Manipulating Models Using Internal Domain-Specific Languages

Filip Krikava, Philippe Collet, Robert France

Manipulating Models Using Internal Domain-Specific Languages

Filip Křikava
University Lille 1 / LIFL
Inria Lille - Nord Europe, France
filip.krikava@i3s.unice.fr

Philippe Collet
Université Nice
Sophia Antipolis, France
I3S - CNRS UMR 7271
philippe.collet@unice.fr

Robert B. France
Colorado State University
Fort Collins, USA
CS Department
france@cs.colostate.edu

ABSTRACT

In Model-Driven Engineering, a number of external *Domain-Specific Languages* (DSL) for model manipulation have been proposed. However, they require users to learn new languages that, together with their execution performance, usability and tool support limitations, can significantly contribute to accidental complexities. In this paper, we present an alternative approach based on internal DSLs in Scala for model consistency checking and model transformations for the Eclipse Modeling Framework.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods

Keywords

domain-specific languages; model-driven engineering; model manipulation; model transformation; scala

1. INTRODUCTION

Model manipulation languages play an important role in Model-Driven Engineering. There have been many different languages and tools proposed to support model consistency checking and model transformations, particularly within the *Eclipse Modeling Framework* (EMF)[1]. While EMF models can be directly manipulated using *General Purpose Programming Languages* (GPLs) such as Java, GPLs do not allow developers to conveniently express model manipulation concepts and the loss of abstraction may give rise to accidental complexities [9]. Therefore, a number of different *Domain-Specific Languages* (DSLs) for EMF model manipulation have been proposed including the OMG standards[2], the Epsilon project[3], Kermeta[4] or ATL[5]. These external DSLs are built from the ground up allowing developers to manipulate models using higher-level abstractions enhancing language expressiveness and ease of use.

However, there are several shortcomings. First, a user has to learn and use potentially multiple similar, but not always consistent languages [4], which requires considerable time to learn [1]. The user might feel limited by more specific, but less versatile language constructs, by the language execution performance or by the provided tools. Second, evolving and maintaining complex external DSLs requires significant effort [2]. Not only does it require domain knowledge and language development expertise, but also involves significant language and tool engineering work [7, 2].

Facing these issues we propose an alternative internal DSL approach, whereby model manipulation constructs are embedded into a GPL. An internal DSL leverages the constructs and tools of its host language and thus considerably less effort is needed to develop and evolve the DSL. We use Scala[6], a statically typed object-oriented and functional programming language, to implement a family of internal DSLs for lightweight model consistency checking and model transformations. The DSLs have similar expressiveness and features found in external model manipulation DSLs, while providing good performance, compact implementation and the ability to take advantage of the Scala tool support.

In the rest of the paper, we present the model manipulation internal DSLs that have been realized as a Scala library called SIGMA[7]. The complete examples and documentation is available at the SIGMA project website.

2. NAVIGATION AND MODIFICATION

Essentially, any task-specific model manipulation is based on a set of basic facilities for model *navigation* and *modification* [3]. The majority of the external DSLs builds on a subset of *Object Constraint Language* (OCL) which, despite its limitations [5], provides an expressive way for navigating models. Therefore, a prerequisite for building any task-specific language is to provide similar OCL-like constructs. For example, let us consider a simplified *Object-Oriented* (OO) model[8]. Retrieving names of all package elements stereotyped as singleton can be expressed using the following OCL query:

```
pkg.ownedElements
->select(e|e.stereotypes->exists(s|s.name = 'singleton'))
->collect(e|e.name)
```

[1] http://www.eclipse.org/modeling/emf/
[2] http://www.omg.org/spec/index.htm#M&M
[3] http://www.eclipse.org/epsilon/
[4] http://www.kermeta.org
[5] http://www.eclipse.org/atl/
[6] http://scala-lang.org/
[7] http://fikovnik.github.io/Sigma
[8] http://fikovnik.github.io/Sigma/PL14.html

SAC'14 March 24–28, 2014, Gyeongju, Korea

Thanks to Scala flexible syntax (omitting dots and infix operator syntax for method invocations), implicit conversions (extending existing types with new behavior), higher-order functions support and type inference, the very same expression can be rewritten to:

```
pkg.ownedElements
.filter(e=>e.stereotypes exists(s=>s.name == "singleton"))
.map(e=>e.name)
```

In order for this to work, each EMF model class has to be accompanied with an auxiliary class that aligns EMF properties accessors and EMF collections with the one of Scala. Currently, these classes are automatically generated, but with the coming support of type macros in the next version of Scala, the generation will no longer be necessary.

