A Model-Driven Approach to Generate External DSLs from Object-Oriented APIs

Valerio Cosentino, Massimo Tisi, Javier Luis Cánovas Izquierdo

A Model-Driven Approach to Generate External DSLs from Object-Oriented APIs

Valerio Cosentino, Massimo Tisi, and Javier Luis Cánovas Izquierdo

AtlanMod team (Inria, Mines Nantes, LINA), Nantes, France
firstname.lastname@inria.fr

Abstract. Developers in modern general-purpose programming languages create reusable code libraries by encapsulating them in Applications Programming Interfaces (APIs). Domain-specific languages (DSLs) can be developed as an alternative method for code abstraction and distribution, sometimes preferable to APIs because of their expressivity and tailored development environment. However the cost of implementing a fully functional development environment for a DSL is generally higher. In this paper we propose DSLit, a prototype-tool that, given an existing API, reduces the cost of developing a corresponding DSL by analyzing the API, automatically generating a semantically equivalent DSL with its complete development environment, and allowing for user customization. To build this bridge between the API and DSL technical spaces we make use of existing Model-Driven Engineering (MDE) techniques, further promoting the vision of MDE as a unifying technical space.

1 Introduction

Modern General-purpose Programming Languages (GPLs) provide facilities for program abstraction and reuse, to foster the development of distributable libraries. Code in a programming library is encapsulated behind Application Programming Interfaces (APIs), that are used in user programs by the mechanisms provided by the GPL (e.g., function call or class inheritance). Sometimes library developers prefer to provide their users with a Domain-Specific Language (DSL), instead of (or in addition to) an API. APIs and DSLs can be seen as alternative methods to access the library functionalities, and are characterized by specific advantages. Programs written in the DSL can be more expressive, maintainable, concise and readable than corresponding GPL programs using the API (e.g., by avoiding the user to write some boilerplate code) [1,2]. On the other side, APIs allow for natural integration in complex programs written in their native language (or in other languages when coupled with suitable interface bindings).

Literature distinguishes DSLs in internal and external [3]. Internal DSLs are created by embedding DSL constructs into an existing GPL, which acts as host language. Although the internal approach allows DSLs to be easily developed [4], the corresponding tooling relies on the existing support for the host language, which limits the domain-specific assistance [5]. External DSLs instead are characterized by a separate syntax and specific development facilities. An important advantage of this approach is that the domain-specific development environment can be tailored to ease coding in the DSL:

- Static validation can be enriched to enforce semantic constraints hidden in the API. Thus some runtime errors can be avoided at compile time.
- Features like syntax highlighting, code completion, outlining, folding, can be tailored to the DSL.
- The DSL interpretation/compilation step can be designed to automatically optimize the DSL code execution.

While, depending on the case, DSL or API (or both) may be the preferable solution [6], the development cost of a DSL, especially if external, is in general much higher. Users have to define the DSL (i.e., abstract and concrete syntaxes, and semantics) and develop the domain-specific environment (e.g., syntax highlighting, code assistance, folding, outline view) which are tedious and time-consuming tasks.

In this paper we propose a method to automatically analyze an existing object-oriented API and generate an external DSL out of it. Our approach leverages model-driven techniques to analyze and represent APIs at high-level of abstraction (i.e., as metamodels) which are later used to automatically generate the DSL components and the corresponding tooling, including parser, compiler and development environment. Developers can influence the DSL generation by editing the model-based API representation and by specifying design choices about the structure of the DSL to generate.

We provide a proof-of-concept implementation of the method in the DSLit tool, that is able to analyze Java APIs and generate external textual DSLs using the Xtext framework [7]. DSLit is currently able to deal with two API categories that we describe. The first category is called *Plain Old Data* (POD[1]) and indicates simple APIs that have the purpose of creating and maintaining a data structure. Usually such APIs are composed by classes made exclusively of getters, setters and constructors. The second category is called *Fluent* and contains those APIs that rely on chaining method calls. The return value of these method calls is an object representing the context of the keyword, and it is used to structure the language, defining which keywords can follow other keywords. For APIs not included in these categories, we also provide a fallback category, called *SimpleJava* based on a subset of Java which includes statements and declarations.

