



HAL
open science

SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations

Filip Krikava, Philippe Collet, Robert France

► **To cite this version:**

Filip Krikava, Philippe Collet, Robert France. SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations. MODELS - 17th International Conference on Model Driven Engineering Languages and Systems, Sep 2014, Valencia, Spain. hal-01010339

HAL Id: hal-01010339

<https://inria.hal.science/hal-01010339v1>

Submitted on 10 Jul 2014

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations

Filip Křikava¹, Philippe Collet², and Robert B. France³

¹ University Lille 1 / LIFL Inria Lille,
Nord Europe, France,
filip.krikava@inria.fr

² Université Nice Sophia Antipolis / I3S - CNRS UMR 7271,
06903 Sophia Antipolis, France
philippe.collet@unice.fr

³ Colorado State University - Computer Science Department,
Fort Collins, CO 80523, USA
france@cs.colostate.edu

Abstract. Model manipulation environments automate model operations such as model consistency checking and model transformation. A number of external model manipulation *Domain-Specific Languages* (DSL) have been proposed, in particular for the *Eclipse Modeling Framework* (EMF). While their higher levels of abstraction result in gains in expressiveness over general-purpose languages, their limitations in versatility, performance, and tool support together with the need to learn new languages may significantly contribute to accidental complexities. In this paper, we present SIGMA, a family of internal DSLs embedded in Scala for EMF model consistency checking, model-to-model and model-to-text transformations. It combines the benefits of external model manipulation DSLs with general-purpose programming taking full advantage of Scala versatility, performance and tool support. The DSLs are compared to the state-of-the-art Epsilon languages in non-trivial model manipulation tasks that resulted in 20% to 70% reduction in code size and significantly better performance.

1 Introduction

Model manipulation languages and tools provide support for automating model operations such as model consistency checking, and *model-to-model* (M2M) and *model-to-text* (M2T) transformations [41]. A number of different model manipulation technologies have been proposed, particularly within the *Eclipse Modeling Framework* (EMF) [43]. The EMF models can be manipulated directly in Java, however, a *General Purpose Programming Language* (GPL) such as Java does not conveniently express model manipulation concepts and the loss of abstraction can give rise to accidental complexities [40]. Therefore, a number of external *Domain-Specific Languages* (DSLs) for EMF model manipulation have been proposed, *e.g.*, the OMG standards including OCL [34] for navigating and expressing constraints on models, QVT [33] and MOFM2T [32] for model transformation; the Epsilon project [36] with an extensive family of model manipulation DSLs; Kermeta [31], a single, but more general imperative language for

all model manipulation tasks; and ATL [20], a M2M transformation language. External model manipulation DSLs provide language constructs that allow developers to manipulate models using higher-level abstractions. This should result in higher expressiveness and ease of use in comparison to GPLs [29].

However, there are several impediments to such approaches. Even for a simple model manipulation task users have to learn one or more new languages and tools, which may require considerable effort [11]. Users might feel limited by the more specific, but less versatile language constructs, by the language execution performance or by the provided support tools [18]. In most cases the languages build on a subset of OCL concepts for model navigation and model consistency checking. Despite that, there are well known inconsistencies, interoperability and reusability issues among these languages [22,23]. Finally, the large dependency stacks associated with these languages can make their integration into existing software projects rather challenging.

A notable exception is the Epsilon project, which alleviates some of these issues. Epsilon provides an extensive family of model management languages and tools such as *Epsilon Validation Language* (EVL) [24], *Epsilon Transformation Language* (ETL) [22], and *Epsilon Generation Language* (EGL) [38]. These task-specific languages are based on a common OCL-like expression language called EOL [23]. While this currently makes it one of the most complete language workbenches for model manipulations, we identify several shortcomings. EOL is a dynamically typed language, providing little compile time checking. Consequently, IDE features such as content assists, static checking or refactoring are rather basic in comparison to what is provided by the other approaches that use static typing. EOL lacks certain programming constructs that makes the code unnecessary lengthy in particular in the case of non-trivial model manipulations. Moreover, Epsilon DSLs are interpreted and their performance is an order of magnitude slower than the compiled languages, but they are also slower than the Eclipse implementation of the OMG stack [25]. As a result, these shortcomings also give rise to some accidental complexities, albeit of a different nature than those associated with GPLs.

These issues are not easy to alleviate. The need to provide a lot of GPL-like constructs together with the necessity of some level of Java interoperability make the external DSLs large and complex. Evolving and maintaining complex DSLs is known to be hard since it not only requires domain knowledge and language development expertise, but also involves significant language and tool engineering effort [29,13]. In this paper we propose an alternative internal DSL approach whereby model manipulation constructs are embedded into a GPL. The intent is to provide an approach that developers can use to implement many of the practical EMF model manipulations within a familiar environment with reduced learning overhead and improved usability.

