clones (the matching phase) using the comparison operators defined in principle 1. After that, we have a set of graphs (one for each DSL) and a set of matching relationships among some of the vertex. At that point we can proceed to create the overlapping defined in principle 2. To this end, we merge the matched vertex as illustrated in the second square of Figure 5. This merging permits to remove cloned metaclasses.

To identify semantic overlapping, we check whether the domain-specific actions of the matched metaclasses are equal as well. If so, they can be considered as clones in the semantic specification, so there is semantic overlapping. In that case, these domain-specific actions are merged. If not all the domain-specific actions associated to the matched metaclasses are the same, different clusters of domain-specific actions are created, thus establishing semantic variation points.

Breaking down: cut and encapsulate. Once overlapping among the DSLs of the portfolio has been identified, we extract a set of reusable language modules. This process corresponds to break-down the graph produced in the last phase using 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 vertices 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 dependencies towards other language modules.

Dependencies between language modules can be viewed through the classical required and provided roles in components-based software development illustrated in Figure 6. 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. 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 that are supposed to be replaced by actual construct provided by another module(s).

![Fig. 6: Interfaces for language modules](image-url)
We use model types [23] 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.

Implementation. The approach presented in this paper is implemented in the Puzzle tool suite 3, which 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. The mapping between metaclasses and domain-specific actions are specified through the notion of aspect introduced by Kermeta [12] and Melange [9].

5 Evaluation

The evaluation of our approach is twofold. First, we evaluate the correctness of the approach using a test oracle that consists of a well-documented case study where we exactly know the existing overlapping among the involved DSLs. We execute the reverse-engineering on the case study, and we check that the produced language modules are consistent with the known overlapping. Second, we evaluate relevance of our proposal. More concretely, we use empirical data to demonstrate that the phenomenon of specification clones actually appears in DSLs that we obtain from public GitHub repositories.

5.1 Evaluating correctness: The state machines case study

Test oracle. To evaluate the correctness of our approach, we use the case study introduced by Crane et al. [8]. It is composed of three different DSLs for expressing state machines: UML state diagrams, Rhapsody, and Harel’s state charts. These three DSLs have some commonalities since they are intended to express the same formalism. For example, all of them provide basic concepts such as StateMachine, State, and Transition. According to the development scenario we address in this paper, these commonalities will be materialized as clones in the DSL specifications. However, not all those DSLs are exactly equal. They have both syntactic and semantic differences.

Syntactic differences are reified by the fact that not all the DSLs provide the same constructs. There are differences in the support for transition’s triggers and pseudostates. Whereas Rhapsody only supports atomic triggers, both Harel’s statecharts and UML provide support for composite triggers. In Harel’s statecharts 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, 3 Puzzle’s website: http://damende.github.io/puzzle/
Reverse-engineering reusable language modules from legacy DSLs

ShallowHistory, and Junction); there are two pseudostates i.e., DeepHistory and Choice that are only supported by UML. The Conditional pseudostate is only provided by Harel’s state charts. Figure 7 shows a table with the language constructs provided by each DSL.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rhapsody</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harel</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>● ● ● ● ● ● ●</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7: Oracle for evaluation of correctness

In turn, semantic differences are reified by the fact that not all the DSLs have the same behavior at execution time. For example, whereas Harel’s statecharts attend simultaneous events in parallel, both UML and Rhapsody follow the run to completion principle. So, simultaneous events are attended sequentially [8]. Consequently, not all the domain-specific actions are the same. In particular, the domain-specific actions eval() and step() in the StateMachine metaclass are different in each DSL.

Results. Figure 8 presents the results produced by Puzzle for the first part of the analysis: identification of overlapping. The figure shows the Venn diagrams for both syntactic and semantic overlapping. In the case of the syntactic overlapping, the cardinalities of the intersections in the Venn diagram match the test oracle. In turn, the domain-specific actions eval() and step() associated to the StateMachine metaclass are correctly identified as different in each DSL.

Figure 9 presents the results for the second part of the approach: breaking down overlapping. There is a language module that contains all the constructs shared by the three DSLs. That is, the constructs existing in the intersection Harel \ UML \ Rhapsody. Note that the behavioral differences are materialized by several implementations of the semantics, i.e., semantic variation points.

Also, other language modules encapsulate pseudostates and triggers separately. This is because pseudostates and triggers are supported differently in the DSLs, so they should be specified in different language modules. In this way, language designers can pick the desired constructs to build a particular DSL. Particularly, to obtain the Harel’s statecharts DSL, 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. The instructions to replicate this experiment are available online

4 Website for experiment 1: http://puzzlestatemachines.weebly.com/
5.2 Evaluating relevance: Are specification clones a real phenomenon in DSLs development processes?

Although our experience indicates that copy & paste is a real practice in language development processes so it is normal to find specification clones, we still need to verify that it is a phenomenon that appears in other development teams, and industrial contexts. To answer that question, we explored public GitHub repositories in search of DSLs that are built on the same technological space that we used in our approach. The intention is to confirm the existence of specification clones among those DSLs. The results are presented in this section, and all the data and tooling needed to replicate these experiments are available on-line.  