For model modification, there is no support in OCL, but the task-specific DSLs extend it with imperative constructs. Relying on the generated auxiliary classes, we provide similar facilities that form a convenient way to author complete EMF models directly in Scala:

```
val cls = Class(name = "MyClass")
cls.stereotypes += Stereotype(name = "singleton")
cls.features += Operation(name = "getInstance",
  returnType = cls)
```

3. MODEL CONSISTENCY CHECKING

Model consistency checking determines whether a model element or a relation between model elements satisfies certain restrictions expressed as a structural constraints. For example, the following listing checks whether a given OO class stereotyped as a *singleton* defines the appropriate `getInstance` method.

```
1  class Singleton extends ValidationContext
2    // express constraints as methods
3    with InvMethods
4    // generated support for navigation and modification
5    with OOPackageScalaSupport {
6
7    type Self = Class // define the context type
8    override def guard = { // context guard
9      self.stereotypes exists (_.name == "singleton")
10   }
11
12   def invHasGetInstance =
13     getGetInstanceOperation(self) != null
14
15   def invGetInstanceIsStatic = guardedBy {
16     self satisfies invHasGetInstance
17   } check {
18     val op = getGetInstanceOperation(self)
19     if (op.ownerScope == ScopeKind.SK_CLASSIFIER) Passed
20     else Error(s"${self.name}.getInstance must be static")
21       .quickFix(s"Make ${self.name}.getInstance static") {
22         op.ownerScope = ScopeKind.SK_CLASSIFIER
23       }
24   }
25
26   def getGetInstanceOperation(c: OOClass): Operation =
27     c.operations find (_.name == "getInstance") orNull
28 }
```

In Sigma, the constraints are organized in classes, called *validation contexts*, that group together related constraints under the same dependency. Each context specifies the type of instances it applies to (line 7). Additionally, the subset of instances can be further narrowed by adding a *context guard* (line 8). Similarly to OCL, `self` represents the current model instance being checked.

Invariants are represented as methods that take no parameters. They can be simple boolean expression such as in OCL (lines 12-13) or more sophisticated (lines 15-24) providing an opportunity to give a *severity level* with a detailed *user feedback* (line 20) and automated *inconsistency repair fixes* (line 21).

4. MODEL TRANSFORMATIONS

Model transformations provide necessary support for translating models into other models or into text. Essentially, they map source model elements into corresponding target model elements or into textual fragments.

Model-to-Model Transformation (M2M). Sigma provides a dedicated internal DSL that combines imperative features with declarative rule-based execution scheme into a hybrid M2M transformation language. A M2M transformation is represented as a class and transformation rules are expressed as methods. Method signatures denote the source elements and the transformation targets. Rule can also use guards to limit their applications. Similarly to other M2M transformation languages, Sigma also supports an *abstract* (rule inheritance) and *lazy* (explicitly called) rules. The following listing shows an example of a simple OO model to database schema transformation.

```
1  class OO2DB extends M2M with RuleMethods
2    with OOPackageScalaSupport with DBPackageScalaSupport{
3
4    def ruleClass2Table(cls: Class, tab: Table, pk: Column) {
5      tab.name = cls.name;
6      tab.columns += pk
7      tab.columns ++= ~cls.properties
8
9      pk.name = "Id"
10     pk.type_ = "Int"
11   }
12   def ruleProperty2Column(prop: Property) = guardedBy {
13     // prevent transformation of multi-valued properties
14     !prop.multi && prop.type_.isInstanceOf[PrimitiveType]
15   } transform {
16     Column(prop.name.toUpperCase, prop.type_.name)
17   }
18 }
```

A common operation in M2M transformation is relation of the target elements that have been already (or can be) transformed from source elements. For this purpose, the DSL provides a unary operator ~ (tilde) that can be applied to both a single model element instance and their collection. It consequently either returns already transformed elements or invokes the appropriate transformation rule (*e.g.* line 6 and 7 trigger `ruleProperty2Column` invocation).

Model-to-Text Transformation (M2T). The M2T internal DSL is using the code-explicit form (escaping the output text). For the parts where there is more text than logic we rely on Scala multi-line string literals and string interpolations allowing one to embed variable references and expressions directly into strings. For example, the listing below is an excerpt of a simple OO class to Java transformation.

```
1  class OO2Java extends M2T with OOPackageScalaSupport {
2
3    type M2TSource = Class // input type for transformation
4
5    // entry point - main template
6    def execute =
7      !s"public class ${root.name}" curlyIndent {
8        !endl // extra new line
9        for (o <- root.operations) {
10         genOperation(o) // call to another template
11         !endl // extra new line
12       }
13     }
14
15   def genOperation(o: Operation) =
16   !s"public ${o.retType.name} ${o.name}()" curlyIndent {
17     !s"""
18     // TODO: should be implemented
19     throw new UnsupportedOperationException("${o.name}");
20     """
21   }
22 }
```

Following the same pattern, a M2T transformation is a Scala class consisting of a set of templates that are represented as methods. The transformation starts in an entry point method (line 6), from which it is split and logically organized into smaller templates in order to increase modularity and readability (line 15). The most common operation in a M2T transformation is a text output. A convenient way to output text in the DSL is through a unary ! (bang) operator that is provided on strings (*e.g.* line 7).