An internal DSL leverages the constructs and tools of its host language. For this approach to be effective, the host GPL must be flexible enough to allow definition of domain-specific constructs. We thus use Scala [35], a statically typed object-oriented and functional programming language, to implement a family of internal DSLs, called SIGMA [28], for model consistency checking and model transformations. In this paper, our contribution is to evaluate the resulting DSLs,

comparing their expressiveness and features to corresponding Epsilon DSLs in several non-trivial model manipulation tasks. We observe this results in 20% to 70% reduction in code size and significantly better performance.

The remainder of the paper is organized as follows. In Section 2 we give a quick overview of the SIGMA languages family. Section 3 develops the common infrastructure for model navigation and modification. This is used for model consistency checking described in Section 4, M2M transformations described in Section 5 and M2T transformations described in Section 6. In Section 7, we overview the current implementation and provide an evaluation of SIGMA. Finally Section 8 discusses related work and Section 9 concludes the paper.

2 SIGMA Overview

SIGMA is a family of internal DSLs for model manipulation that were created with the aim to alleviate some of the main limitations of the currently proposed approaches. It is thus not proposing new concepts in model manipulation languages, but instead providing the existing concepts with the following main requirements: (1) *Epsilon-like features and expressiveness*, (2) *competitive performance*, (3) *usable tool support*, (4) *simple testability with existing unit frameworks*, and (5) *simple integration into existing EMF projects*. We chose Epsilon since it represents the state-of-the-art model manipulation languages with proven features essential for usable model manipulation. Furthermore, it is also presented as an approach that addresses most of the shortcomings of the other external model manipulation DSLs (details in Kolovos *et al.* [22,24,23] and Rose *et al.* [38]).

SIGMA DSLs are embedded in Scala [35], a statically typed production-ready GPL that supports both object-oriented and functional style of programming. It uses type inference to combine static type safety with a “*look and feel*” close to dynamically typed languages. It is interoperable with Java and it has been designed to host internal DSLs [13]. Furthermore, it is supported by the major integrated development environments.

A typical way of embedding a shallow DSL into Scala is by designing a library that allows one to write fragments of code with domain-specific syntax. These fragments are woven within Scala own syntax so that it appears different [16]. Next to Scala flexible syntax (*e.g.* omitting semicolons and dots in method invocations, infix operator syntax for method calls, etc.), it has a number of features simplifying DSL embedding such as implicit type conversions allowing one to extend existing types with new methods, mixin-class composition (*i.e.* reusing a partial class definition in a new class) [35], and lifting static source information with implicit resolutions to customize error messages in terms of the domain-specific extensions using annotations [30]. Furthermore, Scala supports compile-time meta-programming allowing for code self-optimization and to reduce boilerplate code generation.

Figure 1 depicts the general organization of the SIGMA DSLs. The use of EMF models in SIGMA is facilitated by a dedicated support layer that underneath uses the default EMF generated Java classes and the EMF API (implementation details are given in Section 7.1). This layer provides a convenient model

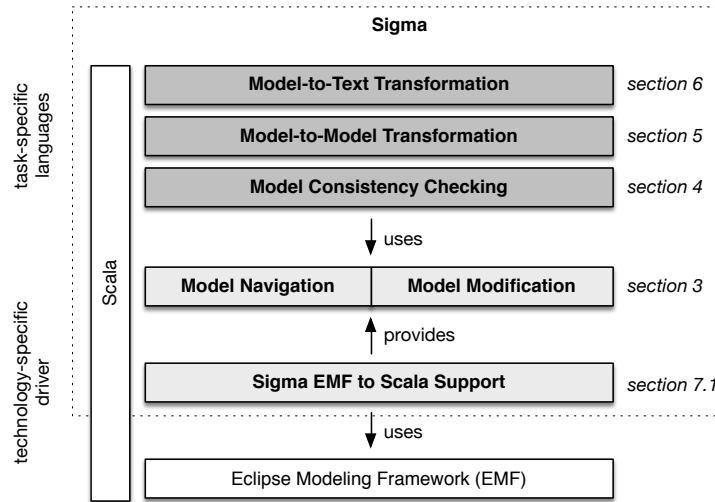


Fig. 1. Sigma EMF to Scala Support

navigation and modification support forming a common infrastructure for the task-specific internal DSLs. While currently SIGMA targets the EMF platform, other meta-modeling platforms could be used since the task-specific languages are technology agnostic (*cf.* Section 7.1).

In the following sections we detail the common infrastructure and the different task-specific DSLs. We deliberately skip some technical details about how certain DSL constructs are implemented. The complete examples with further documentation are available at the project web site [8]. For illustration purposes, in the following sections we consider a simplified *Object-Oriented* (OO) model (*cf.* companion web page [4]).

3 Common Infrastructure

Essentially, any model manipulation technique is based on a set of basic operations for model *navigation* (*e.g.* projecting information from models) and *modification* (*e.g.* changing model properties or elements) [23]. In this section we show their realization in SIGMA for EMF based models. Implementation details are discussed in Section 7.1.

Model Navigation. The model navigation support provides OCL-like expressions for convenient model querying. For example, retrieving names of all OO package elements stereotyped as singletons can be expressed using the following OCL query:

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

In SIGMA, the very same query can be expressed almost identically to OCL:

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

In both versions the `singletons` type is inferred from the expression.

