On the Use of an Internal DSL for Enriching EMF Models

Filip Křikava, Philippe Collet

On the Use of an Internal DSL for Enriching EMF Models

Filip Křikava
Université Nice
Sophia Antipolis, France
I3S - CNRS UMR 7271
filip.krikava@i3s.unice.fr

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

ABSTRACT

The Object Constraint Language (OCL) is widely used to enrich modeling languages with structural constraints, side effect free query operations implementation and contracts. OCL was designed to be small and compact language with appealing short "to-the-point" expressions. When trying to apply it to larger EMF models some shortcomings appear in the language expressions, the invariant constructs as well as in the supporting tools.

In this paper we argue that some of these shortcomings are mainly related to the scalability of the OCL language and its trade-offs between domain-specificity and general-purpose. We present an alternative approach based on an internal DSL in Scala. By using this modern multi-paradigm programing language we can realize an internal DSL with similar features found in OCL while taking full advantage of the host language including state-of-the-art tool support. In particular, we discuss the mapping between the OCL and Scala concepts together with some additional constructs for better scalability in both expressiveness and reusability of the expressions.

Categories and Subject Descriptors

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

1. INTRODUCTION

OCL is used to complement the limited expressiveness of the structural constraints of the modeling languages like UML (Unified Modeling Language) or EMF (Eclipse Modeling Framework). Such model *constraints* are captured as state invariants using a side-effect free expression language that supports first order predicate logic with model querying and navigation facilities [18, 15]. Moreover, these expressions can be further used to include additional information to the model such as operation contracts in form of pre and post conditions, and implementation of derived features and operation bodies. OCL is also embedded in context of other tools such as the Object Management Group QVT model transformation.

OCL is an appealing and expressive language, but when applied to larger EMF models using Eclipse OCL[1], we found a number of shortcomings in the language expressions, the invariant constructs as well as in the supporting tools. While some of these problems are already well identified in the literature either as research agenda or accompanied with some solutions (c.f. Section 2), the lack of an overall integration eventually led us to look for alternatives. According to us, some of these shortcomings are in general related to some scalability issues as a result of trade-offs between domain-specificity and general-purpose.

In this paper we present an alternative approach based on an internal DSL in Scala[2], a statically typed object-oriented and functional programming language that runs on top of a Java Virtual Machine (JVM). Besides the seamless integration with EMF, some Scala features, such as support for higher-order functions, rich collection libraries and implicit type conversions, allow us to write very similar OCL-like expressions, but also leverage from the many libraries found in the Java and Scala ecosystems. Besides Scala also comes with state-of-the-art tool support.

The rest of the paper is organized as follows. In Section 2 we describe the main shortcomings of OCL based on our experience. Section 3 shows how we use Scala as an alternative approach to enrich EMF models. It is followed by Section 4 where the implementation details are presented together with an evaluation. In Section 5, we discuss related work. We briefly conclude and outline future work in Section 6.

2. SOME SHORTCOMINGS OF OCL

In this section we present a list of shortcomings we came across while using OCL in an EMF based MDE toolchain, but various usages of OCL reveal different pros and cons. In our study we were concerned with the practical side of OCL rather than a formal one like in [14] or [4].

For each of the point we also report on works in which similar problems were reported. Despite the fact that many of these issues have already been identified or addressed, the lack of an overall integration is a crucial issue, which, according to us, influences the slow adoption of OCL in the industry.

2.1 OCL Expressions

One of the key points that Anders Ivner mentions in the foreword to the second edition of *The Object Constraint Language* [18] is "Second, it is compact, yet powerful. You can write short and to-the-point expressions that do a lot". While this is true for many of the short and straight-forward expressions, when the complexity grows our ease of reading and writing of these expressions decreases radically. This might be especially hard for the new users when they move from the tutorial like expressions to real world ones.

Complex expressions are hard to write and maintain
OCL constraints and queries are defined in form of expressions that can be chained together. The underlying linearity of this chaining

[1] `http://goo.gl/TECuz`

[2] `http://www.scala-lang.org/`

OCL'12 September 30 2012, Innsbruck, Austria
ACM 978-1-4503-1799-3/12/09.

often leads to long and complex expressions that are difficult to understand and maintain. Since the debugging support in OCL is rather limited, mostly only simple logging or tracing is possible, maintaining of these expressions is particularly hard.

