Softw Syst Model (2015) 14:889–904
DOI 10.1007/s10270-013-0352-6

REGULAR PAPER

Corpus-based analysis of domain-specific languages

Robert Tairas · Jordi Cabot

Received: 24 March 2012 / Revised: 25 December 2012 / Accepted: 18 April 2013 / Published online: 9 June 2013

Abstract As more domain-specific languages (DSLs) are designed and developed, the need to evaluate these languages becomes an essential part of the overall DSL life cycle. Corpus-based analysis can serve as an evaluation mechanism to identify characteristics of the language after it has been deployed by looking at how end-users employ it in practice. This analysis that is based on actual usage of the language brings a new perspective which can be considered by a language engineer when working toward improving the language. In this paper, we describe our utilization of corpus-based analysis techniques and exemplify them on the evaluation of the Puppet and ATL DSLs. We also outline an Eclipse plug-in, which is a generic corpus-based DSL analysis tool that can accommodate the evaluation of different DSLs.

Keywords Domain-specific languages · DSL · Corpus · Analysis · ATL · Puppet

1 Introduction

Domain-specific languages (DSLs) provide their users with a more expressive and easier-to-use language that is targeted for a particular domain than what general-purpose languages (GPLs) can offer [23]. These DSLs are built by language engineers who are tasked with developing languages that represent domain-specific concepts in an effective way. The growing popularity of language workbenches (e.g., Xtext[1]) has provided assistance for language engineers to develop DSLs and their supporting infrastructure in a more automated way. Hence, such language workbenches support the potential for more DSLs to be developed in the near future.

After the initial development of a DSL, the language engineer should monitor several characteristics of the DSL in order to detect possible evolution opportunities that can further improve the language in future versions. The identification of such characteristics can be done through a post-deployment analysis of the DSL. However, it has been observed that this type of evaluation on DSLs has received less focus compared to the actual development of DSLs [9].

In this paper, we focus on analysis techniques based on the evaluation of the *corpus* of a DSL. A DSL corpus in this case consists of instances of a DSL reflecting its actual usage by end-users. By evaluating how a DSL was used by its users, we seek to determine characteristics that can help the language engineer to evolve his or her language. We believe corpus-based analysis can provide information regarding a DSL that can complement other analysis techniques focusing on other aspects of the DSL. The analysis techniques described in this paper can be applied to DSLs that offer a textual notation, which can also include DSLs that have a primary concrete syntax that is not textual, but offer a secondary or intermediate textual representation.

The remainder of this paper is structured as follows: the next section describes the motivation and contributions of our work in more detail. Section 3 introduces the DSL corpora that are used with the analysis techniques that are described in Sect. 4. Section 5 summarizes observations from the use of the analysis techniques, and Sect. 6 describes threats to validity related to our evaluations. Section 7 describes our

Communicated by Prof. Tony Clark.

R. Tairas (✉) · J. Cabot
AtlanMod, École des Mines de Nantes - INRIA - LINA, Nantes, France
e-mail: ratairas@gmail.com

[1] http://www.eclipse.org/Xtext.

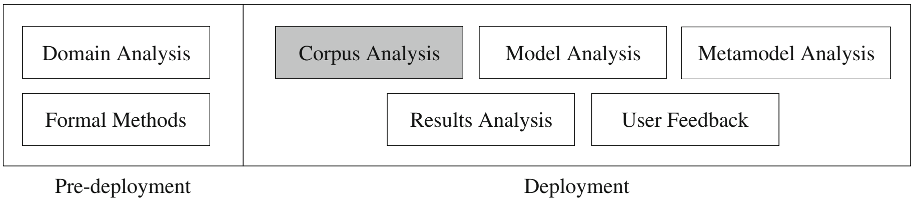


Fig. 1 DSL analysis activities

Eclipse plug-in that generalizes these analysis techniques. Section 8 offers related work, and Sect. 9 concludes the paper and summarizes future work.

2 Motivation

DSL analysis can be performed at various stages of the language's life cycle as can be seen in Fig. 1. Pre-deployment analysis activities can include domain analysis during the initial design of the language [23] and the utilization of formal methods to verify, for example, the satisfiability of the language [5]. After the DSL has been deployed, analysis can be performed on the metamodel representing the DSL [26], on individual models of the DSL [24] and on the results of the DSL (i.e., its run-time execution [12] or its final generated artifact [1]). In addition, analysis of the language can be performed by considering user feedback on the usage of the DSL [7,14,16].

This paper follows a different approach that focuses on DSL corpus analysis (highlighted in Fig. 1) after the language has been deployed and once a reasonable corpus of DSL instances from users becomes available. Evaluating a language based on how it is actually used can yield interesting characteristics about the language. For example, van Amstel et al. [31] identified some tedious coding related to variable initialization in the DSL that they evaluated, which prompted the need to modify the language to eliminate the tedious task. The identification came from the authors manually looking at the instances or corpus of the DSL. Such a scenario motivates the need to provide a mechanism in which language engineers can obtain characteristics of the language by performing analysis on the available corpus of their DSL.

It should be noted that corpus analysis in GPLs has received much focus from the research community. While corpus analysis activities for DSLs resemble those for GPLs, we describe the following characteristics that highlight differences between GPL and DSL corpus analysis in terms of importance and coverage and justify the development of specific techniques for the case of DSLs.

Non-programmer—As many DSLs are to be used by individuals with no computer programming experience, these individuals may not be well suited to express what type of change or evolution is needed to improve a DSL. In contrast, GPLs are typically used by programmers or software engineers who in many cases have adequate knowledge about programming languages. Therefore, while for GPLs we can rely on user feedback to get informative suggestions regarding the improvement of the language, in general we cannot expect the same from DSL users. For DSLs, finding other means of evaluating the language in addition to user feedback can further assist in the evolution of a language. In this case, the role of corpus analysis becomes more important to identify implicit characteristics of the language.

Smaller user base—Typically, because of the specificity of a DSL, its user base will be small compared to that of a GPL. This also reduces the opportunities of a reactive user feedback and reinforces the need for proactive techniques that anonymously analyze the properties of the DSL.

Differing goals and scope of analysis—The analysis of GPLs as compared to DSLs can be based on differing goals. For example, the elimination of code clones (i.e., duplicated sections of code) in GPLs can be done by modularizing the code, whereas for DSLs, a new language construct or metamodel element could be considered. The scope of analysis can also differ. For example, analyzing every metamodel element of a DSL can be beneficial because of what each element represents in the domain. In contrast, analyzing every language construct in a GPL can potentially be overkill as some constructs are less important and need to be grouped with other constructs first to provide better analysis targets.

The need for generalized techniques—Due to the specific nature of a DSL, the number of DSLs in existence can be potentially much larger compared to that of GPLs. Again, because of the growing popularity of language workbenches, the effort it takes to develop a DSL is reduced and hence provides the potential for more DSLs to be generated. GPL corpus analysis has mainly been focused on popular languages such as C, C++, and Java. The complexity and sheer size of these languages allow for many opportunities to analyze the languages from different angles and justify the development of specific techniques only useful for a single GPL. By comparison, DSLs can be smaller in size and thus developing analysis tools for a specific language becomes less efficient as acknowledged by Monperrus et al. [24]. Hence, when developing analysis tools including those that analyze the corpus of a DSL, a generalized mechanism is needed to allow for a large coverage of languages that can be supported by the tools.

In this paper, we seek to utilize corpus-based analysis techniques to determine characteristics of DSLs based on the actual usage of the languages. The following are the main contributions of this paper:

- The proposal of a set of corpus-based analysis techniques for DSLs and their utilization for the evaluation of two DSLs. The techniques were selected, because of their potential to identify properties of DSLs for the language engineer and because they are generic in a sense that they can be applied to many DSLs.
- The description of an Eclipse plug-in offering corpus-based analysis techniques that is applicable for multiple DSLs. This generic method works on the EMF-based representation of DSL instances and their corresponding metamodels.

3 Corpus information

Before describing the analysis techniques we investigated, we first introduce the DSL corpora used as illustrative examples in the remainder of the paper. The chosen DSLs were Puppet[2] and ATL [18]. The Puppet DSL is part of a server automation tool that is used for expressing system configurations. The ATL DSL is used in the context of model-driven engineering (MDE) to express transformations of models that conform to a source metamodel to models that conform to a target metamodel. These DSLs were selected mainly because of the public availability of a corpus for each language consisting of a considerable number of models that could be used to provide significant results and interpretations of the characteristics of both DSLs when applying on them the analysis techniques described in the next section.