While navigating models, one often need to filter the types of the objects to be kept during navigation. For example, selecting the operations of a package abstract classes corresponds to the following OCL expression:

```
pkg.ownedElements
->select(e | e.oclIsKindOf(Class) and e.oclAsType(Class).abstract = true)
->collect(e.oclAsType(Class).operations)
```

This is rather verbose with the recurring pattern of `oclIsKindOf/oclAsType`, which makes longer queries hard to read. Scala, on the other hand, provides support for pattern matching that can be used in combination with partial functions to obtain the following SIGMA code⁴:

```
pkg.ownedElements collect { case c: Class if c.abstract_ => c.operations }
```

In addition SIGMA can also prevent null pointer exceptions when navigating over potentially unset references and attributes (0..1 multiplicity). It wraps them into Scala `Option` type, a container explicitly representing an optional value, which consequently forces one to always check for the presence of the value.

Model Modification. The model modification support provides facilities for seamless creation, updates and removal of model elements. By design, the OCL does not have model modification capabilities, but in Epsilon for example, an OO singleton class can be created using the code in Listing 1.1. Using SIGMA the same model instance is created in Listing 1.2.

```
var cls = new Class;
cls.name = "MyClass";

var singleton = new Stereotype;
singleton.name = "singleton";
cls.stereotypes.add(singleton);

var op = new Operation;
op.name = "getInstance";
op.returnType = cls;
cls.operations.add(author);
```

Listing 1.1. EOL

```
val cls = Class(name = "MyClass")
val singleton = Stereotype(name = "singleton")
cls.stereotypes += singleton

val op = Operation(name = "getInstance",
  returnType = cls)
cls.features += op
```

Listing 1.2. SIGMA

These methods provide a convenient way to author complete EMF models directly in Scala. Additionally, SIGMA provides support for delayed initialization in the cases an element initialization should only happen after its containment, and for lazy resolution of contained references⁵.

4 Model Consistency Checking

Model consistency checking provides facilities to capture structural constraints as state invariants and to check model instances against these constraints. In OCL or EVL, a structural constraint is a boolean query that determines whether a model element or a relation between model elements satisfies certain restrictions.

⁴ The `_` suffix to `abstract` is automatically added by the SIGMA EMF support since it is a Scala keyword.

⁵ Technical details at <http://bit.ly/18javEY>

For example, in the OO model, an invariant may represent a restriction that within one package, there cannot be two classes having the same name. In SIGMA, such an invariant can be expressed as:

```

1 class ClassInvs extends ValidationContext with OOPackageSupport {
2   type Self = Class // context type
3
4   def invUniqueNamesWithinPackage =
5     self.pkg.ownedElements forall (e => e != self implies e.name != self.name)
6 }

```

Invariants are represented as regular Scala methods (line 4). They are organized into a validation context class (line 1) that specifies a context type (line 2), *i.e.*, the type of instances the invariants can be applied to. As in OCL, `self` represents the current instance that is being checked. The `OOPackageSupport` trait (*cf.* Section 7.1) mixes-in the Sigma EMF to Scala support for model navigation and modification (line 1). By organizing invariants as methods in classes we can simply reuse them through inheritance. Furthermore, it allows one to easily test invariants using any of the Java unit testing frameworks.

Listing 1.3 shows an extended OO class validation with additional features. A validation context class can narrow its applicability by providing a context guard (line 4). Invariant violation can distinguish different severity levels such as errors and warnings (line 10). In order to prevent meaningless evaluation of constraints whose dependencies are not satisfied, invariants can also have guards (line 7). Finally, a user can be provided with a feedback including a meaningful message (line 10), as well as means to repair the inconsistency with change suggestions over the affected model elements (line 11).

```

1 class ClassInvs extends ValidationContext with OOPackageSupport {
2   type Self = Class // context type
3
4   override def guard = self.annotations exists (_.name == "ignore") // context guard
5
6   def invUniqueNamesWithinPackage = guardedBy {
7     self satisfies invHasValidName // invariant guard
8   } check { // invariant body
9     self.pkg.ownedElements find (e => e != self && e.name == self.name) match {
10      case Some(c) => Error(s"Class $c has the name")
11        .quickFix("Rename '${self.name}' to '${self.name}_2'") { self.name += "_2" }
12      case None => Passed
13    }
14  }
15
16  def invHasValidName = // ...
17 }

```

Listing 1.3. Example of model consistency checking

5 Model-to-Model Transformations

M2M transformations provide necessary support for translating models into other models, essentially by mapping source model elements into corresponding target model elements. An imperative style of M2M transformation [15]

is already supported thanks to the common infrastructure layer described in Section 3. On the other hand, the lower level of abstraction of the imperative transformation style leaves users to manually address issues such as orchestrating the transformation execution and resolving target elements against their source counterparts [22]. Therefore, inspired by ETL and ATL, we provide a dedicated internal DSL that combines the imperative features with declarative rule-based execution scheme into a hybrid M2M transformation language.