While currently limited in scope, the DSLit prototype, freely available at the project website[2], demonstrates the feasibility and usefulness of the approach.

The paper is structured as follows. Section 2 presents concrete examples to motivate our approach. Section 3 describes the conceptual framework applied to obtain a DSL from a Java API, while Section 4 presents the implementation of the prototype tool and the solution of the running cases. Section 5 lists the related work and Section 6 finalizes the paper and outlines some further work.

2 Motivating Examples

While APIs have proven to be a flexible means to encapsulate and reuse program logic, their usage can be cumbersome. A typical example is Java Swing, an API based on the

[1] http://en.wikipedia.org/wiki/Plain_old_data_structure
[2] http://www.emn.fr/z-info/atlanmod/index.php/DSLit

Java Swing

```
JFrame jFrame1 = new JFrame();
jFrame1.setTitle("My Java Application");
jFrame1.setSize(500, 300);
JLabel jLabel1 = new JLabel();
jFrame1.add(jLabel1);
jLabel1.setText("Hello World!");
jFrame1.setVisible(true);
```

(a)

JavaFX Script

```
Frame {
  title: "My Java Application"
  width: 500
  height: 300
  content: Label {
    text: "Hello World!"
  }
  visible: true
}
```

(b)

DSLit

```
JFrame {
  title: "My Java Application"
  size: 500, 300
  JLabel {
    text: "Hello World!"
  }
  visible: true
}
```

(c)

Fig. 1. A Swing example using (a) Java code, (b) JavaFX and (c) DSLit

Abstract Window Toolkit (AWT[3]) for the development of graphical user interfaces for Java applications. Several DSLs have been developed to allow more concise and readable interface specifications with respect to Swing-like code. An example is JavaFX[4], a framework specifically tailored to create rich internet applications (RIA) that includes the so-called JavaFX Script, which is a DSL enabling the fast definition of user interfaces. Figures 1a and 2b compare two equivalent chunks of code written in Java Swing and JavaFX Script, respectively.

Both examples specify the creation of a frame including a title and a label and the JavaFX Script version is remarkably more concise and readable. Developers in JavaFX are free from writing most of the boilerplate code and can use a declarative language specifically adapted to the creation of user interfaces. However, JavaFX Script was not developed as a DSL for targeting the Java Swing API but as a DSL to develop user interfaces rapidly and effectively. Thus, JavaFX Script code does not compile to the corresponding Java Swing code and it incorporates extra features such as declarative animation or mutation triggers.

As can be seen, a clear correspondence can be drawn among the DSL constructs and the Java API calls. For instance the ”title:” element corresponds to a call to setTitle(). In this case DSL and API seem to lay at the same abstraction level, thus theoretically allowing for a purely syntactic translation. Note that this example does not show how to handle user interface event handlers, which can execute arbitrary actions, and are therefore generally written in GPL code.

The snippet in the Fig. 1c is written in the DSL obtained with DSLit by analyzing the Swing API. The snippet shows only a few lexical differences with the JavaFX version. In this sense, the constructs of the automatically-generated DSL mimic the structure defined in the API, e.g., there is a DSL element for each API method. In addition, a compiler is also generated by DSLit that translates this snippet to the program in the Fig. 1a.