The corpus for the analysis of Puppet comes from PuppetForge,[3] a publicly accessible Web site that allows Puppet users to share and download modules. Modules represent projects containing multiple Puppet models related to the configuration of certain aspects of a server. The corpus was downloaded through Geppetto,[4] an Xtext-based IDE for Puppet. At the time of download, 176 modules consisting of 728 Puppet models were retrieved. During the initial runs of clone detection related to the clone analysis technique that is described in Sect. 4.3, it was noticed that an exact copy of one module called “lab42-activemq” existed in a sub-directory of another module called “puppetlabs-activemq.” We removed the duplicate copy of the “lab42-activemq” module in the “puppetlabs-activemq” module. This omission reduced the total model count from 728 to 706 models.

The corpus for the analysis of ATL was taken from the ATL transformation zoo,[5] a publicly accessible Web site that lists transformation scenarios that have been contributed by ATL users. The site consists of about 100 transformation scenarios, where each scenario can contain multiple ATL transformation models, as some scenarios require intermediary steps in their processes. Hence, more than 200 ATL transformation models were available among these scenarios. However, only 189 models were used in our analysis, because upon manual observation, several of the models in the zoo were near exact duplicates of each other similar to the case of the duplicate Puppet module.

The corpora of Puppet and ATL differ in terms of model sizes. Out of the 706 models from the Puppet corpus, the largest model contains 2,466 model elements and the average size of the models is 79. In comparison, out of the models from the ATL corpus, the largest model contains 12,500 model elements and the average size of the models is 772.

4 Corpus-based analysis techniques

In the following subsections, we will propose the utilization of several corpus-based DSL analysis techniques. Although the evaluation is performed on a corpus consisting of instances represented in the concrete syntax of the language, the evaluation is performed at the abstract syntax level, i.e., our analysis is geared toward the understanding of characteristics of the DSL’s metamodel elements.

The techniques that will be applied are: *instance analysis*, which seeks to identify the usage characteristics of metamodel elements, *relationship analysis*, which seeks to identify relationship characteristics among metamodel elements, and *clone analysis*, which seeks to identify duplicate usage of a collection or sequence of metamodel elements in the language. We consider these techniques to form an important group of techniques to assist the language engineer of a DSL to identify useful characteristics of the language. These techniques are generic enough to be applied to many DSLs.

In all three analysis techniques, we evaluated the EMF model representations of the DSL instances in their respective corpora to associate metamodel elements with their usages in each model. For Puppet, the EMF models were obtained from the in-memory representation of Puppet files in Geppetto. This is possible as Xtext, which Geppetto is based on, represents a DSL instance or file as an EMF model. For ATL, we obtained the EMF models by injecting ATL files as models using the ATL IDE.[6]

[2] http://docs.puppetlabs.com/learning.

[3] http://forge.puppetlabs.com.

[4] http://cloudsmith.github.com/geppetto.

[5] http://www.eclipse.org/atl/atlTransformations.

[6] http://www.eclipse.org/atl.

4.1 Instance analysis

Instance analysis considers the extent of the usage of a metamodel element as evidenced in the corpus of a DSL. This basically implies counting the number of times an element is used in the corpus. Adapting the evaluation of instances to all metamodel elements in a DSL is feasible, because in contrast with GPLs, the number of DSL primitives tends to be much smaller. In addition, these elements in many cases represent a specific concept in the domain, hence this counting for all elements is meaningful and the identification of instances of the use of all elements can potentially show how frequent a domain concept is being represented in the DSL.

The simple statistic of counting the number of times a metamodel element is used in the corpus can suggest the *popularity* or lack there of for the actual use of the element by DSL users. From a language improvement perspective, the existence of several rarely used elements could potentially signal a language that was over-developed and that could be pruned by removing the unused elements. In contrast, the high usage of an element can potentially signal the need to focus future language improvements around that element.

More specifically, we consider two metrics for instance analysis. The first is the number of times a metamodel element is used throughout the corpus of a DSL. This provides a general count of the usage of metamodel elements in the corpus. A second metric is to count in how many models a metamodel element is used at least once. In this case, we can see how distributed the usage of the a metamodel element is among the models in the corpus.

The remainder of this subsection reports on the instance analysis of Puppet and ATL and shows how the metrics computed were useful to uncover interesting data as confirmed by the feedback provided by individuals who work closely with these languages. For Puppet, the instance analysis revealed characteristics related to the use of a newly supported feature and the varied use of interpolated strings. For ATL, the analysis revealed the extent of the use of imperative features in the mainly declarative ATL language.

4.1.1 Puppet

Tables 1 and 2 document usage of Puppet metamodel[7] elements. Table 1 documents each usage of a metamodel element, whereas Table 2 documents in how many Puppet models a metamodel element was used at least once and subsequently the percentage of models in the corpus that the element is used in. The elements followed by a (*) denote abstract elements in the metamodel.

As can be seen in both tables, most elements were used in the corpus. Abstract elements were correctly unused, but still a few elements were not used at all. It could be the case that these elements represent a specific concept that is very rarely used in the Puppet language. In the extreme case, they could signal the non-use of such elements and the need to decide whether they should be supported in future versions of the language.

A separate observation considers the use of the *IfExpression* element. In Table 2, out of 706 Puppet models, 89 contained *IfExpression*. Half of the models (i.e., 42) contained *ElseExpression*. However, only two models contained *ElseIfExpression*. *ElseIfExpression* was not initially implemented even though it was part of the language definition.[8] The feature was eventually supported, but as of our download of the modules from PuppetForge, the usage of *ElseIfExpression* is still very limited. This observation could signal that in fact adding this construct was not really necessary and, if the situation does not change, could be removed if later on it is decided to simplify the language.

The results of our Puppet instance analysis were forwarded to the developer of Geppetto. The developer suggested a customized query to determine the pattern of usage of text and interpolated variables. In Puppet, variable names can be interpolated within strings. This introduces several options of interpolating variables in strings, which in some cases can be inefficient. For example, using curly brackets with variables reduces ambiguities of identifying variables. In order to determine the dominant practice of variable interpolation, more detailed instance analysis on the metamodel elements associated with strings values and variables can be performed. In this case, the initial instance analysis results became a stepping stone for identifying more specialized queries on a DSL that can provide further details on the use of the language.

4.1.2 ATL

Table 3 documents the total usage of each ATL metamodel[7] element, whereas Table 4 documents in how many and the percentage of ATL models a metamodel element was used at least once. Again, the elements followed by a (*) denote abstract elements in the metamodel. As our research group is the developer of ATL, we were able to ask team members who are experts in ATL to evaluate the metamodel element instance data. ATL is primarily a declarative language, but in certain situations allows for imperative coding. Although supported, imperative style coding is discouraged in ATL. Because of this, the developer was encouraged to see that the metamodel elements associated with the imperative part of ATL was not used much (i.e., *CalledRule* and its associ-

[7] https://code.google.com/a/eclipselabs.org/p/dsl-analysis/wiki/Metamodels.

[8] http://projects.puppetlabs.com/issues/2713.

Table 1 Puppet metamodel element usage (all instances)

Name	Total	Name	Total	Name	Total
VerbatimTE	10,278	DefinitionArgumentList	230	MatchingExpression	3
LiteralNameOrReference	8,378	Definition	210	ElseIfExpression	2
DoubleQuotedString	7,280	CaseExpression	186	ExprList	1
AttributeDefinition	6,169	EqualityExpression	185	UnquotedString	1
VariableExpression	3,106	ElseExpression	96	AppendExpression	0
ResourceBody	2,132	ParenthesisedExpression	89	AttributeOperation*	0
AttributeOperations	1,964	LiteralUndef	65	BinaryExpression*	0
ResourceExpression	1,785	VirtualNameOrReference	57	BinaryOpExpression*	0
VariableTE	1,676	ImportExpression	44	Expression	0
AtExpression	1,476	LiteralRegex	39	ExpressionBlock*	0
SingleQuotedString	1,368	CollectExpression	34	ICollectQuery*	0
ExpressionTE	1,304	RelationshipExpression	33	InterpolatedVariable	0
SelectorEntry	1,116	VirtualCollectQuery	21	IQuotedString*	0
AssignmentExpression	948	OrExpression	20	LiteralExpression*	0
DefinitionArgument	795	UnaryNotExpression	16	LiteralName	0
LiteralBoolean	733	AndExpression	14	MultiplicativeExpression	0
PuppetManifest	706	AttributeAddition	13	ParameterizedExpression*	0
LiteralDefault	639	ExportedCollectQuery	13	ShiftExpression	0
SelectorExpression	557	HashEntry	13	StringExpression*	0
FunctionCall	538	NodeDefinition	11	TextExpression*	0
HostClassDefinition	507	LiteralHash	8	UnaryExpression*	0
LiteralList	373	AdditiveExpression	4	UnaryMinusExpression	0
Case	367	InExpression	4		
IfExpression	261	RelationalExpression	4		

ated *ActionBlock* were used only 78 and 135 times as seen in Table 3). Furthermore, it can be seen that only 25 out of 189 models evaluated used these elements as seen in Table 4, which also shows only a limited number of transformations that used these elements. These data can be taken into consideration in future changes to ATL in terms how much support for imperative style coding should be continued.