Transformation rules constitute the abstract syntax of the M2M transformation DSL. Similarly to ETL or ATL, a rule defines a source and a target element to which it transforms the source. It may optionally define additional targets, but there is always one primary source to the target relation. A rule can also be declared as *lazy* or *abstract*. Each non-lazy and non-abstract rule is executed for all the source elements it is applicable. Lazy rules have to be called explicitly. When a rule is executed, the transformation engine initially creates all the explicitly defined target elements and passes them to the rule that populates their content using arbitrary Scala code. Similarly to consistency checking constraints, transformation rules can optionally limit their applicability by defining a guard.

```

1 class OO2DB extends M2M with OOPackageSupport with DBPackageSupport {
2
3   def ruleClass2Table(cls: Class, tab: Table, pk: Column) {
4     // standard Scala
5     tab.name = cls.name
6     tab.columns += pk
7     pk.name = "Id"
8     pk.type_ = "Int"
9
10    // SIGMA specific: target elements resolution
11    tab.columns += cls.properties.sTarget[Column]
12  }
13
14  def ruleProperty2Column(prop: Property, col: Column) = guardedBy {
15    !prop.multi // prevent transformation of multi-valued properties
16  } transform {
17    col.name = prop.name.toUpperCase
18    col.type_ = prop.type_.name
19  }
20 }

```

Listing 1.4. Example of M2M transformation

Listings 1.4 illustrates the internal DSL using the traditional example of OO model to database schema transformation⁶. A M2M transformation is a Scala class that extends the M2M base class with the generated package support traits for model navigation and modification (line 1). Transformation rules are represented by methods. For example, the `ruleClass2Table` denotes a rule that, for a given class, produces a table and a column (line 3). Additional target elements can be constructed within the rule body, but in such a case a developer is responsible for their proper containment.

During the M2M transformation, there is often the need to relate the target elements that have been already (or can be) transformed from source elements. For this purpose, SIGMA provides a set of operations. An example is shown on

⁶ While it is a worn example, it enables one to easily compare to similar examples provided by ETL and ATL *cf.* companion web page [4].

the line 6 where `sTarget[Column]` transforms class properties into columns. It does that by looking up a rule with *property-to-column* mapping, which in this case is the `ruleProperty2Column` rule. This operation can be applied both to a single instance as well as to a collection of model elements. Similarly, SIGMA includes support for resolving source elements from their corresponding targets.

6 Model-to-Text Transformations

M2T transformations translate models into text by mapping source model elements into corresponding textual fragments. We focus on template-based approach whereby string patterns are extended with executable logic for code selection and iterative expansion [15]. This approach is used by all the major M2T transformation languages including EGL and MOFM2T. In model-driven software development, the aim is to synthesize a running system implementation and therefore our primary focus is on generating source code artifacts.

Unlike EGL and Acceleo, our internal DSL for M2T transformation is using the code-explicit form, *i.e.*, it is the output text instead of the transformation code that is escaped. This is one of the syntax limitations that cannot be easily overcome. On the other hand, from our experience, in non-trivial code generations, the quantity of text producing logic usually outweighs the text being produced. 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.

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

Listing 1.5. Example of M2T transformation

Listing 1.5 shows an example of OO class to Java transformation⁷. Following the same pattern, a M2T transformation is a Scala class extending from the M2T base class (line 1). Line 2 defines the type of model element, *i.e.*, the transformation source. A M2T transformation consists in a set of templates that are represented as methods (lines 4 and 11). The `execute` method is the entry point, which will be invoked when the transformation is executed. Usually, from

⁷ Similar examples are provided for both EGL and Acceleo *cf.* companion web page [4].

there, a transformation is split and logically organized into smaller templates in order to increase modularity and readability.

The most common operation in a M2T transformation is a text output. A convenient way to output text in our DSL is through a unary ! (bang) operator that is provided on strings (*e.g.* line 4). The prefix `s` right before the string double quote denotes an interpolated string, which can include Scala expressions in a type-safe way.

An important aspect of any M2T transformation language is the template readability, *e.g.*, 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. For example, on line 4 we use `curlyIndent` decorator, that wraps its body into a pair of curly brackets and indent each line. Smart whitespace handler removes extra whitespace from multi-line strings that are there only for the template readability. For example the whitespaces prefixing the text on lines 14 and 15 will be discarded. Relaxed newlines loosen the necessity to output new line characters by doing it automatically after every text output. Both smart whitespace and newlines handlers are enabled by default, but can be turned off.

Finally, the DSL also allows one to fork new text sections. This makes it possible to output text into different locations at the same time. All sections are appropriately merged in the final text at the end of the transformation. This is useful for example for handling imports while generating Java code, as they can be resolved one-by-one during the model traversal.