In [10] Correa et al. provide some refactoring techniques to simplify some of these complex expressions. Ackermann et al. propose in [1] to utilize specification patterns for which OCL constraints can be generated automatically, together with a collection of OCL specification patterns formally defined. These techniques might help in certain cases, but in general there is no simple solution to break the linearity of the expressions. Similar observations regarding the complexity are also reported in [17].

Limited extensibility and reuse
The operations available in the OCL Standard Library are limited and thus might not be sufficient to express certain needs such as a string regular expression matching [6]. Adding new operations is usually a difficult process[3] and is also tool specific. In [7] authors position a promising approach to define a set of useful and easily reusable OCL expressions that can be packaged in libraries. Another approach based on a modular redefinition of OCL enabling consistent extension and customization is discussed in [3].

Side-effect free expressions
By design OCL expressions have no side-effects, there are no assignment semantics, no possibility of creating new instances directly and all data types are immutable. This is particularly useful for capturing constraints, but at the same time, it makes the state changing operations impossible to be implemented in pure OCL. In the case of EMF this means that a user always have to fall back into Java-platform based languages to implement the state changing operations. There are some OCL extensions proposed to introduce imperative constructs into OCL like the *ImperativeOCL* within the QVT, *SOIL* [5] or EOL [12], but they are not directly applicable in the Eclipse OCL. However, an OCL environment might provide some mutation capability in an appropriately disciplined fashion.

Inconsistency/Confusing issues
There are a number of little inconsistencies and confusions that have already been reported [19, 20, 6] and that we encountered during our development, such as: different symbols to navigate through sets (`->`) and scalars (`.`); implicit `oclAsSet` when applied onto `null` object; implicit collection flattening when using `collect` operation; logical operators with `undefined` values; differences between UML constructs and corresponding OCL capabilities that have no correspondents in EMF (Ecore) and vice-versa like `oclContainer()` resulting in usage of different OCL meta-models. Also, while OCL has supported generic collections before Java, it does not allow to work with generic class type parameters.

2.2 Constraints Capturing Constructs

Structural constraints extend the specification of models with state invariants. There are numerous problems with these invariants constructs offered by OCL. In [13], Kolovos et al. gives a detailed overview of these issues, namely: support for user feedback; for warning critiques; support for dependent constraints; flexibility in context definition; support for repairing inconsistencies. To their list we would only add **invariant reuse** across different contexts and different models, so they do not have to be copy and paste.

In the same paper the authors also propose an extension that addresses the identified shortcomings together with an implemented in prototype that is part of the Epsilon[4] family. However, their solution, EVL (Epsilon Validation Language), is based on yet another external DSL - EOL (Epsilon Object Language) which tries to tame some of the OCL problems, but with limited tooling support.

2.3 Tool Support

There is a close relationship between a language and its supporting tools. Often a language choice is influenced by the accessibility of appropriate tools that facilitates the language use. Even though implementing a solid and feature rich OCL support is a difficult and resource consuming task, we currently see an impressive number of OCL tools developed within the academia [8]. Unfortunately, this trend does not seem to be followed in the industry where there is a significant lack of commercial tools [9].

In our development, based on EMF, we have chosen the Eclipe OCL project as it enables a tight integration of OCL expressions using EMF delegates and has an active community. We worked mainly with the OCLINECORE editor which embeds the OCL expressions directly into EMF models by annotating the relevant elements, which makes it very convenient to work with.

Similarly to many other software applications when they are used at large scale, one of the main experienced difficulties was related to **scalability and stability** when the models started to grow. After reaching 800 lines the responsiveness of the editor became a serious problem[5]. Of course this issue is likely be reduced in future versions.

Another problem is related to **developer feedback** and **insufficient debugging capability**. The former is associated with propagation of runtime exceptions (`null` values), which may be difficult to trace. The later one makes it difficult to narrow bugs in incorrect OCL expressions. While the interactive console can help with the simpler cases, for some complex OCL expressions that include evaluation of derived features and operations that are also implemented in OCL debugging is really needed. Moreover OCL is either directly interpreted or compiled into some other language like Java, therefore the connection to the original OCL expression is lost during debugging. In case of Eclipse OCL this might go away in further versions since new editors are based on Xtext[6] that now includes Java's debugging support for other languages (JSR-045).