4.2 Relationship analysis

Instance analysis is mainly concerned with the usage of individual elements within the corpus. In contrast, in relationship analysis, we consider how certain elements are present together or are grouped together based on a particular criterion to determine interesting relations among two or more elements. It should be noted that in many cases two or more metamodel elements will always be related, because together they form a complete construct in the language. For example, an *ElseExpression* will always be associated with an *IfExpression*. We would like to identify more non-common relationships among the metamodel elements. Such relationships could suggest, for example, a sub-language within the DSL, because of the strong relationships among a group of metamodel elements. This could help us realize that our DSL needs to be further decomposed in order to make sure it is really domain-specific.

Clustering is a technique originating from the field of data mining that can be used to associate two or more elements [13]. One type of clustering evaluates distance values between elements to determine a grouping or cluster of elements that are considered “related.” Elements that have a closer distance value could be considered to be more related with each other. In our case, we adapt the clustering technique such that the distance values between metamodel elements are determined by counting the number of times a pair of metamodel elements is used in the same “instance.” We consider an instance as a model; hence, when a pair of metamodel elements is used in the same model, then we increment the count for that element pair. For example, consider metamodel elements A, B, and C and models X, Y, and Z. If models X and Y both contain metamodel elements B and C, and model Z contains metamodel elements A and B, then Table 5 shows a matrix that represents the co-occurrences among the metamodel elements. In the matrix in Table 5, each metamodel element is represented by a row and column. The cells contain the number of times one metamodel element was associ-

Table 2 Puppet metamodel element usage (model count)

Name	Total (%)	Name	Total (%)	Name	Total (%)
PuppetManifest	706 (100 %)	SelectorEntry	96 (14 %)	MatchingExpression	2 (<1 %)
LiteralNameOrReference	657 (93 %)	SelectorExpression	96 (14 %)	RelationalExpression	2 (<1 %)
DoubleQuotedString	481 (68 %)	IfExpression	89 (13 %)	ExprList	1 (<1 %)
VerbatimTE	481 (68 %)	EqualityExpression	59 (8 %)	UnquotedString	1 (<1 %)
ResourceBody	477 (68 %)	ElseExpression	42 (6 %)	AppendExpression	0 (0 %)
ResourceExpression	477 (68 %)	ParenthesisedExpression	28 (4 %)	AttributeOperation*	0 (0 %)
AttributeDefinition	465 (66 %)	ImportExpression	22 (3 %)	BinaryExpression*	0 (0 %)
AttributeOperations	465 (66 %)	LiteralUndef	22 (3 %)	BinaryOpExpression*	0 (0 %)
HostClassDefinition	409 (58 %)	OrExpression	15 (2 %)	Expression	0 (0 %)
AtExpression	295 (42 %)	VirtualNameOrReference	15 (2 %)	ExpressionBlock*	0 (0 %)
VariableExpression	290 (41 %)	CollectExpression	14 (2 %)	ICollectQuery*	0 (0 %)
FunctionCall	204 (29 %)	LiteralRegex	14 (2 %)	InterpolatedVariable	0 (0 %)
AssignmentExpression	181 (26 %)	NodeDefinition	11 (2 %)	IQuotedString*	0 (0 %)
LiteralBoolean	178 (25 %)	ExportedCollectQuery	10 (1 %)	LiteralExpression*	0 (0 %)
SingleQuotedString	175 (25 %)	RelationshipExpression	10 (1 %)	LiteralName	0 (0 %)
LiteralList	173 (25 %)	UnaryNotExpression	8 (1 %)	MultiplicativeExpression	0 (0 %)
ExpressionTE	171 (24 %)	AndExpression	5 (1 %)	ParameterizedExpression*	0 (0 %)
DefinitionArgumentList	159 (23 %)	AttributeAddition	5 (1 %)	ShiftExpression	0 (0 %)
DefinitionArgument	153 (22 %)	InExpression	4 (1 %)	StringExpression*	0 (0 %)
VariableTE	151 (21 %)	VirtualCollectQuery	4 (1 %)	TextExpression*	0 (0 %)
LiteralDefault	148 (21 %)	AdditiveExpression	3 (<1 %)	UnaryExpression*	0 (0 %)
Definition	135 (19 %)	ElseIfExpression	2 (<1 %)	UnaryMinusExpression	0 (0 %)
Case	116 (16 %)	HashEntry	2 (<1 %)		
CaseExpression	116 (16 %)	LiteralHash	2 (<1 %)		

ated with another metamodel element. This value is what we consider as the ”distance” between two metamodel elements (i.e., the larger the number, the closer the distance between the elements).

We record all pairwise instances of metamodel elements in each model of the corpus. For example, Tables 6 and 7 display the top pairwise relationships for Puppet and ATL metamodel elements, respectively. In the tables and in the clustering processes of the DSLs that we evaluated, the pairwise relationships involving the *PuppetManifest* element for Puppet and the *OclModel*, *OclModelElement*, *VariableExp*, and *NavigationOrAttributeCallExp* elements for ATL were excluded, because these elements appeared in all models (i.e., as seen in Tables 2, 4, respectively). This was the only consideration of element relationships that was performed based on manual observation. No semantic-based relationships were specifically considered at this point.

We use a stand-alone clustering tool called gCluto[9] to perform agglomerative clustering in which elements are clustered until a predefined number of clusters has been reached. The input of the clustering mechanism is the pairwise relationship counts as the distance values between a pair of elements. Since the clustering process considers pairs with smaller distance values to be more related, we need to calculate the inverse of each value.

The evaluation of some clusters revealed relationships of metamodel elements of interest, such as the related use of two types of switch statements in Puppet. Other clusters revealed no relationship of interest even after further manual analysis. Both types of observations are described in the remainder of this subsection. The elements in the described clusters are not fully independent elements, in that in some cases they are closely located within the metamodel structure. However, their relationships are not forced due to structural constraints of the metamodel. It should be noted that large models could potentially influence the results of this analysis, because these models represent a large number of recorded co-occurrences. However, these models represent a way of using the language that should be included in the results.

4.2.1 Puppet

Figure 2 shows a dendrogram of 10 clusters of Puppet metamodel elements. In cluster no. 3, *SelectorEntry*,

[9] http://glaros.dtc.umn.edu/gkhome/cluto/gcluto/overview.