As we will show, the conciseness of the previous DSL comes from the particular containment structure of the Swing API. As a significantly different example we show in Fig. 2a a program using jRTF[5], a Fluent API to construct Rich Text Format (RTF) documents by Java. According to M. Fowler[6], a Fluent API is an implementation of

[3] http://java.sun.com/products/jdk/awt
[4] http://docs.oracle.com/javafx
[5] http://code.google.com/p/jrtf/
[6] http://martinfowler.com/bliki/FluentInterface.html

jRTF

```
rtf().section(
  p( "first paragraph" ),
  p(
    tab(),
    " second par ",
    bold( "with something in bold" ),
    " and ",
    italic( underline( "italic underline" ) )
  )
).out( out );
```

(a)

DSLit

```
rtf section {
  p { "first paragraph" },
  p {
    tab,
    " second par ",
    bold{ "with something in bold" },
    " and ",
    italic{ underline{ "italic underline" } }
  }
} out { out }
```

(b)

Fig. 2. An excerpt of Java code using the jRTF Fluent API and the corresponding automatically-generated DSL

an object-oriented API that aims to provide more readable code. It is normally implemented by using method chaining to relay the instruction context of subsequent calls (but the Fluent paradigm is not limited to method chaining). Fluent APIs are becoming a very popular way to implement internal DSLs in Java. The jRTF example in Fig. 2 shows how a method chain in the Fluent API can closely resemble a DSL.

Fig. 2b shows the DSL automatically generated from jRTF by DSLit. Differently from the Swing case, the DSL in Fig. 2 provides very little syntactic simplification w.r.t. its corresponding Java code. However, even in this case, the generation allows, for instance, generating an environment that has a domain-specific outline representing the RTF document structure, and it can be augmented with static checking capabilities (that are poor in the fluent API version).

In this paper we present a method that, given an API, generates an equivalent DSL. Our current application of this approach supports APIs fitting in one of the two categories previously defined plus a fallback category which resembles Java-like languages. We provide DSLit, a tool that generates such DSL, together with its development environment and a Java compiler, providing the following benefits:

- The generated DSL development environment has features like syntax highlighting or code completion that are tailored to the API domain.
- Semantic constraints can be made explicit and static validation can be enriched by parameterizing the generation process. E.g., the DSL for Swing can be customized so that labels are always created in a single container frame, and the frame name is a mandatory attribute (avoiding at compile time some common mistakes in interface development).
- The DSL compiler can be manually improved to optimize the resulting API code (e.g., reordering DSL definition elements to get optimal performance).

3 From API to DSL

Fig. 3 gives an overview of the linguistic architecture of our approach, spanning over three technical spaces (TS)[7]: (1) the API TS, in which API objects (i.e., in memory)

[7] The concept of Technical Space (TS) is introduced in [8]. It is defined according to a conformance relationship that associates artifacts (e.g. program) with meta-artifacts (e.g. grammar). Bridges can be defined to transfer artifacts from one to another TS.

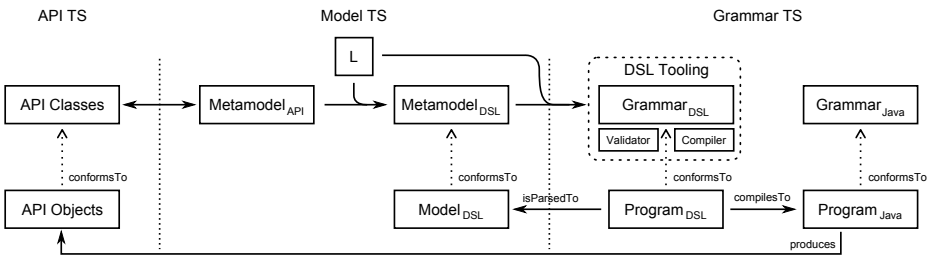

Fig. 3. Technical spaces and bridges

conform to the set of defined API classes, (2) the Model TS (we refer to the MOF/Ecore TS), in which models conform to metamodels, and (3) the Grammar TS, in which programs conform to the grammar (e.g., a GPL).

The process for obtaining a DSL from an API is split into three steps: (1) extracting an API metamodel for the set of class definitions in the API, (2) computing a DSL metamodel from the API metamodel, and (3) generating the grammar and the DSL tooling from the DSL metamodel. The steps are detailed in the following subsections.