3. INTERNAL DSL APPROACH

From our point of view, the shortcomings identified in the previous section are mostly related to scalability in the OCL language itself, as well as in the performance, stability and features offered by the OCL tools. The problems with OCL starts when used in the large, having many invariants and complex expressions that go beyond simple object navigation, etc.

Similar issues can be found in the alternatives that are also based on external DSL like the EOL/EVL. The underlying problem is that the creation of an external DSL is an challenging task from both the language design and the tool implementation perspectives. On the other hand the main benefit of using DSLs like OCL is in raising the level of abstraction, which allows one to express the domain specific concerns more clearly. For example, we could simple use Java with its powerful ecosystem to implement constraints checking. But with the lack of first-order logic collective operations, as in OCL, the resulting code would be far from clean and concise and the expressed concern would be lost among Java commands.

[3]Example of this process in Eclipse OCL `http://goo.gl/fWzBn`

[4]`http://www.eclipse.org/epsilon/`

[5]Tested with Eclipse 3.7, OCL 3.1 on MacBook Pro 2.53 Ghz Intel i5, 8GB RAM

[6]`http://www.eclipse.org/Xtext/`

We advocate that with a general purpose language that delivers both powerful and flexible constructs together with state-of-the-art tool support, one would be able to make similar expressions, like in OCL, with all the benefits of this host language. We thus propose a pragmatic approach based on an internal DSL i.e., a language that is represented within the syntax of a general-purpose language with a stylized usage of that language for a domain-specific purpose [11]. Concretely, relying on a modern multi-paradigm language, in our case Scala, we define an internal DSL that allows us to enrich EMF models in the similar fashion as OCL, but without the shortcomings identified in the previous section.

The examples used in this section comes from the well-known *Royal and Loyal* system presented in [18].

3.1 Principles

The main difference between an external and internal DSL is the level of abstraction they can work with [11]. 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.

3.1.1 Basics

The main usage of our internal 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 (the actual integration with EMF is shown in section 4.2). 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 3.2.

3.1.2 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*[7] 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.

3.1.3 Reusability

Scala allows both imperative and functional language constructs. This allows one to break the complex and long expressions into

[7] `http://goo.gl/MfkxZ`

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.

3.1.4 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 class `Option[T]`. 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[8] 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)
```

3.1.5 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
}
```

[8] `http://goo.gl/vNuB`

3.2 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[T]` 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.

3.3 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 ([21]). 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.

3.4 Why Scala?

While there are other languages such as Xtend[9] 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[10]. 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.

4. IMPLEMENTATION

The internal DSL presented in the previous section has been implemented on the top of a framework called Sigma[11].

4.1 Overview of the Sigma Framework

The Sigma framework offers an API that allows one to enrich EMF models using Java class methods with appropriate signature. It is a small layer written in Java that acts on the top of the EMF API notably the EValidator and EMF delegates (see next section). It allows to delegate computation of model features such as derived properties, operation bodies and constraints to Java class methods and it uses EMF model annotations to store information about where to find these methods. The very similar concept is used by the Eclipse OCL with the difference that it also uses the annotation to store the actual code. It is a general framework that can supports any Java-platform language. We provide a default support for Java and Scala.

4.2 Integration with EMF

The integration between Sigma Framework and EMF is realized using the EMF delegation.

4.2.1 Integration Through Delegation

From Eclipse Helios milestone 4, EMF supports a delegate computation of structural invariants, derived features and operation. This was the initial step to make it possible to define and compute model extensions in languages other than Java. We build on this feature and provide a set of delegates for handling the execution delegation of the supported model extensions.

Following is an example of steps (illustrated in figure 1) to be done in order to be able to define a structural invariant using our DSL: Defining a structural invariant using our DSL can be done through the following steps: 1. register delegation to sigma support by attaching an annotation on the target model package (only once), 2. attach the Sigma annotation to the class one want to add the invariant and set the annotation `delegate` detail to reference Scala object that will serve as the delegate for this model class, 3. attach the Ecore annotation to same class and set the `constraints` detail, 4. provide the actual implementation of the constraint in the appropriate method of the Scala object. Similar steps are required to be performed using the Eclipse OCL, although for example the OCLINECORE can take care of them automatically.