Table 3 ATL metamodel element usage (all instances)

Name	Total	Name	Total	Name	Total
VariableExp	25,251	IfStat	503	Library	13
OclModelElement	18,021	Parameter	403	OrderedSetType	8
NavigationOrAttributeCallExp	16,231	LetExp	379	Query	8
Binding	14,105	Attribute	316	OrderedSetExp	3
OclModel	10,932	IntegerType	273	BagExp	0
OperationCallExp	7,792	RealType	257	BagType	0
StringExp	7,697	BooleanType	217	CollectionExp*	0
OperatorCallExp	5,613	SetType	209	CollectionType*	0
VariableDeclaration	4,940	ExpressionStat	202	DerivedInPatternElement	0
SimpleOutPatternElement	4,785	OclUndefinedExp	199	Element	0
CollectionOperationCallExp	3,287	IterateExp	169	InPatternElement*	0
Iterator	2,257	Module	168	IterateInPatternElement	0
OutPattern	2,006	LazyMatchedRule	143	LoopExp	0
SimpleInPatternElement	1,960	RuleVariableDeclaration	142	ModuleElement*	0
IteratorExp	1,956	TupleTypeAttribute	141	NumericExp*	0
InPattern	1,940	ActionBlock	135	NumericType*	0
MatchedRule	1,797	ForEachOutPatternElement	115	OclExpression*	0
IfExp	1,484	SetExp	109	OclFeature*	0
StringType	1,132	CalledRule	78	OclType*	0
SequenceExp	1,106	TuplePart	77	OutPatternElement*	0
Helper	1,021	TupleType	74	PatternElement*	0
OclFeatureDefinition	1,021	RealExp	68	Primitive*	0
IntegerExp	974	MapElement	55	PrimitiveExp*	0
BooleanExp	717	OclAnyType	55	PropertyCallExp*	0
Operation	702	MapType	50	Rule*	0
OclContextDefinition	687	LibraryRef	36	Statement*	0
BindingStat	681	TupleExp	29	Unit	0
SequenceType	670	MapExp	28		
EnumLiteralExp	535	ForStat	17		

SelectorExpression, *Case*, and *CaseExpression* are clustered together. *SelectorExpression* and *CaseExpression* function similar to the *Switch-statement* in Java. The difference between *SelectorExpression* and *CaseExpression* is that the former returns a value, while the latter does not. Based on Table 2, *CaseExpression* occurs in 116 models and *SelectorExpression* in 96 models. If a large number of models contained both elements or one of them was by far most common than the other, it could suggest the need to consolidate their functionalities. A deeper analysis revealed that only 36 models contained both elements, and hence, not many models contained both metamodel elements. From a different angle, in Table 1, *SelectorExpression* was used 557 times compared to 186 for *CaseExpression*. This could support consolidating the elements as one is used much more than the other in all models.

4.2.2 ATL

Figure 3 shows a dendrogram of 10 clusters of ATL metamodel elements. Metamodel elements representing the different types used in ATL are clustered together (i.e., ordered sets in cluster no. 2, tuples in cluster no. 4, and maps in cluster no. 8). In addition, a grouping of elements used together can also be seen. The elements representing the core functionality of ATL can be seen in the grouping in the last part of cluster no. 10. *Module* elements consist of *InPattern* elements representing the source models and *OutPattern* elements representing the target models. *Binding* elements define the transformation between the source and target models. In addition, elements related to the imperative part of ATL can also be seen in cluster 3, where *CalledRule* and *ActionBlock* elements are associated with the imperative expression *BindingStat*, *ExpressionStat*, and *IfStat*.

Table 4 ATL metamodel element usage (model count)

Name	Total (%)	Name	Total (%)	Name	Total (%)
OclModel	188 (99 %)	BooleanExp	78 (41 %)	Query	8 (4 %)
OclModelElement	188 (99 %)	LetExp	75 (40 %)	MapElement	6 (3 %)
VariableExp	188 (99 %)	IterateExp	65 (34 %)	OrderedSetExp	3 (2 %)
NavigationOrAttributeCallExp	187 (99 %)	BooleanType	59 (31 %)	OrderedSetType	2 (1 %)
OperationCallExp*	176 (93 %)	EnumLiteralExp	44 (23 %)	BagExp	0 (0 %)
Module	168 (89 %)	IntegerType	41 (22 %)	BagType	0 (0 %)
Binding	167 (88 %)	LazyMatchedRule	39 (21 %)	CollectionExp*	0 (0 %)
OutPattern	167 (88 %)	RuleVariableDeclaration	38 (20 %)	CollectionType*	0 (0 %)
SimpleOutPatternElement	167 (88 %)	OclUndefinedExp	37 (20 %)	DerivedInPatternElement	0 (0 %)
InPattern	165 (87 %)	SetType	30 (16 %)	Element	0 (0 %)
SimpleInPatternElement	165 (87 %)	LibraryRef	28 (15 %)	InPatternElement*	0 (0 %)
MatchedRule	162 (86 %)	ActionBlock	25 (13 %)	IterateInPatternElement	0 (0 %)
OperatorCallExp	160 (85 %)	CalledRule	25 (13 %)	LoopExp	0 (0 %)
CollectionOperationCallExp	159 (84 %)	ForEachOutPatternElement	24 (13 %)	ModuleElement*	0 (0 %)
VariableDeclaration	159 (84 %)	SetExp	22 (12 %)	NumericExp*	0 (0 %)
Iterator	158 (84 %)	BindingStat	19 (10 %)	NumericType*	0 (0 %)
StringExp	155 (82 %)	ExpressionStat	18 (10 %)	OclExpression*	0 (0 %)
IteratorExp	146 (77 %)	MapExp	17 (9 %)	OclFeature*	0 (0 %)
Helper	144 (76 %)	MapType	17 (9 %)	OclType*	0 (0 %)
OclFeatureDefinition	144 (76 %)	IfStat	15 (8 %)	OutPatternElement*	0 (0 %)
IfExp	129 (68 %)	RealType	15 (8 %)	PatternElement*	0 (0 %)
OclContextDefinition	116 (61 %)	Library	13 (7 %)	Primitive*	0 (0 %)
Operation	115 (61 %)	OclAnyType	10 (5 %)	PrimitiveExp*	0 (0 %)
SequenceExp	103 (54 %)	RealExp	10 (5 %)	PropertyCallExp*	0 (0 %)
IntegerExp	101 (53 %)	ForStat	9 (5 %)	Rule*	0 (0 %)
StringType	101 (53 %)	TupleExp	9 (5 %)	Statement*	0 (0 %)
SequenceType	95 (50 %)	TuplePart	9 (5 %)	Unit*	0 (0 %)
Parameter	82 (43 %)	TupleType	9 (5 %)		
Attribute	80 (42 %)	TupleTypeAttribute	9 (5 %)		

Table 5 Sample co-occurrence matrix

	A	B	C
A	–	1	0
B	1	–	2
C	0	2	–

It should be noted that not all clusters are meaningful. Cluster no. 5 displays a potential relationship between *ForEachOutPatternElement* and *IterateExp*, but upon manual examination of the ATL models involved, only one of the models contained *IterateExp* within a *ForEachOutPatternElement* element. In the remaining models, the two elements were found in differing locations in the models in which case did not suggest a relationship of interest.

4.3 Clone analysis

Clone analysis is concerned with the detection and evaluation of duplications in the usage of a language. This analysis technique was first considered in GPLs. The term *code clones* refers to duplicated sections of code. The similarity among these clones can vary from being exact duplicates of each other to being near duplicates of each other based on looser matching properties, which, for example, allows for differing names or the addition/deletion of a few statements.

One reason for the need of clone analysis on DSLs is that such activity has not received as much attention compared to the evaluation of cloning in popular GPLs. Figure 4 shows a tag cloud that is based on the number of times a language was evaluated for cloning in papers listed in a bibliography of clone-related papers.[10] It can be seen that most research on

[10] http://students.cis.uab.edu/tairasr/clones/literature.

Table 6 Top Puppet metamodel element relationships

Elements	Count
LiteralNameOrReference–ResourceBody	467
ResourceExpression–LiteralNameOrReference	467
ResourceExpression–ResourceBody	467
AttributeOperations–AttributeDefinition	463
LiteralNameOrReference–AttributeDefinition	463
LiteralNameOrReference–AttributeOperations	463
ResourceBody–AttributeDefinition	463
ResourceBody–AttributeOperations	463
ResourceExpression–AttributeDefinition	463
ResourceExpression–AttributeOperations	463

Table 7 Top ATL metamodel element relationships

Elements	Count
Module–Binding	167
Module–OutPattern	167
Module–SimpleOutPatternElement	167
OutPattern–Binding	167
OutPattern–SimpleOutPatternElement	167
SimpleOutPatternElement–Binding	167
Binding–InPattern	165
Binding–SimpleInPatternElement	165
InPattern–SimpleInPatternElement	165
Module–InPattern	165

software clones has mainly focused on GPLs, such C, C++, and Java.

The evaluation of clones can be performed for various reasons. For example, detected clones can be evaluated for elimination of the associated duplication. Higo et al. proposes a metric-based approach to identify clones for refactoring activities to modularize the code associated with the clones [15]. Modularization by abstracting a section of code into a function is a common activity in GPLs. For DSLs, such modularization is also possible, if the language supports it. In contrast, a separate solution for DSLs would be to create a new metamodel element that represents the commonly cloned constructs. In this case, cloning is removed by the addition or modification of the language itself.