3.1 API Classes to API Metamodel

The first step generates an API metamodel by applying a bridge between the API and Model TSs. This bridge maps API class definitions into metamodel elements. Thus Java classes are mapped into metaclasses, while attributes and methods are mapped into metaclass attributes and references/operations, respectively. The mapping is not trivial because of the semantic differences between class definitions and metaclasses, but it is well studied in works such as [9] and [10].

When applied to big APIs, this step may generate very large metamodels. For instance in Fig. 1 the bridge would create a metaclass for each class and interface of the Swing API. Our approach provides a customization mechanism that allows filtering out API elements (e.g., classes, methods, attributes) in order to influence the construction of the DSL metamodel. Filtered elements will typically include technical classes that are out of the developer's interest or do not belong to the level of abstraction of the DSL. For instance, in the Swing example the developer may be interested only in the JFrame and JLabel class, with all their ancestors in the inheritance hierarchy.

Once the API metamodel is generated, developers can leverage in metamodel techniques to make explicit semantics in the DSL that are hidden in Java. An example is the semantics of references in metamodels: references can have containment semantics and multiplicity constraints that are implicit in Java attributes. To this aim, developers may manually enrich the API metamodel to exploit these aspects in the generated DSL, which can be statically checked on the DSL code. For instance, by adding a containment property to the reference between JFrame and JLabel the resulting DSL may automatically check that a label is not contained in two distinct frames.

3.2 API Metamodel to DSL Metamodel

API classes represent an internal abstraction mechanism of object-oriented languages, i.e., an *in-language abstraction*. The API metamodel we obtained in the previous step is an artifact describing this in-language abstraction. The purpose of the second step is to transform the in-language abstraction in a *linguistic abstraction*, i.e., an abstraction defined by language constructs. We perform the transformation between in-language abstraction and linguistic abstraction within the Model TS as a model transformation, thus creating the DSL metamodel.

Lifting the abstraction to the linguistic level is not a trivial step, as the logic to apply is strongly dependent on the structure of the DSL the user wants to obtain. For instance in the Swing DSL we want to generate language concepts for classes (e.g. JFrame, JLabel) and attributes (e.g. title, size) of the API. Conversely, in the RTF example the language structure contains a concept for each method of the API.

The linguistic abstraction of a DSL contains: (a) the domain concepts, which are extracted from the API metamodel; and (b) its structure and capabilities, which define how the concepts can be defined, linked and composed (e.g., which concepts become *Statements*, whether the DSL is going to use *Blocks*, etc.). While the former is domain-specific, the latter can be considered domain-independent and be reused in different DSLs. For instance, in our first example the domain contains the elements JFrame and JLabel (and their attributes) while the structure of programs in the Swing DSL may be composed by a sequence of statements initializing the JFrame and JLabel attributes.

In order to generate the DSL metamodel we built a template system which receives as inputs: (a) the API metamodel, and (b) a template defining the structure and capabilities of the languages. Our approach currently provides three templates for the categories considered in DSLit but more templates can also be plugged in.

3.3 DSL Metamodel to DSL Environment

The last step is a bridge between the Model TS and the Grammar TS which produces the needed artefacts for the DSL. The Model TS already contains several well-known tools that help in generating the components of an external textual DSL environment (e.g., Xtext). Therefore, this step is devoted to generate the input artifacts for these tools, including: (1) the mapping of metamodel elements (i.e., the abstract syntax definition of the DSL) into the grammar rules of the concrete syntax, (2) development environment (e.g., validators, type system, etc.) and (3) compiler.

The generation process is parameterized by the DSL metamodel and the template chosen. While the DSL metamodel provides domain-specific information (e.g., concepts, attributes, references) and basic semantics (e.g., cardinalities, containment, etc.), the template drives the grammar structure of the resulting DSL and also the development environment and compiler.

The resulting compiler is able to transform DSL programs in their corresponding Java programs. As shown in Fig. 3 the compiler is an artefact of the Grammar TS, and the execution of the compiled Java program produces the set of API objects in the API TS (plus possibly other objects). Most tools create also a parser towards the Model TS that extracts from programs the corresponding instance of the DSL metamodel.

