Sigma Documentation

Filip K?ikava and Philippe Collet

August 1, 2012

Contents

1 Introduction

Sigma is a framework that allows enriching EMF models using JVM based languages. In the distribution there is a special support for Scala that leverage its features to write elegant and concise expressions together with an extended support for capturing structural constraints.

Currently, it is in an internal process to open source it and make it available to the community. For more information, please contact Filip Krikava (filip dot krikava at i3s dot unice dot fr)

2 Reference Documentation

2.1 Overview

The main difference between an external and internal DSL is the level of abstraction they can work with. While in the former, appropriate concepts can be freely chosen, the latter must always operate on the concepts found in the host language. In our case however, thanks to the following features, we can seamlessly write similar powerful OCL-like expressions:

1. The EMF generator transforms the model concepts into Java code i.e. model classifiers maps into Java classes, structural and behavioral features into appropriate methods.
2. Scala allows one to omit parenthesis in methods without parameters so that similar OCL-like object navigation expressions can be written: self.getMembership.getParticipant.getDateOfBirth The *noise* generated by successive *get* calls can be removed from these expressions by generating (using the EMF code generator dynamic templates) additional methods without the get prefix that simply delegate their execution to the corresponding getters. This way the above expression becomes the same as the one in OCL: self.membership.participant.dateOfBirth

3. With the large number of collection operations with support of higher-order functions we can get OCL-like collection navigation but in a more uniform way. For example, a selection of customer cards whose transactions are worth more than 10000 points is expressed in OCL as follows:

```
self.cards->select(
  transactions->collect(points)->sum() > 10000)
```

   and in Scala:

```
self.cards filter (
  _.transactions.map(_.points).sum > 10000)
```

   The _ is used to as a placeholder for parameters in the anonymous function instead of specifying a concrete name:

```
self.cards filter (c =>
  c.transactions.map(t => t.points).sum > 10000)
```

Besides, since we manipulate the EMF generated Java code, we have a support for multiple models out of the box as it only means accessing classes from different packages. The Java generics are also supported in Scala.

2.1.1 Basics

The main usage of our DSL is to enrich EMF models with

1. structural constraints,
2. derived features definition,
3. operations bodies implementation.

Each of these constructs is represented as a Scala object method with an appropriate signature. inline]ref to EMF integration The first parameter of these methods is always the surrounding context representing the contextual instance (like self in OCL). The other parameters and return type depends on the concrete construct:

- *Derived property* The return type is the type of the property itself. The following function defines a code that will be executed by EMF when a derived property printedName, defined in a Customer class, is accessed:

```
def getPrintedName (self: Customer): String =
  self.owner.title + " " + self.owner.name
```

- *Operation body* Additional parameters and the return type represent the operation parameters and its type. Following the same pattern, the function below defines the code for the getTransaction operation from the CustomerCard class, which has two parameters of type Date and returns a set of Transaction references:

```
def invokeGetTransactions(self: CustomerCard,
  until: Date, from: Date): Set[Transaction] =
  self.transactions filter (
    t => t.date.isAfter(from) &&
      t.date.isBefore(until) )
```

The invariant method signature is discussed in section˜Structural Invariants (§2.1.2).

Extensibility

Beside all the functionality that is brought by its standard library, the Scala language counters the limited expressiveness of OCL by leveraging the extensive amount of existing Java libraries. To use any of them is only a matter of adding a new dependency to the project.

The real Scala extensibility, however, lies in the ability to extend existing types, statically and in a type safe way. For example, in Scala there is no logical operator equivalence to OCL implies. Of course we could simply define a function that takes two boolean expressions and returns their logical implication, but this would feel very unnatural to use. With Scala, we can define this function to be a method on an existing boolean type by using the Pimp my Library approach:

```
class ExtendedBoolean(a: Boolean) {
  def implies(b: => Boolean) = !a || b
}
// add an implicit conversion between the types
implicit def extendedBoolean(a: Boolean) =
  new ExtendBoolean(a)
```

With the above definitions imported one can now use the new method directly and it feels like being part of the language:

```
a = true; b = false; c = a implies b
```

Using the same pattern we can create other missing OCL operations like closure, but also create completely new constructs.

Reusability

Scala allows both imperative and functional language constructs. This allows one to break the complex and long expressions into smaller pieces and store the intermediate values into local variables in order to improve the overall readability. The reusability of expressions can be easily achieved by simply organizing these expressions into object methods and libraries that can be shared across models and projects.

We can push the reusability even further as Scala also supports *structural typing*. Thanks to this feature one can write a very generic expression that can be applied across different and completely unrelated models. For example:

```
def validateNonEmptyName(self:
  {def name: String}) = !self.name.isEmpty
```

represents a generic invariant checking that an attribute name is non empty. It can be applied to any class regardless of its type.