In GPLs, clone detection is typically performed on syntactically meaningful sections of code. For example, all methods in an object-oriented language are compared. At a higher granularity level, all statements in the language are compared. For DSLs, the question posed is what are "meaningful" sections in the language? This must be determined during the adaption of clone detection for a particular DSL. In the case of Puppet, a *statement* metamodel element is conveniently

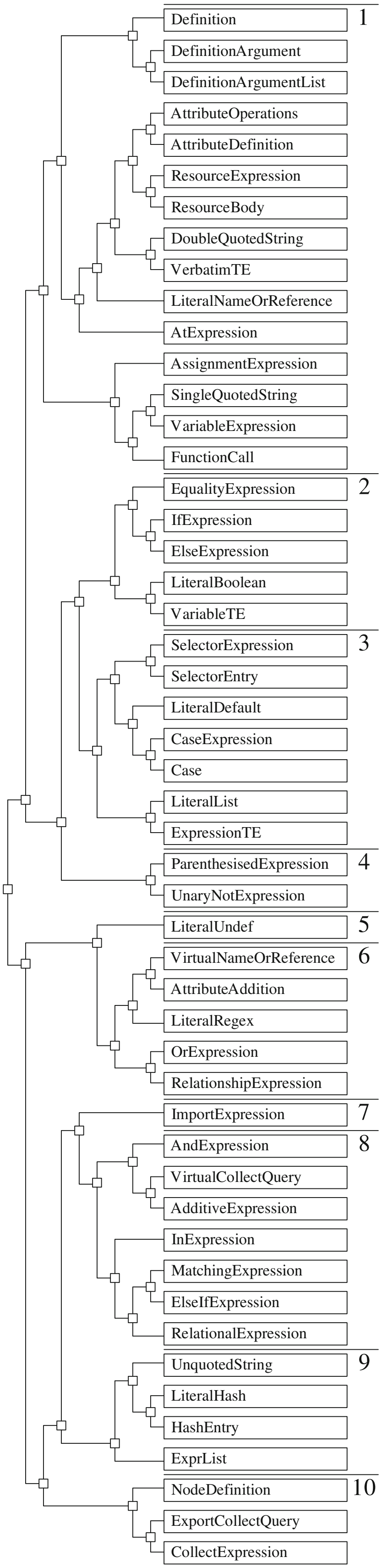


Fig. 2 Puppet metamodel element clustering

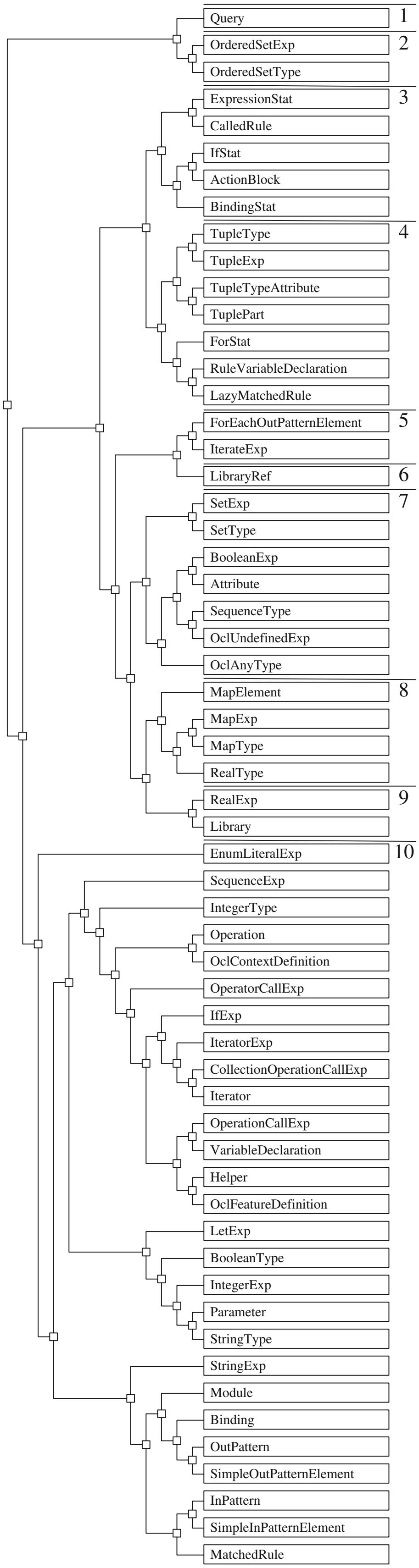


Fig. 3 ATL metamodel element clustering

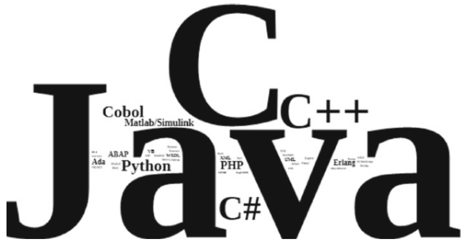


Fig. 4 Tag cloud of languages focused in clone research

part of the language. Hence, we adapt clone detection for Puppet by evaluating all *statement* metamodel elements and their underlying sequence of elements for cloning. We consider this element as representing a meaning collection of constructs of the language. Determining the proper grouping of elements to evaluate may not be as straightforward in other DSLs. In Sect. 7, we consider a more general mechanism that detects elements that are fully contained within another element in an EMF model without specifically focusing on one top-level metamodel element during the detection.

We detect clones using the suffix tree technique [11], because of its popularity as a clone detection technique [2,8,19,29]. In some usages of this technique (i.e., [8,29]), the abstract syntax tree representation of the language is used to generate a suffix tree, which is then subsequently searched for duplicate sequences. In our case, we use the EMF-based representation of Puppet instances to generate the tree. The results of the detection process must be associated with the actual concrete syntax of Puppet to allow the display of actual code snippets that are identified as clones. We use features from the Xtext infrastructure to re-associate the abstract syntax to the concrete syntax. Because currently ATL does not have an Xtext-based solution, an evaluation of ATL is not included in this subsection. However, we refer the reader to our previous work on the clone analysis of the object constraint language (OCL) part of ATL in [28].

The evaluation of the results of the clone detection is given in the remainder of this subsection. The analysis revealed that cloning in Puppet occurs throughout the corpus that was evaluated. Hence, it is not restricted to a specific module or specific authors of the modules.

4.3.1 Puppet

For the case of Puppet, we perform detection in the statement level of the Puppet models. The detection identifies statements that are Type I and II clones [3]. Type I clones are clones that are exactly the same where whitespace and comments are ignored. Type II clones are clones that may have the same sequence or structure of metamodel elements, but can differ in terms of the values associated with the elements.

Table 8 Exact clones in Puppet

Statement filter	Clone groups	SLOC
All	96	756 (4 %)
Three or more nodes	42	434 (3 %)
Six or more nodes	33	391 (2 %)
10 or more nodes	16	250 (1 %)

Table 9 Parameterized clones in Puppet

Statement filter	Clone groups	SLOC
All	197	4,492 (26 %)
Three or more nodes	195	4,443 (26 %)
Six or more nodes	182	4,238 (25 %)
10 or more nodes	151	3,673 (21 %)

Tables 8 and 9 provide a general summary of cloning in Puppet showing the amount of Type I (exact) and Type II (parameterized) clones, respectively. Different sizes of statements were considered during the detection process, which was intended to remove any small clones that may be superfluous because of its size. It can be seen that considerable cloning occurs in the statement level in Puppet (i.e., around 20 % of all SLOC in Table 9).

As stated previously, modules are a collection of Puppet models related to the configuration of certain aspects of a server. In PuppetForge, modules written by the same author can also be identified. We next consider the distribution of clones among the Puppet modules. This is related to where each clone in a clone group is located. A clone group in this case contains clones that represent the same duplication. Four types of distributions are considered:

- *Single*: the clones in a clone group reside in the same model
- *Module*: the clones in a clone group reside in the same module
- *Author*: the clones in a clone group reside in two or more modules having the same author
- *Multiple*: the clones in a clone group reside in two or more modules having different authors

Table 10 depicts the distribution of Type II clones within their respective clone groups. If we combine the *Module* and *Author* distributions, we can see three distributions that each comprise one-third of the clone groups in Puppet:

- Clone groups with clones all residing in a single model
- Clone groups with clones residing in one or more modules written by the same author
- Clone groups with clones residing in multiple modules of different authors

This observation suggests that the occurrence of cloning is evident throughout the Puppet modules that were evaluated. In other words, cloning is not restricted to specific models or modules written by specific authors. Hence, the extent of cloning that is commonplace throughout the Puppet corpus suggests that any effort to deal with the clones through, for example, introducing a new metamodel element can be considered as a general language solution.