7 Evaluation

Our aim is to propose an approach that improves the overall usability of model manipulations through DSLs. However, defining usability of a DSLs and associated tool support tend to be subjective, since it largely depends on the preferences and background of its users [41] and its improvement cannot be measured directly. Therefore, we structure the evaluation as follows. First we give details about the implementation and current applications. Next, we compare SIGMA DSLs to their corresponding Epsilon counterparts with regard to implementation effort measured in terms of code size, performance and features. Finally, we discuss the limitations of the approach and threats to validity.

7.1 Implementation

SIGMA is implemented as a Scala library which is available from the project website [8]. Its task-specific languages are all relying on a common infrastructure.

Common infrastructure. The common infrastructure aligns EMF generated Java classes with Scala to enable use of model navigation and modification notation similar to OCL and EOL. This involves (1) model navigation without “*get noise*” (*e.g.* `getSuperPackage.getName` becomes `superPackage.name`), and (2) promoting EMF collections to corresponding Scala collections to benefit from convenient first-order logic operations (*e.g.*, `map`, `filter`, `collect`) similar

to OCL. Both issues are addressed by generating extension traits⁸ that make EMF model elements interoperable with Scala. These traits implicitly extend all model classes with property accessors without the `get` prefix and convert EMF collections into the corresponding Scala ones. The conversion only happens at the interface level leaving the underlying data storage unchanged. In the same way, existing Scala types are extended with missing OCL operations (*e.g.* `implies`). These traits are either generated by provided SIGMA M2T transformation executed explicitly by a user or implicitly using the experimental Scala macro annotations [12].

Task-specific languages. The model consistency checking and M2M transformation have similar abstract syntax (*i.e.* constraints and rules) to EVL and ETL respectively, and their execution semantics is the same as defined in Epsilon. The M2T transformation is a purely imperative DSL and as such it does not contain any particular execution engine. All DSLs are implemented following the same pattern, *i.e.* organizing task-specific concerns into Scala classes that extend from a task-specific base class. The main constructs such as invariants, rules and templates are expressed as methods in order to foster reuse and extensibility.

7.2 Applications

SIGMA has been used in the SALT project [7] to develop a modeling environment for developing self-adaptive software systems [27]. The main motivations were the shortcomings of OCL used for the initial implementation [26]. SIGMA has been also adopted by the Yourcast project [10] for M2T transformations replacing Velocity [1] and plain Java templates, gaining 20% reduction in code size, mainly thanks to more expressive model navigation and more compact text outputting constructs.

7.3 Code Comparison

In order to evaluate the overall usability of our approach we re-implemented larger Epsilon model manipulation tasks for each of the SIGMA DSLs. As suggested by the Epsilon community⁹, we chose Eugenia [3] GMF Ecore constraints for model consistency checking, Unicaneer2sql [9] (an ER to relational model transformation) for M2M transformation, and EglDoc [2] (an Epsilon tool for generating Ecore documentation in HTML including Graphviz diagram) for M2T transformation. The Eugenia and EglDoc comes directly from Epsilon, which should guarantee certain quality of the source code. The complete implementation is available from the paper companion web page [4].

Table 1 summarizes the implementation effort in terms of *Source Lines of Code* (SLOC) for the three scenarios, for both Epsilon and SIGMA. Interpreting SLOC metrics is always problematic. The issue of what is the right level of “*verbosity*” in a language is complex and should not be reduced naively to just counting SLOC. Our assumption, however, is that usability is not achieved by

⁸ Technical details at <http://bit.ly/18javEY>

⁹ <http://www.eclipse.org/forums/index.php?t=rview&goto=1235103>

Scenario	Epsilon	SIGMA	difference
Model consistency checking	364 EVL	286	22%
M2M transformation	733 ETL	389	47%
M2T transformation	1400 EGL	412	70%

Table 1. SLOC comparison

having fewer lines of code, but instead, by having more expressive and concise code, which is beneficial to writers as well as to readers. On the other hand, code bloat resulting from code duplication and from lack of constructs that enable the building of more concise but expressive statements, is not desirable.

Most of the code reduction comes from the fact that Scala contains more general programming constructs than EOL. In particular, pattern matching and inheritance helped to reduce many of the code duplications. In the case of model consistency checking, the code reduction is the least significant one since SIGMA contains the same constraints constructs as EVL and the invariant expressions were mostly simple first-order logic queries. The EVL code is therefore almost identical to the SIGMA one and the only reduction was in the EOL helper methods. They can be expressed more concisely in Scala primarily thanks to pattern matching.

The case of M2M transformation led to a similar situation. SIGMA supports the same M2M transformation rules and thus the ETL code is very close to the SIGMA one. However, the M2M transformation involved a lot of imperative EOL code which could be reduced by using more expressive Scala statements. Furthermore, about 15% of the transformation generates text for which we could use SIGMA M2T constructs reducing the code even further.

The M2T transformation scenario involved generating both HTML code and Graphviz code. Generally, code-explicit forms of M2T transformation are not particularly suitable for generating HTML code since in this case the quantity of text outweighs the text producing logic. However, despite this, the code has been reduced by 70%. The main reason is that, by the use of inheritance, a lot of code duplication present in the EGL templates was avoided.