The next section explains how we implement the approach and illustrates in detail how the motivating examples are addressed.

4 DSLit

As a proof of concept of the described approach, we have implemented DSLit, a prototype DSL generator integrated in the Eclipse platform. The current prototype contains three DSL templates that aim to address the APIs that fall under the categories previously introduced, respectively *POD*, *Fluent* and *SimpleJava*.

Once one of the provided templates is selected, our tool is able to generate a domain-specific development environment, using Xtext. Currently DSLit supports the generation of a Proposal Provider and Validator components of the environment. In future work we plan to investigate the domain-specific customization of other components.

4.1 Grammar Generation

In this section, we describe the DSL templates included in DSLit, covering three possible representations of the information contained within the API metamodel.

POD DSL Generator. DSLit provides an ad-hoc POD DSL Generator. The generator can be applied to any API, but it only considers their POD part (setters, getters and constructors) for the definition of the DSL. This generator has been applied to a Swing subset and generates the DSL in Fig. 1c.

In the following we briefly describe the transformation logic for the generation of the DSL grammar model (conforming to the Xtext metamodel) from the API metamodel. The full code of the transformation is available at the paper website.

The POD DSL extracted out of the API metamodel takes into account only attributes and references contained in the classes defined in the metamodel. All the classes, attributes and references are mapped to grammar rules in Xtext and their names will appear as terminals in the grammar. Each class is transformed into a rule that contains, wrapped into braces, the features that correspond to the attributes and references for that class. Each of those features expands to the rule that represents the type of the corresponding attribute or reference, while its cardinality depends on the cardinality of the corresponding attribute or reference. In particular, a multi valued attribute is expressed as a feature list, an optional attribute is expressed as an optional feature and finally a single valued attribute is expressed as a single feature. Finally, since we know the semantics of the POD template, our tool is able to append additional grammar rules in order to avoid identifying the root class of the API metamodel. Thus, it adds to the template the rules *Grammar* and *Element*. The former is the grammar root rule and contains a list of *Elements*, while the latter includes as alternatives all the rules that correspond to the metaclasses.

In Fig.4, an excerpt of the generation of the Java Swing DSL is shown. The classes *Frame*, *MenuBar* are mapped to the corresponding grammar rules (generated according to the schema *className-attrOrRefName-AttrOrRefTypeName*). The rule *Frame* contains as terminal the name of the related class and two optional features *menuBar* and