5 Discussion

The main implication of this work is that the three analysis techniques and their associated results that are applied to DSLs have yielded information about each language based on their respective corpora, which are highlighted below. We have focused on analysis techniques that can be applied to multiple DSLs and thus become beneficial for the analysis of more than one language. Related to this, an Eclipse-based plug-in that is described in Sect. 7 offers potential for the analysis techniques to be performed on other Xtext-based DSLs.

A common knowledge gained from the results is the popularity of usage of certain metamodel elements. For example, in Puppet the *ElseIfExpression* was not used as much as the related *if* and *else* expressions. Similarly, between the two types of *select* statements, one is more prominently used than the other. This information can be used as the basis for deciding to drop the unused constructs from future versions of the language. The popularity of metamodel element usage can also provide insight into whether a DSL is being used as it is intended. For example, for ATL, the corpus analysis revealed a promising trend of the use of the declarative constructs of the language compared to the usage of imperative constructs that were included in the language. In this case, the DSL developer is reassured regarding the declarative usage of the language. These observations in both Puppet and ATL were not necessarily based on the most numerous elements listed in Tables 1, 2, 3 and 4. Instead, based on the knowledge of the language, certain elements in the table were focused on after observing their instance rate. Hence, knowledge and experience with the DSL are essential in interpreting numbers put forth by

The corpus analysis also revealed a common trend in the DSL, in which case it can provide initial evidence of the need to manage the particular trend in the language usage. For example, in Puppet, clone analysis revealed a common trend of cloning occurring throughout the language usage. The developer can determine whether these clones need to be eliminated through modularization features in future language versions.

Table 10 Parameterized clone distribution

Distribution	All	Statement filter		
		3 or more nodes	6 or more nodes	10 or more nodes
Single	65 (33 %)	64 (33 %)	61 (34 %)	56 (37 %)
Module	18 (9 %)	18 (9 %)	18 (10 %)	11 (7 %)
Author	52 (26 %)	52 (27 %)	49 (27 %)	46 (30 %)
Multiple	62 (31 %)	61 (31 %)	54 (30 %)	38 (25 %)

The initial analysis results can also be a stepping stone to more detailed analysis on a DSL. For example, in Puppet, further analysis of how DSL users interpolate string was suggested by the Puppet IDE developer after evaluating the results from our initial analysis. Further analysis of instances can include statistical distributions to find trends of occurrences of the metamodel elements. Related to the co-occurrence matrix described in Sect. 4.2, a heat map can be superimposed on the matrix to visualize high co-occurrence instances. In addition, more detailed analysis can be obtained through the use of other related analysis techniques. For example, latent semantic analysis (LSA) [6] could be considered as part of relationship analysis to determine more latent relationships embedded in the corpus of a DSL. These relationships can be based on LSA of naming usage in the corpus of the DSL if such a feature is available in the language.

Despite the observations outlined above related to the ATL and Puppet DSLs, a major challenge for corpus-based analysis on DSLs in general is the availability of a corpus for these DSLs. The evaluation of GPLs is aided by source code that is widely available from public repositories such as SourceForge and in individual open source project repositories. We have observed that this is not the case for DSLs. This situation has also been noted by other researchers [17,27]. Public repositories containing a DSL corpus are not as widely available as GPLs. This may be due to the fact of the domain-specific nature of the languages, which limits their number of users. Another possibility is the sometimes proprietary nature of a DSL, which restricts the exposure of an associated corpus to the public.

6 Threats to validity

Despite our analysis techniques being generic in the sense that they can be applied to any DSL, the fact that they have been validated using only two specific DSLs is a clear threat to validity of this study. More DSLs need to be examined in order to confirm the usefulness of corpus-based analysis techniques. As commented before, a major hurdle for this extended analysis is the limited availability of repositories of DSL models. However, the sizes of the corpora used in our analysis are comparable in size to the corpora used in other related DSL corpus analysis research. The corpora used in related work that will be described in Sect. 8 consisted of between under 100 to over 1,000 items/models. The corpora sizes of ATL (i.e., 189 models) and Puppet (i.e., 706 models) are comparable to other corpora evaluated.

The quality of the corpus can also be a bias in the analysis. A corpus must be representative of the DSL instances created by end-users. For instance, one of the ATL developers we consulted with suggested that the ATL corpus we were using mainly consisted of good (i.e., well-written) ATL transformations, which could explain why the presence of (undesired) imperative constructs was very limited. This could inevitably threaten the validity of the analysis results. To alleviate this, the corpus of a DSL must be evaluated to determine whether it is representative of the users of the DSL and not only of, for instance, expert users.

Related to clone analysis, the concrete syntax of a DSL can be very different from that of popular GPLs such as C and Java. In some cases, such as in Puppet, the ordering in specific constructs is not important. For example, in the following snippet of Puppet code, the list of attributes and their respective values can be written in a different order, but will mean the same in Puppet.

```
file {’testfile’:
 path => ’/tmp/testfile’,
 ensure => present,
 mode => 0640,
 content => ’’I’m a test file.’’,
}
```

Our suffix tree-based clone detection technique cannot identify clones where elements that have the same meaning are ordered differently. Hence, special consideration must be included for DSLs that exhibit similar characteristics.

7 Tool support

In this section, we describe an Eclipse plug-in[11] that offers the corpus-based analysis techniques from Sect. 4 for Xtext-

[11] http://code.google.com/a/eclipselabs.org/p/dsl-analysis.

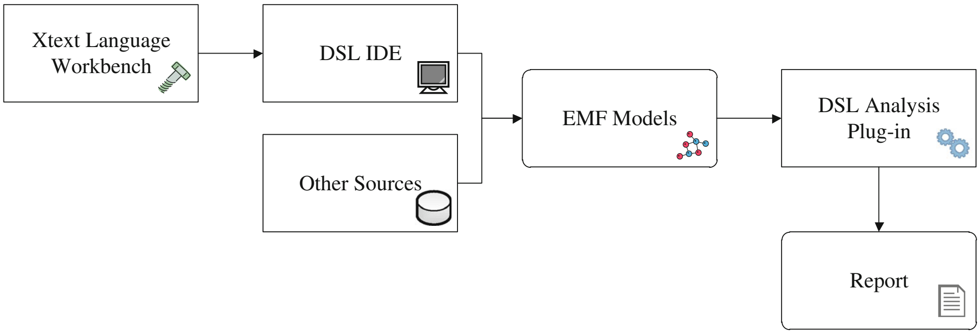


Fig. 5 DSL analysis process

generated DSLs. We selected Xtext not only due to its current popularity as a language workbench, but also because the DSL instances in the Xtext DSL infrastructure are represented in-memory as an EMF model [32]. EMF models in their own right are widely used within the MDE paradigm.

Given that we process the DSL corpus through their EMF model representations, DSLs written outside Xtext but that are represented as EMF models can also utilize at least the instance and relationship analysis techniques. This is because these two techniques extract their information directly from the models. In contrast, the clone analysis technique requires additional information, because we must associate the concrete syntax of the clone segments to allow the language engineer to actually see what parts of the corpus are cloned. This forces to have available the Xtext infrastructure in order to associate cloned segments of the model with the actual DSL code.

Figure 5 outlines the DSL analysis process. The Eclipse plug-in performs its analysis on the EMF model representation of DSL instances. For DSLs written with Xtext, APIs are available that allow for the extraction and manipulation of EMF models representing DSL instances by external sources such as our plug-in. In addition, the plug-in can also obtain the models from sources other than Xtext (i.e., *other sources*) for the instance and relationship analyses, but not for clone analysis.

It should be noted that the clone analysis for Puppet described in Sect. 4.3 performed clone detection on all statements under the *Definition* metamodel element in the Puppet corpus. This allowed us to focus detection on a structurally meaningful metamodel element. However, for the generic clone detection version, the dependence on a specific metamodel element of a specific DSL must be removed. In this case, we perform suffix tree-based clone detection without identifying a structurally meaningful metamodel element or group. This makes the detection process require an additional step, as it must identify *whole clones* from the results. In the plug-in, we only display clones that represent an entire metamodel element and its contained objects or a sequence of metamodel elements and their corresponding contained objects. This is similar to the process of finding clones representing meaningful syntactic blocks as described in [8].