An important aspect of any M2T transformation language is the template readability such as layout and indentation. The internal DSL maintains it through dedicated support for *decorators*, *smart whitespace* handling and *relaxed newlines*. Decorators are nestable string operations that reformat a given block (*e.g.* `curlyIndent` decorator on line 7 wraps the body into a pair of curly brackets indenting each line). The smart whitespace handler removes extra whitespace from multi-line strings that are there only for the template readability (*e.g.* white spaces on lines 18 and 19 will be discarded). Relaxed newlines automatically appends a new line after every text output, again increasing the template readability.

5. DISCUSSIONS

Applications. The presented SIGMA model manipulation DSLs have been used to implement an EMF-based MDE environment for experimenting with self-adaptive software systems [6]. It has been also adopted by the Yourcast[9] project for M2T transformations replacing Velocity[10] and plain Java templates, gaining 20% code size reduction mainly due to more expressive model navigation and more compact text outputting constructs.

Assessment. Here we summarize the main SIGMA properties for which we consider it to be a *lightweight* approach to model manipulation as opposed to the more heavy-weight external DSLs: (1) It relies only on Scala and does not require any special runtime environment as opposed to the interpreted DSLs. Executing a SIGMA model manipulation task is no different from executing a regular JVM-based application. This simplifies the integration into software projects since, unlike the external DSLs, no specialized support is required. (2) SIGMA requires only basic set of Scala skills. The DSLs are rather small and thus less learning effort is likely to be needed in comparison to languages such as OCL or Epsilon. (3) It has small API, relying on existing Scala concepts and functionalities. All DSLs support common features such as *modularity* to allow for organizing task-specific concerns into modules, their *reuse* without the need to duplicate them, and meta-model *extensions* (*i.e.* helper methods and queries). Furthermore, the expressions static safety, yet with a clear syntax similar to dynamically typed languages. (4) All the DSLs are built on top of the same host language using the same pattern. They are interoperable, consistent and it is also possible to embed them together (*e.g.* execute a M2T transformation from a M2M transformation). (5) Its performance is close to the one of Java. SIGMA compiles directly into Java bytecode and therefore outperforms interpreted external DSLs. There is a small overhead, less than 10%, cased by SIGMA in comparison with model manipulation done in pure Scala. (6) It is testable with any Java unit testing framework. The method-based style of the DSLs allows to cherry-pick the fragments of model manipulation to be tested, which is especially useful for larger manipulations.

(7) Finally, one of the main advantages of an internal DSL is that it can directly reuse the host language tool support including editor and debugger. Being an internal DSL is also notably reflected in the implementation size. SIGMA is currently implemented in 3500 lines of Scala code, which is an order of magnitude less than the Epsilon Object Language or Kermeta, which is an order of magnitude less than Eclipse OCL.

Limitations. Apart from the syntax limitations, an internal DSL is in general a leaky abstraction and traditionally, the support for domain-specific error checking and optimization is difficult to realize [2]. For example structural constraints can contain arbitrary code and there is no simple way to make sure they are side-effect free without using an external checker such as IGJ[11]. Finally, depending on the target audience, the use of Scala can be seen as a drawback rather than a merit. It has not yet reached the popularity of some of the mainstream programming languages and thus it might be difficult to justify its learning solely for the purpose of model manipulation.

Further Work. Current work in progress consists in carrying out more evaluations to further assess the usability of the proposed DSLs. Further, we want to explore the Scala advanced DSL embedding techniques to address some of the limitations outlined above. Concretely, to use language virtualization and staging [8] to build, check and optimize intermediate representations of the DSL application using different semantics.

Acknowledgments. The authors thank Sébastien Mosser for valuable comments. This work is partly funded by the ANR SALTY project under contract ANR-09-SEGI-012.

6. REFERENCES

[1] D. Akehurst, B. Bordbar, M. Evans, W. Howells, and K. McDonald-Maier. SiTra: Simple Transformations in Java. In *MoDELS 2006*, 2006.

[2] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. *International conference on Object oriented programming systems languages and applications*, 2010.

[3] D. Kolovos, R. Paige, and F. Polack. The Epsilon Object Language (EOL). *Model Driven Architecture – Foundations and Applications*, LNCS 4066, 2006.

[4] D. Kolovos, R. Paige, and F. Polack. The Epsilon Transformation Language. In *International Conference on Model Transformations*, 2008.

[5] F. Krikava and P. Collet. On the Use of an Internal DSL for Enriching EMF Models. In *International Workshop on OCL and Textual Modelling*, 2012.

[6] F. Krikava, P. Collet, and R. B. France. ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures. In *Dependable and Adaptive Distributed Systems track, SAC'14*, 2014.

[7] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[8] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Generative programming and component engineering*, 2010.

[9] D. C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[9] http://www.yourcast.fr/

[10] http://velocity.apache.org/

[11] http://types.cs.washington.edu/checker-framework/