We do not provide a comparison of the coding time. First we do not have the measures of the Epsilon versions. Second it is always easier to port an existing code to a new language than to write it from scratch. However, we expect some strong points of the SIGMA DSLs to reduce coding time. Static type safety prevents runtime typing errors. The highly usable tool support provided by the Scala IDE [5] with a code completion and a debugger and the ability to easily run and test the transformations should facilitate development.

7.4 Performance Comparison

The performance of SIGMA is determined by the host language one and the overhead of the SIGMA API. SIGMA compiles directly into Java byte-code and thus it significantly outperforms Epsilon and other interpreted DSLs. For example,

generating QVT meta-model (159 classifiers) documentation using Eglodoc takes on average 8 times more time than using the SIGMA version.

However one of the concerns is related to the extensive use of implicit type conversions and other Scala constructs that might have negative performance impact. Therefore, as a part of the performance evaluation, we have implemented the same M2T transformation in SIGMA and the related M2T languages¹⁰. In addition, we have also implemented it in pure Java and Scala with no additional libraries. The Java version is used as a performance baseline. The Scala version is used to measure the overhead of Scala in comparison to Java and the performance penalty caused by SIGMA. The implementation in the other languages aims at evaluating our requirement of competitive performance. The M2T transformation has been chosen because (1) it uses many Scala constructs that might cause performance issues (*e.g.*, implicit conversions, string interpolation), (2) M2T transformation is one of the most often used model manipulation tasks [21], and (3) the implementation in the other languages was straightforward, limiting the possibility of misusing some features. As a concrete transformation we chose our simple OO model to Java transformation, since nearly all the listed languages provide an example that is based on it. Table 2 shows the median of 20 consecutive runs for two different model sizes, (A) corresponds to 250 classes with 50 methods and properties each, while (B) is 500 classes with 100 methods and properties each.

Scenario	Java	SIGMA	EGL	Acceleo	Xtend	Kermeta	Scala
A	1.0	1.0	18.6	11.9	0.9	1.0	0.9
B	1.0	1.8	48.1	16.4	1.0	1.0	1.0

Table 2. Performance of different M2T languages normalized to Java version

As expected, the performance of SIGMA together with the other compiled languages is close to Java, while the interpreted ones are an order of magnitude higher (also their memory footprint is double). The decrease in SIGMA performance in the case of the larger model is caused by the whitespace handling decorator. Every appended string is checked for whitespaces to be removed, and its complexity increases with the indent level. Without this decorator, the performance is again close to Java (A: 0.9, B: 1.0).

7.5 Evaluation of Requirements

In the previous section we evaluated the competitive performance requirement. The following is the evaluation of the other requirements identified in Section 2: (1) *Epsilon-like features and expressiveness*. In Table 3, we evaluate the requirement stating that our DSL should contain similar features as in Epsilon

¹⁰ Kermeta version was put together by Didier Vojtisek, a Kermeta committer. All the source code is available from the companion web page [4].

languages, *i.e.* EOL, EVL, ETL and EGL. The listed Epsilon features are taken from the Epsilon website [6].

- (2) *Usable tool support.* One of the advantages of an internal DSL is that it can directly reuse the tool support provided for the host language. As mentioned above, the recent versions of the Scala IDE [5] provide solid tools facilitating Scala development. Moreover, additional tools operating on the JVM class level such as profilers can be directly used.
- (3) *Simple testability with existing unit frameworks.* The method-based styles of all the three languages allows to cherry-pick the fragments of model manipulation to be tested by any Java-based unit testing framework, which is especially useful for larger model transformations.
- (4) *Simple integration into existing EMF projects.* Executing a model manipulation task in SIGMA is no different from executing a regular JVM-based application and therefore it can be included in many building tools.

With the basic set of Scala skills necessary to use SIGMA, we consider that the DSLs are rather small and thus less learning effort is likely to be required in comparison to language such as OCL or EOL. Finally, being an internal DSL is notably reflected in the code size of the implementation. SIGMA is currently implemented in 3500 lines of Scala code, which is an order of magnitude less than just EOL, which is an order of magnitude less than Eclipse OCL.

7.6 Limitations

Apart from the syntax limitations, an internal DSL is in general a leaky abstraction [42]. For example an implementation of guards and structural constraints can contain arbitrary code and by default. There is no simple way to make sure they are side-effect free without employing an external checker such as IGJ [45], which brings additional overhead. Traditionally, the support for domain-specific analysis, error checking and optimization has been difficult to realize in internal DSLs. However, Scala offers some more advanced methods for DSL embedding, using language virtualization and lightweight modular staging [37,13].

Depending on the target audience, the use of Scala can be seen as a drawback rather than a merit. It is a new language that has not yet reached the popularity of some of the mainstream programming languages. It might be hard to justify learning a language such as Scala solely for the purpose of model manipulation. Finally, there is a small compile-time overhead of generating the common infrastructure.