The analysis report generated by the plug-in is displayed in views and an HTML file. The top view in Fig. 6 displays the number of times a metamodel element is used overall (i.e., "All count" column) and the number of models an element is used in (i.e., "Model count" column). It should be noted that an Eclipse plug-in that incorporates instance analysis on individual models of Xtext-based DSLs has been proposed.[12] In contrast, we seek to perform instance analysis on an entire corpus of a DSL rather than just a single instance.

The bottom view in Fig. 6 displays the generated clusters. For the plug-in, we replaced the gCluto stand-alone clustering tool that was used in Sect. 4.2 with Java-based libraries from Weka[13] to allow the clustering results to be directly available in Eclipse. This is the reason for the different clustering results in Fig. 6 compared to that in Fig. 2. A Newick tree format[14] can be obtained from the plug-in and exported to a graphical renderer to display a cluster dendrogram similar to Figs. 2 and 3.

Currently the results of the clone detection process are displayed as an HTML file, a snippet of which can be seen in Fig. 7. For each clone group reported by the detection process, the number of clones in the group, the number and names of the files that contain the clones are given. The actual code associated with each clone is also displayed. A more streamlined mechanism where the clones are highlighted directly in the source editor is being considered.

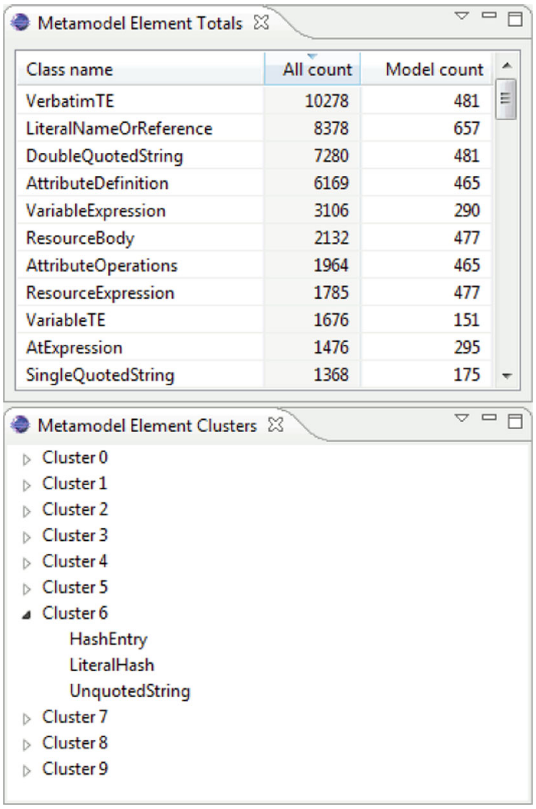
8 Related work

In the following, we describe and compare several related works that have considered the evaluation of DSL(s) based on an available corpus.

[12] http://www.sigasi.com/content/view-complexity-your-xtext-ecore-model.

[13] http://www.cs.waikato.ac.nz/ml/weka.

[14] http://evolution.genetics.washington.edu/phylip/newicktree.html.



Class name	All count	Model count
VerbatimTE	10278	481
LiteralNameOrReference	8378	657
DoubleQuotedString	7280	481
AttributeDefinition	6169	465
VariableExpression	3106	290
ResourceBody	2132	477
AttributeOperations	1964	465
ResourceExpression	1785	477
VariableTE	1676	151
AtExpression	1476	295
SingleQuotedString	1368	175

- Cluster 0
- Cluster 1
- Cluster 2
- Cluster 3
- Cluster 4
- Cluster 5
- Cluster 6
  - HashEntry
  - LiteralHash
  - UnquotedString
- Cluster 7
- Cluster 8
- Cluster 9

Fig. 6 Eclipse plug-in views

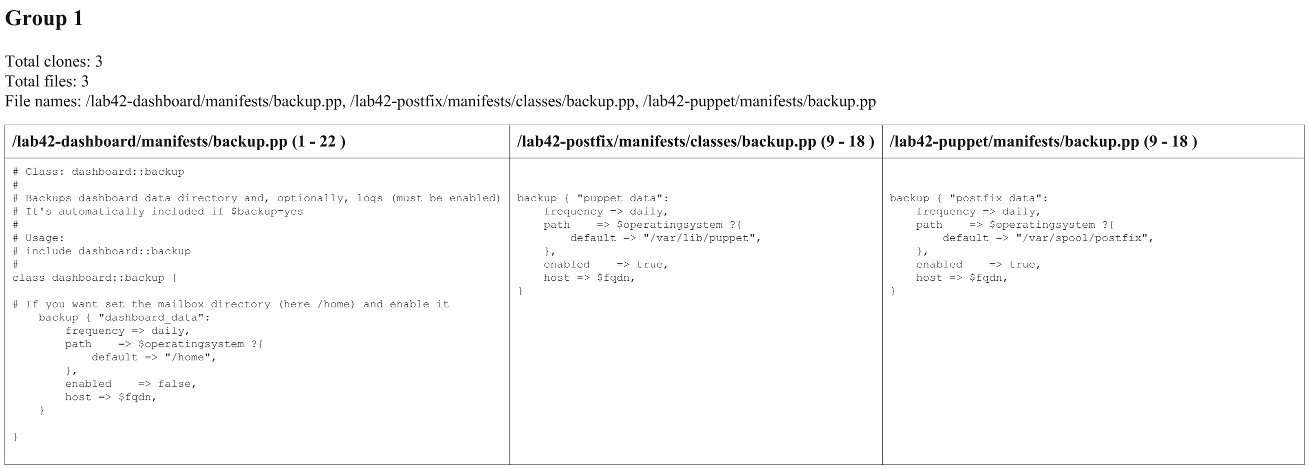
Muehlen and Recker performed analysis on a corpus consisting of business process modeling notation (BPMN) models [25]. Instance and relationship analyses were considered, but clone analysis was not. Muehlen and Recker wanted to determine what parts of BPMN were used more than others based on the corpus. They proposed the results as a way to educate future BPMN users on how to reduce the complexity of BPMN models. In this sense, the target of the results is the users of the DSL rather than the DSL language engineer, which is who we focus on in our work.

Lämmel and Pek evaluated a corpus of the P3P Web policy language using various techniques including clone and instance analysis [21]. The analysis provided general characteristics of the P3P language such as the prevalence of cloning among P3P files and extensions of the language that exceeded the complexity of the base language. Although the paper considered the abstract syntax of P3P, much of the evaluation is on actual values of the configuration, such as data constants and extensions. In contrast, we deal with the evaluation of the use of the metamodel elements of the language.

Jeanneret et al. [17] proposed techniques to determine the footprints of pre-defined operations on models to determine to what extent were the elements in the models touched or used in certain operations. The target of the results are for the users or developers of the models to assist them in generating models with proper levels of detail as they relate to the operations that will be performed on them. Such information could also be beneficial for the language engineer to consider whether the language used to create the models is too detailed for the operations performed on them.

Monperrus et al. [24] defined a generative approach of measuring domain-specific models. A model measurement tool can be generated by specifying the desired metrics as a model that conforms to a metric specification metamodel. Our Eclipse plug-in differs in that it is generic in the sense that it does not rely on any specific information from a DSL's metamodel.

The observation of cloning in UML models was considered by Störrle [27]. UML models were translated into Prolog where the detection process was performed. As we focus on DSLs developed in Xtext, the textual nature of these languages allowed us to adapt a cloning technique from GPLs. We will consider incorporating a clone detection technique that is based more on the graphical representation of DSLs.



Group 1

Total clones: 3
Total files: 3
File names: /lab42-dashboard/manifests/backup.pp, /lab42-postfix/manifests/classes/backup.pp, /lab42-puppet/manifests/backup.pp

/lab42-dashboard/manifests/backup.pp (1 - 22)

```
# Class: dashboard::backup
#
# Backups dashboard data directory and, optionally, logs (must be enabled)
# It's automatically included if $backup=yes
#
# Usage:
# include dashboard::backup
#
class dashboard::backup {

# If you want set the mailbox directory (here /home) and enable it
    backup { "dashboard_data":
        frequency => daily,
        path    => $operatingsystem ?{
            default => "/home",
        },
        enabled    => false,
        host => $fqdn,
    }

}
```

/lab42-postfix/manifests/classes/backup.pp (9 - 18)

```
backup { "puppet_data":
    frequency => daily,
    path    => $operatingsystem ?{
        default => "/var/lib/puppet",
    },
    enabled    => true,
    host => $fqdn,
}
```

/lab42-puppet/manifests/backup.pp (9 - 18)

```
backup { "postfix_data":
    frequency => daily,
    path    => $operatingsystem ?{
        default => "/var/spool/postfix",
    },
    enabled    => true,
    host => $fqdn,
}
```

Fig. 7 Snippet of clone detection report

Still related to UML, several works have performed evaluation on a UML corpus [4,10]. However, many of the metrics that are reported are specific to UML (e.g., number of associations, generalizations, and use cases). Our goal of evaluation is to be applicable to different DSLs; hence, we selected properties that are generic enough to be applied to different languages. Nevertheless, evaluations that are specific to a particular language is worth identifying if they can be of benefit. In addition to these works, Kim and Boldyreff [20] and Lange et al. [22] propose tools that can report several types of metrics on UML corpus. However, no actual evaluation was performed on a particular UML artifact.