Handling Undefined and Invalid Values

When evaluated, some expressions in OCL can result into invalid or undefined values such as when an empty collection is traversed or an unset reference is navigated. Since neglecting them will lead to null pointer exceptions, we also need to handle these cases in Scala.

In order to simplify the code we make use of the Scala Option class. As the name suggests, it is just a simple abstract container that wraps around an instance of some type T which represents an optional value. The two possible instances are Some and None denoting whether there is an actual value for T.

For instance, in the previous example, we checked if a name attribute is non empty string. However, if the multiplicity of this attribute had been defined as 0..1, in the cases where the name had not been set the code will throw a null pointer exception. The EMF code generator does not make any difference between 0..1 and 1..1 and outputs the same getter signature:

```
public String getName();
```

% However, we can simply extend the EMF code generator dynamic templates and implicitly generate Option return type:

```
public scala.Option<String> name() {
    scala.Option.apply(this.getName());
}
```

Because we use a different name for getters (without the get) the resulting class is still compatible with the rest of the EMF world.

As the Scala documentation suggests the most idiomatic way to use an Option instance is to treat it as a collection or monad, which results in a very concise and null pointer safe implementation:

```
self.name.filter(!_.isEmpty).getOrElse(false)
```

Type Casts

Another often used Scala construct is type *pattern matching*. It helps us in simplifying type casts in OCL, which are often used when constraining metamodels. Instead of an expression like:

```
if self.oclIsKindOf(Customer) then
  self.oclAsType(Customer).someAction()
else
  // something else
endif
```

one can simply write:

```
self match {
  case c: Customer => c.someAction()
  case _ => // something else
}
```

2.1.2 Structural Invariants

The improved support of invariant constructs is addressed by flexible return types of the invariant functions. We need to be able not only to specify whether an invariant holds

on a certain object, but in case it does not, we should say why, how severe the problem is and also be able to provide a support for automatic repair of such inconsistencies (where applicable).

Therefore a function representing an invariant can return either:

1. A simple boolean representing whether an invariant holds on self.

```
def validateOfAge(self: Customer) = self.age >= 18
```

2. A string representing an error message in case it does not.

```
def validateOfAge(self: Customer): Option[String]=
  self.age >= 18 match {
    case true => None
    case false => Some("The person %s is under age"
      format self.printedName)
  }
```

   In this case we use the Option construct to avoid using null as a valid return value.

3. An object encapsulating the additional details.

```
def validateDefinesGetInstance(self: UMLClazz) = {
  self.features
    .find(_.name == "getInstance") match {
      case Some(_) => Success
      case None => Error("Missing getInstance",
        QuickFix("Add a getInstance operation", {
          clazz: UMLClazz =>
            clazz.features += create[UMLFeature] {
              op => op.setName("getInsatnce")
              // ...
            }
        }))
    }
}
```

Context definition and invariant dependency are solved by annotations that are processed by the EMF custom validator.

```
@satisfies("DefineGetInstance")
```

```
def validateGetInstanceIsStatic(self: UMLClazz)
```

The validation functions are not called directly but via a proxy that ensures each invariant is called for a specific instance at most once. In order to be able to implement all the above we also need to extend both the user interface and the runtime part of the EMF validator.

Since referencing the dependent invariants as strings is not very practical, we are currently looking into how the upcoming Scala 2.10 macros can help us in building a type safe alternative to the annotations.

2.1.3 Drawbacks

Obviously there are also some shortcomings in our approach.

First, since the implementation of an invariant or a derived property can contain arbitrary code, by default there is no way to make sure they are side-effect free. One way to verify this would be by using external checker such as IGJ. These checkers work as Java language extensions and can verify that an object may not be mutated through a read only reference (a reference annotated by @ReadOnly annotation).

Second, another problem is in the loss of formal reasoning and analysis. Nevertheless the analysis part related to performance can be solved using a regular profiler.

Finally, in the DSL, we do not support some of the constructs related to postconditions.

2.1.4 Why Scala?

While there are other languages that could be used to build an internal DSL, we find Scala a particularly good fit for our purposes. It is a modern general purpose language that runs on the top of a JVM, and was designed from the start to be an extensible language for building internal DSLs. It combines both object-oriented and functional style of programming with static typing that uses type inference to provide type safety without adding unnecessary syntactic clutter. It is also well supported by the major tool vendors and has already established an active community.

List of External Links

http://www.scala-lang.org/
http://www.scala-lang.org/api/current/scala/Option.html
http://www.artima.com/weblogs/viewpost.jsp?thread=179766
http://www.scala-lang.org/node/1403
http://types.cs.washington.edu/checker-framework/current/checkers-manual.html