7.7 Threats to Validity

There are few potential threats to validity of the evaluation presented in Sections 7.3 and 7.4. First, the implemented model manipulation tasks represent only a small subset of possible scenarios. To the best of our knowledge, we do not know about other publicly available larger model manipulations implemented in Epsilon. In some parts, the Epsilon code itself could be improved which, would result in more concise solutions, nevertheless, we believe that this would not make a major difference.

DSL	Feature	Sigma support
EOL	Simultaneously accessing/modifying many models of (potentially) different metamodels	+ Accessing a model in SIGMA is the same as accessing a Scala/Java class
	All the usual programming constructs	+ SIGMA is based on Scala GPL
	First-order logic OCL operations	+ All OCL collection operations are supported
	Create and call methods of Java objects	+ Scala is interoperable with Java
	Dynamically attaching operations to existing meta-classes and types	+ Supported through Scala implicit conversions
	Cached operations	+ Supported using Scala implicit conversions and lazy values
	Extended properties	+ Supported using Scala implicit conversions
	User interaction	+ Supported through Scala and Java libraries
EVL	Create reusable libraries of operations	+ Scala has notions of packages and imports that goes beyond the one in Epsilon
	Distinguish between errors and warnings during validation	+ Both errors and warnings are supported
	Guarded constraints	+ Constraints can have guards (line 5-7 in Listing 1.3)
	Specify constraint dependencies	+ Constraint dependency can be specified in a guard condition (line 6 in Listing 1.3)
	Break down complex constraints to sequences of simpler statements	+ SIGMA constraints can contain arbitrary Scala code (<i>cf.</i> Section 7.6)
ETL	Automated constraint evaluation	+ SIGMA execution semantics is the same as in Epsilon
	Out-of-the-box integration with the EMF validation framework and GMF	+ SIGMA provides an EValidator implementation
	Ability to query/navigate/modify both source and target models	+ Accessing a model in SIGMA is the same as accessing a Scala/Java class
	Declarative rules with imperative bodies	+ Method signature declares a rule and the method body can contain any Scala code
	Automated rule execution	+ SIGMA execution semantics is the same as in Epsilon
	Lazy and greedy rules	+ Both rule types are supported using annotations
EGL	Multiple rule inheritance	0 Currently, inherited rules must be called explicitly
	Guarded rules	+ Rules can have guards (lines 12-14 in Listing 1.3)
	Decouple content from destination	+ The result of M2T transformation is a string that can be outputted to any destination
	Call templates (with parameters) from other templates	+ An M2T template is just a Scala class that can be used from any Scala code
	Define and call sub-templates	+ A sub-template is a Scala method that can be used from any Scala code
	Mix generated with hand-written code	- There are several problems with mixing generated and non-generated code (<i>e.g.</i> , complicated merge, non-generated code is lost among the generated one, generated code has to be put under version control) and therefore SIGMA promotes the generation gap pattern [17] instead

Table 3. Supported Epsilon features. (+) supported, (-) unsupported, (0) partially supported

As for the performance evaluation, it is a form micro benchmark and as such it should be considered with all the validity threats micro benchmarking brings. We have implemented only a small model manipulation task, yet, we already see the trends of the different approaches whose performance will likely remain in the same order of magnitude even for other model manipulations.

8 Related Work

Cuadrado *et al.* [14] developed RubyTL, a Ruby internal DSL for ATL-like M2M transformations. Later, they used it for a comparison on the effort of building an internal DSL and an external one. They concluded that the success of an internal DSL highly depends on the selection of the host language, its support for DSL embedding, execution performance, tool support, and popularity [39]. The main difference with SIGMA is that using a dynamic language prevents any compile type checking. Also RubyTL relies on its own EMF model parser facilities and is not directly interchangeable with the mainstream EMF.

George *et al.* [18] used Scala to build a M2M transformation DSL for the EMF platform that resembles ATL. Since we use the same host language, their DSL is fully interoperable with ours, *e.g.* the common infrastructure (Section 3) can be directly used in the transformation rules. However their internal DSL is not completely type safe and they represent transformation rules directly as anonymous classes, which limits their modularity and reusability. Wider [44] presents an interesting approach to bidirectional model transformations by embedding lenses (a combinator-based approach to bidirectional term-transformations) into Scala and showed how they can be used in an MDE context. Akehurst *et al.* [11] developed a Java library for simple imperative M2M transformations. Being based on Java gives it performance and tool support advantages, as well as a wider audience. On the other hand, there is no particular support for improving the expressiveness of model navigation and modification, resulting in rather verbose and complicated code.

9 Conclusion

In this paper we have presented an alternative internal DSL approach for model manipulation whereby the supporting constructs are embedded into a GPL. We used Scala as the host language to design and fully implement SIGMA, a family of type-safe internal DSLs for EMF model consistency checking, M2M and M2T transformations. We have shown that the resulting DSLs have similar expressiveness and features found in external model manipulation DSLs, while providing competitive performance, compact implementation, and the ability to take advantage of the advanced Scala tool support.