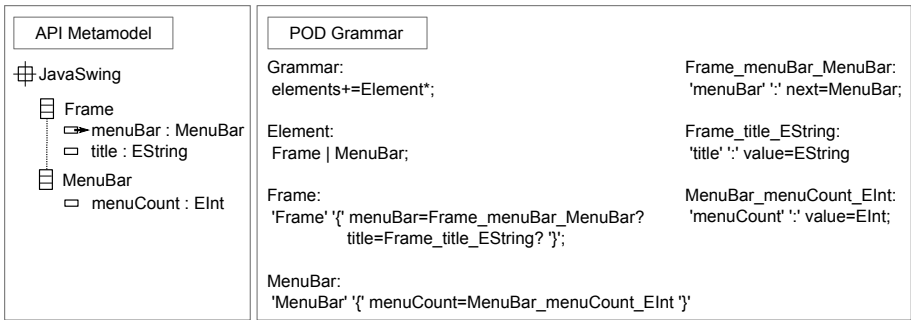

Fig. 4. Example of the POD DSL for Swing

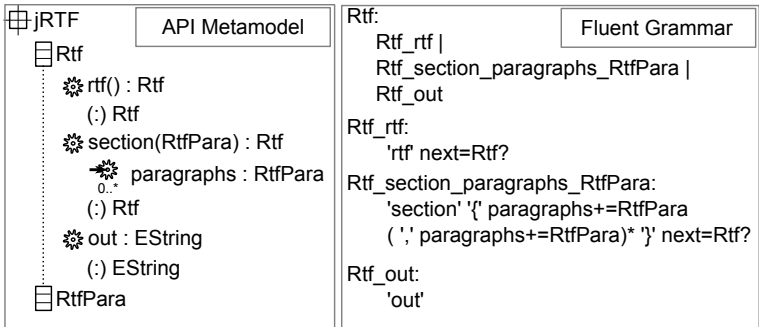

Fig. 5. Example of the generation of a Fluent DSL

title that represent respectively the reference and the attribute embedded in the class *Frame*. They expand in two rules *Frame_menuBar_MenuBar* and *Frame_title_EString*. The former contains the name of the reference *menuBar* as terminal and the feature *next* (inserted by the generator) that expand in the rule *MenuBar*; the latter contains the name of the attribute *title* as terminal and the feature *value* (inserted as well by the generator), since the *title* is defined as a primitive type. The remaining rules *MenuBar* and *MenuBar_menuCount_EInt* follow the same mapping strategy.

Fluent DSL Generator. The Fluent DSL generator transforms a fluent API into an equivalent external DSL. The generator included in DSLit is able to handle simple fluent APIs like the jRTF (Fig. 2).

The API metamodel generated from a Fluent API is composed by operations defined over the classes of the metamodel. For each class, a grammar rule is created. It contains a set of alternatives that are the grammar rules corresponding to the operations for that class. Inside these rules, each operation parameter is mapped to a feature that, depending on the parameter cardinality, can be optional or a list. In addition, an optional feature *next* is defined, that expands to the rule representing the return type of the operation.

It is important to note that in the Fluent DSL, all the names of the operations defined in the classes of the metamodel appear as terminal in the grammar.

```
Program:
    statements+=Statement*;
Statement:
    Declaration | Assignment;
Declaration:
    var=Var;
Assignment:
    left=VariableExpression '=' right=Expression;
Expression:
    VariableExpression | Literal;
VariableExpression:
     var=ReferenceVar ('.' attr=Attribute)?;
ReferenceVar:
    value=[Value];
Literal:
     FloatLiteral | StringLiteral | ...;
Var:
    name=ID ':' type=Type;
Attribute:
    name=__AllAttributeNames__;
Type:
    name=__AllClassNames__;
```

Fig. 6. Excerpt of the DSL Template for the SimpleJava DSL generator

Fig.5 shows the generation of a Fluent DSL for a small subset of the jRTF API. The classes *Rtf* and *RtfPara* are mapped to the corresponding rules; while the operations *rtf()*, *section(RtfPara)* and *out()* are mapped to rules which names follow the schema: *className-operationName-parameterNames-returnType*.

In particular, *Rtf_rtf* contains as terminal the name of the operation *rtf()* and an optional feature *next*, that represents the return type *Rtf* of the operation *rtf()*. *Rtf_section _paragraphs_RtfPara* contains the terminal *section*, a list of paragraphs that expand the rule *RtfPara* (not shown in the example) and the optional feature *next*. Finally, *Rtf_out* contains only the terminal *out*, since the operation *out()* has neither parameters nor a return type that is a class of the input API metamodel.

SimpleJava DSL Generator. In the case in which the API does not suit one of the previous templates, the user has still the possibility to generate a fallback DSL. Since no assumptions can be made on the structure of the API and on its intended use, the generated DSL needs to offer similar capabilities to the Java language. The SimpleJava DSL generator produces a DSL that resembles Java but is restricted to the use of the analyzed API. The generated DSL in this case has the purpose of being a starting point for DSL development, since 1) the DSL developer can easily add domain-specific features to the generated environment, 2) moving the domain information to the linguistic level makes it more suitable to automated analysis.

The SimpleJava template (an excerpt is shown in Fig.6) defines typical Java concepts, e.g. *Declaration* and *Assignment*. The transformation *API2Grammar* expands the SimpleJava template with the information contained within the API metamodel. It is important to note that names of classes, attributes and references in the API metamodel are inserted in the template respectively as alternatives of the grammar rules *Type* and *Attribute*.