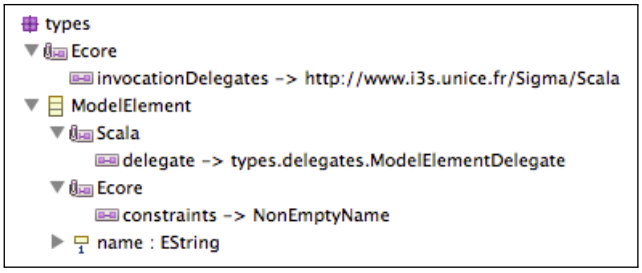

Figure 1: Sigma delegation represented in EMF

4.2.2 Runtime Overhead

There is some extra overhead caused by using the EMF delegation. A part of this overhead is related to querying the registry in order to find which delegate should be used. Another and more significant one is due to the reflection used to find and execute the target method. To overcome both issues we are currently working on extending the dynamic templates to generate direct calls to the appropriate target methods instead of going through the delegation layer.

4.3 Tool Support

For writing the actual expressions in the internal DSL we can immediately leverage the tooling support that comes with the host language. On the other hand, there is no support for maintaining the synchronization between the model annotations and their delegating method counterparts. Doing this manually can become very tedious and error-prone, especially in large models. In the current version of Sigma, we address this issue by providing the following tools that given an EMF model: 1. generate the appropriate delegate method signatures, 2. attach the appropriate Sigma annotation to the various model elements. More advanced tooling in form of Eclipse plugins is currently in progress.

5. RELATED WORK

There has been a lot of work addressing different OCL shortcomings (cf. section 2). Since OCL has become popular in the research community, there is also a lot of work that reflects on the formal aspects of the language, its syntax, semantics and expressiveness comparing to relational calculus [14, 4, 19, 3]. Assessing the OCL tool support has also been investigated [8, 9]. In [16] authors evaluates expressiveness of OCL in the context of their model assessment framework. However, their evaluation is based rather on the type of queries and checks they can express using OCL and do not discuss using OCL in large scale model driven developments.

In [7] the authors discuss another challenge in pragmatic OCL development. Their solution addresses both the conceptual and implementation level challenges proposing new OCL extensions and tools. We believe that our approach is more pragmatic, at least until all the promising extensions and production level tools are implemented. Our solution leverages the growing community around Scala and needs only a tiny API layer where all constructs are in forms of libraries, not language extensions.

Another alternative is the Epsilon project with the EOL. [12]. While it comes with an Eclipse based environment and a solid documentation, it has positioned itself not to be a rival to the OMG standards, but rather a prototype in which one can easily experiment with novel approaches in MDE [13].

[9] `http://www.eclipse.org/xtend/`

[10] `http://goo.gl/7lhvy`

[11] More information about this framework can be found on the project web site `http://nyx.unice.fr/projects/sigma/`

In [2], Akehurst et al. question whether the modern programming languages do not make OCL redundant since they offer similar expressibility. They provide a concrete examples of writing OCL-like expressions in C#. While we have the same motivating idea of using a general purpose programming language, beside the language choice, we go beyond only enabling OCL-like expressions and we also offer more advanced constructs and improved invariant support.

The project JS4EMF[12] aims at providing Javascript for scripting EMF and at the same time using EMF objects in Javascript code. This includes also support for implementing same modeling extensions using the EMF delegates mechanism. The main difference is that we represent these model extensions as functions so that at the code level they are first class citizens. In JS4EMF the model extensions are simply snippets of Javascript code attached to model elements in the similar fashion as in Eclipse OCL. Therefore we find the similar shortcomings with regards to constraint constructs and tool support.

The XCore project[13] extends concrete syntax for Ecore that, in combination with Xbase expression language, transforms it into a fully-fledged programming language - external DSL. Currently it does not support definition of constraints and can be seen as a work in progress.