Non-trivial model manipulation tasks often involve a lot of general purpose programming. By using a GPL such as Scala with rich general purpose programming constructs, we were able to significantly reduce the code size of the model manipulation tasks implemented in our evaluation process, without jeopardizing their readability.

Current work in progress around SIGMA consists in carrying out more evaluations to further assess the usability of the proposed DSLs. SIGMA has notably participated in the 2014 edition of the Transformation Tools contest [19]. For the future we first want to apply the Scala advanced DSL embedding techniques to address identified limitations such as the problem of *leaky abstraction*. We also plan to tackle DSL composition issues by exploring appropriate ways to couple SIGMA with other DSLs in different case studies.

Acknowledgments. This work is partially supported by the Datalyse project www.datalyse.fr.

References

1. Apache Velocity, <http://velocity.apache.org/>
2. Epsilon Egldoc, <https://wiki.eclipse.org/EDT:EGLDoc>
3. Epsilon Eugenia, <http://www.eclipse.org/epsilon/doc/eugenia/>
4. Paper Companion Web Page
5. Scala IDE, <http://scala-ide.org/>
6. The Epsilon Project Documentation, <http://eclipse.org/epsilon/doc/>
7. The SALTY Project, <https://salty.unice.fr>
8. The SIGMA Project, <https://github.com/fikovnik/Sigma>
9. The Unicaneer2sql Project, <https://code.google.com/p/unicaneer2sql/>
10. The YourCast Project, <http://yourcast.fr/>
11. Akehurst, D., Bordbar, B., Evans, M., Howells, W., McDonald-Maier, K.: SiTra: Simple Transformations in Java. In: 9th International Conference, MoDELS 2006.
12. Burmako, E.: Scala macros: let our powers combine! In: Proceedings of the 4th Workshop on Scala, 2013
13. Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language virtualization for heterogeneous parallel computing. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2010
14. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A Practical, Extensible Transformation Language. In: Model Driven Architecture – Foundations and Applications, 2006
15. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 2006
16. Dubochet, G.: Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming. Ph.D. thesis, Ecole Polytechnique Fédérale de Lausanne, 2011
17. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 1st edn., 2010
18. George, L., Wider, A., Scheidgen, M.: Type-Safe Model Transformation Languages as Internal DSLs in Scala. In: Proceeding of the 5th International Conference on Theory and Practice of Model Transformations. ICMT, 2012
19. Horn, T., Krause, C., Rose, L.: 7th Transformation Tools Contest , 2014
20. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.M. (ed.) Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science, vol. 3844, 2006
21. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press, 2008
22. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. In: Proceedings of the 2008 International Conference on Model Transformations. 2008
23. Kolovos, D., Paige, R., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture – Foundations and Applications, LNCS 4066, 2006
24. Kolovos, D., Paige, R., Polack, F.: On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In: Abrial, J.R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis, 2009
25. Krikava, F.: Domain-Specific Modeling Language for Self-Adaptive Software System Architectures. Ph.D. thesis, University of Nice Sophia-Antipolis, 2013
26. Krikava, F., Collet, P.: On the Use of an Internal DSL for Enriching EMF Models. In: Proceedings of the Proceedings of the 2012 International Workshop on OCL and Textual Modelling, 2012

27. Krikava, F., Collet, P., France, R.: ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures. In: Symposium on Applied Computing (SAC), track on Dependable and Adaptive Distributed Systems (DADS). 2014
28. Krikava, F., Collet, P., France, R.B.: Manipulating Models Using Internal Domain-Specific Languages. In: Symposium on Applied Computing (SAC), track on Programming Languages (PL). 2014
29. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Comput. Surv.* 37(4), 316–344 (2005)
30. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation, 2012
31. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems, 2005
32. Object Management Group: MOF Model to Text Transformation Language (MOFM2T). Tech. rep., Object Management Group, 2008
33. Object Management Group: MOFTM Query / View / Transformation (QVT). Tech. rep., Object Management Group, 2011
34. Object Management Group: OMG Object Constraint Language (OCL). Tech. rep., Object Management Group, 2012
35. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An Overview of the Scala Programming Language. Tech. rep., École Polytechnique Fédérale de Lausanne, 2014
36. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., a.C. Polack, F.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, 2009
37. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: Proceedings of the ninth International Conference on Generative programming and component engineering, 2010
38. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: The Epsilon Generation Language. In: Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications. pp. 1–16. ECMDA-FA, 2008
39. Sánchez Cuadrado, J., Canovas, J., Garcia Molina, J.: Comparison between internal and external DSLs via RubyTL and Gra2MoL. In: Mernik, M. (ed.) Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global 2012
40. Schmidt, D.C.: Guest Editor’s Introduction: Model-Driven Engineering. *Computer* 39(2), 2006
41. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. *Software, IEEE* 20(5), 20003
42. Siek, J.G.: General purpose languages should be metalanguages. In: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation, 2010
43. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional, 2008
44. Wider, A.: Towards combinators for bidirectional model transformations in scala. *Software Language Engineering*, 2011
45. Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., Ernst, M.D.: Object and reference immutability using java generics. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007