As mentioned above, the SimpleJava template does not make any assumption on the structure of the API. Therefore, it is applicable to any API, including those for which other templates such as Plain Old Data structure, and Fluent DSL are applicable. Fig.7 revisits the Swing and jRTF examples: the DSL code samples of Fig.1 and 2 are now expressed in textual syntaxes derived using the SimpleJava template. This figure further illustrates how SimpleJava works.

SimpleJava for Swing

```
f : Frame
f.title = "My Java Application"
f.width = 500
f.height = 300
l : Label
l.text = "Hello World!"
f.content = l
f.visible = true
```

SimpleJava for jRTF

```
r : Rtf
s : Section
r.section = s
p1 : RtfPara
p1.contents = "first paragraph"
p2 : RtfPara
t : Tab
p2.contents = t
p2.contents = " second par "
b : Bold
b.contents = "text in bold"
p2.contents = b
p2.contents = " and "
i : Italic
u : Underline
u.contents = "italic underline"
i.contents = u
p2.contents = i
```

Fig. 7. SimpleJava template applied to Swing and jRTF

4.2 Development-Environment Generation

Once the DSL grammar is generated, Xtext is able to produce several artifacts composing a DSL development environment. In particular, it offers 1) a *proposal provider* that provides a list of accessible keywords according to the current terminal of the grammar (i.e. content assist) and 2) a *validator* performing static analysis during editing. However, since we know the semantics of the DSL template that has been used to generate the grammar, we can automatically derive improved versions of such environment components by mixing the DSL domain-independent part, that comes from the template structure, and the DSL domain-specific part, that is inferred from the API metamodel.

In our prototype, these improved components are generated for the POD and SimpleJava DSLs and can be can be eventually redefined by the developer if needed. For the Fluent DSL, since the proposal provider and validator are embedded in the structure of the grammar (i.e., how the feature of a grammar rule expands in other rules), we rely instead on the Xtext default components.

4.3 Compiler Generation

Since the semantics of the DSL template is well-defined, a DSL instance can be transformed into its equivalent in Java. For instance concepts like *Declaration*, *Assignment* and *Statement* in the SimpleJava template (Fig.6) have a one-to-one correspondence with Java programming language's constructs. Xtext provides the capability to generate a model-representation of the DSL grammar according to the Xtext metamodel. Such a DSL model is then transformed to a Java model leveraging on MoDisco[8] that is in turn translated to a Java readable file using Acceleo[9], a model-to-text transformation tool.

5 Related Work

The work presented in [11] about Framework-Specific Modeling Languages studies how to identify and extract domain-specific knowledge from APIs. Some of the ideas

[8] http://www.eclipse.org/gmt/modisco/technologies/J2SE5/
[9] http://www.eclipse.org/acceleo

in that work inspired our research. Works such as [12] investigate current analysis techniques to understand APIs and extract some usage patterns. Such studies could complement ours in identifying specific API features and therefore improve our process.

Integration between the model and the API TS has been considered in works such as [9,10,13], where approaches to define bridges between these two TSs are presented. However, none of them enables the generation of a DSL from an API definition nor the selection of an appropriate structure for the resulting DSL.

Existing approaches such as METABORG [14], SugarJ [15] and Helvetia[10] enable the definition of internal DSLs and the corresponding generation of the domain-specific environment in the host language. Instead, our approach targets external DSLs. Furthermore ours can automatically generate a DSL definition for an API, which could be used as input for the aforementioned approaches.

Some GPLs with flexible concrete syntaxes like Haskell or Ruby enable direct definitions of DSLs directly using GPL syntax. An example of such a DSL is given in [16]. One problem with this kind of approaches is that DSL concrete syntax must be a subset of GPL concrete syntax (i.e., DSL syntax must be valid GPL code). Another problem is that the corresponding API has to be defined specifically so that GPL syntax may be used directly as a DSL. Therefore, compromises must be made on both API and DSL.