6. CONCLUSION

In this paper we have presented some shortcomings of the OCL language that makes it difficult to use in larger EMF based applications. Addressing these issues we have proposed an internal DSL based on Scala and have provided an overview of some of the resulting benefits and drawbacks. Current work in progress consist in improving tool support and EMF integration to provide a solid set of tools to the community. Next, we need to more precisely specify the deviations from OCL and its consequences (the different semantics between OCL and Scala, unlimited numerals, etc.) together with the trade-offs of standardization of OCL versus flexibility and extensibility of the DSL approach. Finally, carrying out more case studies should help us further assess the practicality of the proposed solution and explore what other advantages of a DSL approach, beyond the traditional usage of OCL, may benefit users. This work is a first step into a study how the Scala features could be leveraged in Model-Driven Engineering in conjunction with EMF.

Acknowledgments

The authors would like to thank R. B. France, E. D. Willink and the reviewers for valuable discussions and comments. The work reported in this paper is partly funded by the ANR SALTY project under contract ANR-09-SEGI-012.

7. REFERENCES

[1] J. Ackermann and K. Turowski. A Library of OCL Specification Patterns for Behavioral Specification of Software Components. In *Advanced Information Systems Engineering*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006.

[2] D. H. Akehurst, W. G. J. Howells, M. Scheidgen, and K. D. McDonald-Maier. C# 3.0 makes OCL redundant! *ECEASST*, 9, 2008.

[3] D. H. Akehurst, S. Zschaler, and W. G. J. Howells. OCL: Modularising the Language. In *Proceeding of the Ocl4All: Modelling Systems with OCL*, OCL, page 20, 2007.

[4] A. D. Brucker and B. Wolff. Semantic Issues of OCL : Past, Present, and Future. In *Proceedings of the 6th OCL Workshop*. ECEASST, 2006.

[5] F. Büttner and M. Gogolla. Modular Embedding of the Object Constraint Language into a Programming Language. In *Proceedings of the 14th Brazilian conference on Formal Methods*, 2011.

[6] J. Cabot and M. Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *12th International School on Formal Methods (SFM)*, 2012.

[7] J. Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In *Model Driven Engineering Languages and Systems*. Springer Berlin / Heidelberg, 2009.

[8] J. Chimiak-Opoka, B. Demuth, A. Awenius, D. Chiorean, S. Gabel, L. Hamann, and E. D. Willink. OCL Tools Report based on the IDE4OCL Feature Model. *ECEASST*, 44, 2011.

[9] D. Chiorean, V. Petrascu, and D. Petrascu. How My Favorite Tool Supporting OCL Must Look Like. In *Proceedings of the 8th International Workshop on OCL Concepts and Tools*, volume 15, page 17, 2008.

[10] A. Correa, C. Werner, and M. Barros. Refactoring to improve the understandability of specifications written in object constraint language. *Software, IET*, 3(2):69–90, 2009.

[11] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[12] D. Kolovos, R. Paige, and F. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture – Foundations and Applications*. Springer Berlin / Heidelberg, 2006.

[13] D. Kolovos, R. Paige, and F. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis*. Springer Berlin / Heidelberg, 2009.

[14] L. Mandel and M. Cengarle. On the Expressive Power of OCL. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods*, volume 1708. Springer, 1999.

[15] Object Management Group. OMG Object Constraint Language (OCL). Technical report, Object Management Group, 2012.

[16] J. C. Opoka and C. Lenz. Use of OCL in a Model Assessment Framework: An experience report An experience report. In *Proceedings of the Sixth OCL Workshop*, volume 5 of *OCLApps*, page 17, 2006.

[17] M. Siikarla, J. Peltonen, and P. Selonen. Combining OCL and Programming Languages for UML Model Processing. *Electronic Notes in Theoretical Computer Science*, 2004.

[18] J. Warmer and A. Kleppe. *The Object Constraint Language, Second Edition*. Addison-Wesley, 2003.

[19] C. Wilke and B. Demuth. UML is still inconsistent ! How to improve OCL Constraints in the UML 2.3 Superstructure. In *Workshop on OCL and Textual Modelling*, 2010.

[20] E. Willink. Modeling the OCL Standard Library. In *Workshop on OCL and Textual Modelling*, 2011.

[21] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[12] `http://www.eclipse.org/js4emf/`
[13] `http://wiki.eclipse.org/Xcore`