Website for experiment 2: http://empiricalpuzzle.weebly.com/
Data. We conducted an automatic search on GitHub repositories to find Ecore metamodels enriched with operational semantics written as Kermeta aspects in Xtend. As a result of this search, we obtained a data set composed of 2423 metamodels. Nevertheless, because Kermeta and its implementation in Xtend is a quite recent idea, we found very few data for the semantics part. Besides, all of them have been developed in our research team. We decided to conduct the analysis only in the metamodels since we consider that detection of specification clones at the level of the abstract syntax can give us a good insight about the existence of copy&paste practices in DSLs development processes.

Experiment. To identify specification clones in the metamodels from our data set, we performed a pair-wise comparison among all the metamodels (w.r.t. the $\equiv$ operator introduced in section 4). Then, we compute the matrix $O(i,j)$ where each cell $(i,j)$ contains the number of cloned metaclasses between the metamodels $i$ and $j$. $O(i,j) = 0$ means that there is no cloned metaclasses between the metamodels $i$ and $j$. We are interested in the cells $(i,j)$ such that $O(i,j) \neq 0$ and $i \neq j$. Those cells correspond to a pair of metamodels with some specification clones. Then, we analyze the matrix with two questions in mind: (1) how many metamodels have some specification clones among them?; and (2) how many classes are cloned from one metamodel to the other?.

Results. Figure 10 shows two charts with the results to the experiment. The chart at the left is intended to answer the first question. In this chart, each entry $x$ of the horizontal axis represents one metamodel of the data set. In turn, the vertical axis i.e., $y(x)$ shows the amount of metamodels with some specification clones for $x$. Formally, $y(x) = (\{k \mid 0 \geq k \geq 2423 \land O(x,k) > 0\} : 1)$. For example, the metamodel with ID 1.053 has some specification clones with 272 metamodels. Note that each point located up the zero line of the vertical axis represents a metamodel with some specification clones with one or more metamodels, thus suggesting that specification clones is a real phenomenon.

The chart at the right of the Figure 10 is intended to answer the second question. In this chart, each entry $x$ of the vertical axis represents one metamodel of the data set. The vertical axis i.e., $z(x)$ shows the average amount of cloned classes for $x$. Formally, $z(x) = 1/y(x) * (\{k \mid 0 \geq k \geq 2423 : O(x,k)\})$. For example, the metamodel 1.928 shares, in average, 99.4 metaclasses with other metamodels. Note that there is an important amount of metamodels whose average overlapping size is between 0 and 100 metaclasses. Note also that there are four metamodels that share about 600 metaclasses. This case corresponds to a set of different versions of a metamodel for UML.

6 Related work

Reuse in DSLs development processes. The research community in software language engineering has previously studied mechanisms to leverage reuse in the development of DSLs. In this context, languages modularization is probably
the most popular approach. We can find approaches supporting complex modularization scenarios such languages extension (e.g., [10]) applicable to diverse technological spaces such as metamodeling [24] or attribute grammars [17].

Another approach to leverage reuse in DSLs is the definition of domain-specific metamodeling languages [14,26]. The idea is to define abstract language constructs that can be useful in several DSLs, and to provide mechanisms to specialize such abstract constructs to particular application contexts. For example, a language designer can define a DSL for finite state machines with an abstract behavior, and adapt it to several DSLs according to the needs of the final users.

More recent approaches are focused on facilitating the reuse process itself. For instance, Melange [9] is a tool-supported language 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 into new developments.

The main contribution of our approach is the advance towards the automation of the reuse process. We show that, under certain conditions, the process can be automated through reverse-engineering techniques. We exploit the reuse opportunities in the form of specification clones, thus reducing maintenance costs and facilitating the construction of future DSLs.

**Déjà vu in object-oriented programming?** There is a symbiosis between executable metamodeling and object-oriented programming. Besides, there are several approaches intended to extract reusable modules from legacy object-oriented software systems (e.g., [6,19]). Our approach, however, should not be viewed as yet another technique to extract reusable object-oriented components. Rather, we propose to take advantage of such symbiosis and use advances achieved in object-oriented programming to solve problems that also occur during the development of executable DSL. Indeed, there is still large room to exploit those ideas to improve reverse-engineering techniques in DSLs. In doing so, the central issue to consider is the separation of concerns in DSL specifications. That is, the fact that the syntax and semantics of the DSLs are usually specified separately, in many cases, using different metalanguages.
7 Conclusion

In this paper, we presented an approach to exploit reuse during the construction of DSLs. We show that it is possible to automate the reuse process by identifying specification clones in DSLs and automatically extracting reusable language modules that can be later used in the construction of new DSLs. We evaluated our approach in an industrial case study, and we demonstrate that there is an important amount of potential reuse in DSLs we obtain from public repositories.

Acknowledgments

This work is supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), the bilateral collaboration VaryMDE between Inria and Thales, and the bilateral collaboration FPML between Inria and DGA.

References