In [17] the authors evaluate 10 different approaches to implement DSLs, concluding that embedded DSLs are the simplest to implement. Our approach could be considered as an additional approach with which a significant amount of DSL customization is attainable at a comparatively low cost: (a) it provides a specific textual syntax not limited by a host GPL and (b) it kickstarts a DSL tooling ready to be used for the DSL.

The approach presented in [18] provides abstractions for repeatedly used patterns. Instead, we target on abstracting API calls into DSL constructs. However, both approaches could be combined so that simple DSL construct could be mapped to complex patterns of API usage.

6 Conclusion and Future Work

In this paper we have presented an approach to automatically generate an external DSL out of an object-oriented API by transforming in-language abstraction into linguistic abstraction. The generation process has been presented as a bridge between the involved technical spaces (i.e., API TS, Model TS, and Grammar TS) and uses a template mechanism, which allows customization of the resulting DSL. Our approach has been implemented on the Eclipse platform, as a plugin called DSLit, which generates an Xtext-based textual DSL out of Java-based APIs. The current prototype incorporates two templates that generate specific DSL structures (POD and Fluent) as well as a fallback template covering a subset of Java (SimpleJava).

In future work we plan to study how our method could cope with APIs that allow custom code extension (e.g., providing implementations of interfaces or abstract classes). We would also like to define more templates allowing for different types of DSLs, which in turn will need a deeper study on API characterization. Another possibility to explore is the generation of DSL interpreters instead of compilers. They would

[10] `http://scg.unibe.ch/research/helvetia`

for instance make it possible to load and execute DSL code at runtime. Finally, since a GPL allows interleaving calls to distinct APIs, one open question to study is how the generated DSLs may be combined in order to achieve the same kind of interleaving achievable with a GPL.

References

1. Lédeczi, A., Bakay, A., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. Computer **34**(11) (2001) 44–51
2. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer (2008)
3. Fowler, M.: Domain-Specific Languages. Addison-Wesley (2010)
4. Sánchez Cuadrado, J., García Molina, J.: A model-based approach to families of embedded domain-specific languages. IEEE Trans. Softw. Eng. **35**(6) (2009) 825–840
5. Sánchez Cuadrado, J., Cánovas Izquierdo, J.L., García Molina, J.: Comparison Between Internal and External DSLs via RubyTL and Gra2MoL. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. (2013) 109–131
6. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (2005) 316–344
7. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: SPLASH. (2010) 307–309
8. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces : an Initial Appraisal. In: DOA. (2002) 1–6
9. Cánovas Izquierdo, J.L., Jouault, F., Cabot, J., García Molina, J.: API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering. Inform. Software Tech. **54**(0) (2012) 257–273
10. Cuadrado, J.S., Guerra, E., de Lara, J.: The program is the model: Enabling transformations@ run. time. In: SLE. (2013) 104–123
11. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of Framework-Specific Modeling Languages. IEEE Trans. Softw. Eng. **35**(6) (2009) 795–824
12. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API Property Inference Techniques. IEEE Trans. Softw. Eng. (2012) 613–637
13. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting runtime software architecture: A bidirectional-transformation-based approach. J. Syst. Software **84**(5) (2011) 711–723
14. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: SPLASH. (2004) 365–383
15. Erdweg, S., Rendel, T., Kastner, C., Ostermann, K.: SugarJ: Library-based Syntactic Language Extensibility. In: OOPSLA. (2011) 391–406
16. Cunningham, H.C.: A little language for surveys: constructing an internal dsl in ruby. In: ACMSE. (2008) 282–287
17. Kosar, T., Barrientos, P.A., Mernik, M., et al.: A preliminary study on various implementation approaches of domain-specific language. Inform. Software Tech. **50**(5) (2008) 390–405
18. Chodarev, S., Pietriková, E., Kollár, J.: Towards Automated Program Abstraction and Language Enrichment. In: SLATE. (2013) 51–64