These works and ours have a commonality in that all perform or facilitate the evaluation of DSLs based on the actual usage of the language (i.e., based on a corpus). Although the purposes of the analyses vary and the languages evaluated differ, these works provide a growing collection of research on DSL analysis, which can potentially contribute positively to the overall DSL developmental process and to the DSL community.

9 Conclusion and future work

In this paper, we have shown the extraction of DSL characteristics that are based on the evaluation of the actual usage of the language. The utilization of instance, relationship, and clone analyses on the Puppet and ATL DSLs has revealed useful characteristics about these languages for the respective language engineer to take into consideration in identifying future improvements in the language. Corpus-based analysis can complement other DSL analysis techniques in an overall effort to evaluate a DSL.

For future work, we will consider evaluating other DSLs that have an associated corpus. We will also consider additional analysis techniques based on discussions with DSL authors. For example, we have previously investigated the use of LSA on C programs [30], which could also be considered for DSLs. We would also like to consider shared characteristics in DSLs that can point to certain useful ”design patterns” (or on the contrary, anti-patterns) of DSLs. In addition, we would also like to integrate our tool with other popular language workbenches. In a separate direction, the comparisons of analysis results of GPLs and DSLs will also be investigated to see whether both share common characteristics or there are actually different language structures that are common to DSLs but not in GPLs.

References

1. Abrahão, S., Iborra, E., Vanderdonckt, J.: Usability evaluation of user interfaces generated with a model-driven architecture tool. In: Law, E.L.-C., Hvannberg, E.T., Cockton, G. (eds) Maturing Usability, Human-Computer Interaction Series, pp. 3–32. Springer, London (2008)
2. Baker, B.: On finding duplication and near-duplication in large software systems. In: Working Conference on Reverse Engineering, pp. 86–95. IEEE Computer Society, Washington, DC (1995)
3. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. **33**(9), 577–591 (2007)
4. Berenbach, B.: The evaluation of large, complex UML analysis and design models. In: International Conference on, Software Engineering, pp. 232–241 (2004)
5. Cabot, J, Clarisó, R, Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: International Conference on Automated Software Engineering, pp. 547–548. ACM, New York, NY (2007)
6. Deerwester, S., Dumais, S., Furnas, G., Landauer, T., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**, 391–407 (1990)
7. Dobing, B., Parsons, J.: Dimensions of UML diagram use: a survey of practitioners. J. Database Manag. **19**, 1–18 (2008)
8. Falke, R., Frenzel, P., Koschke, R.: Empirical evaluation of clone detection using syntax suffix trees. Empir. Softw. Eng. **13**, 601–643 (2008)
9. Gabriel, P., Afonso Goulão, M., Amaral, V.: Do software languages engineers evaluate their languages? In: Congreso Iberoamericano en ”Software Engineering, pp. 149–162 (2010)
10. Genero, M., Manso, E., Visaggio, A., Canfora, G., Piattini, M.: Building measure-based prediction models for UML class diagram maintainability. Empir. Softw. Eng. **12**, 517–549 (2007)
11. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
12. Hage, J., Keeken, P.: Neon: a library for language usage analysis. In: Gašević, D., Lämmel, R., Wyk, E. (eds.) International Conference on Software Language Engineering, Volume 5452 of Lecture Notes in Computer Science, pp. 35–53. Springer, Berlin (2008)
13. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco, CA (2005)
14. Hermans, F., Pinzger, M., van Deursen, A.: Domain-specific languages in practice: a user study on the success factors. In: Schürr, A., Selic, B. (eds.) International Conference on Model Driven Engineering Languages and Systems, volume 5795 of Lecture Notes in Computer Science, pp. 423–437. Springer, Berlin (2009)
15. Higo, Y., Kusumoto, S., Inoue, K.: A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. J. Softw. Maint. Evol. Res. Pract. **20**(6), 435–461 (2008)
16. Hubaux, A., Boucher, Q., Hartmann, H., Michel, R., Heymans, P.: Evaluating a textual feature modelling language: four industrial case studies. In: Malloy, B., Staab, S., van den Brand, M. (eds.) International Conference on Software Language Engineering, volume 6563 of Lecture Notes in Computer Science, pp. 337–356. Springer, Berlin (2010)
17. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: International Conference on Software Engineering, pp. 601–610. ACM, New York, NY (2011)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
19. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7), 654–670 (2002)
20. Kim, H., Boldyreff, C.: Developing software metrics applicable to UML models. In: Workshop on Quantitative Approaches in Object-Oriented, Software Engineering (2002)
21. Lämmel, R., Pek, E.: Vivisection of a non-executable, domain-specific language-understanding (the usage of) the P3P language.

In: International Conference on Program Comprehension, pp. 104–113 (2010)

22. Lange, C., Wijns, M., Chaudron, M.: Metricviewevolution: UML-based views for monitoring model evolution and quality. In: European Conference on Software Maintenance and Reengineering, pp. 327–328 (2007)
23. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**, 316–344 (2005)
24. Monperrus, M., Jézéquel, J.-M., Baudry, B., Champeau, J., Hoeltzener, B.: Model-driven generative development of measurement software. Softw. Syst. Model. **10**, 537–552 (2011)
25. Muehlen, M., Recker, J.: How much language is enough? Theoretical and practical use of the business process modeling notation. In: Bellahsène, Z., Léonard, M. (eds.) International Conference on Advanced Information Systems Engineering, volume 5074 of Lecture Notes in Computer Science, pp. 465–479. Springer, Berlin (2008)
26. Recker, J., Muehlen, M., Siau, K., Erickson, J., Indulska, M.: UML versus BPMN. In: Americas Conference on Information Systems, Measuring method complexity (2009)
27. Störrle, H.: Towards clone detection in UML domain models. Softw. Syst. Model. pp. 1–23 (2012)
28. Tairas, R., Cabot, J.: Cloning in DSLs: experiments with OCL. In: Assman, U. Sloane, A. (eds.) International Conference on Software Language Engineering, volume 6940 of Lecture Notes in Computer Science, pp. 60–76. Springer, Berlin (2011)
29. Tairas, R., Gray, J.: Phoenix-based clone detection using suffix trees. In: ACM Southeast Regional Conference, pp. 679–684. ACM, New York, NY (2006)
30. Tairas, R., Gray, J.: An information retrieval process to aid in the analysis of code clones. Empir. Softw. Eng. **14**, 33–56 (2009)
31. van Amstel, M., van den Brand, M., Engelen, L.: An exercise in iterative domain-specific language design. In: Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pp. 48–57. ACM, New York, NY (2010)
32. Xtext. Xtext documentation (2012)

Author Biographies

Robert Tairas received his M.Sc and Ph.D. in computer science from the University of Alabama at Birmingham. He has been a post-doctoral research fellow for the AtlanMod team, an INRIA and LINA research group at École des Mines de Nantes. His research interests include code clone analysis, software language analysis, and model-driven engineering.

Jordi Cabot is currently leading the AtlanMod team, an INRIA and LINA research group at École des Mines de Nantes (France). Previously, he has been a post-doctoral fellow at theUniversity of Toronto, a senior lecturer at the UOC (Open University of Catalonia) and a visiting scholar at the Politecnico di Milano. He received the B.Sc and Ph.D. degrees in computer science from the Technical University of Catalonia. His research interests include software modeling, model-driven and Web engineering, formal verification, and social aspects of software engineering. Apart from his scientific publications in international conferences and journals in these areas, he writes and blogs about all these topics in his Modeling Languages portal (http://modeling-languages.com).