Towards Using OCL for Instance-Level Queries in Domain Specific Languages

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
{dkolovos,paige,fiona}@cs.york.ac.uk

Abstract. The Object Constraint Language (OCL) provides a set of powerful facilities for navigating and querying models in the MOF metamodelling architecture. Currently, OCL queries can be expressed only in the context of MOF metamodels and UML models. This adds an additional burden to the development and use of Domain Specific Languages, which can also benefit from an instance-level querying mechanism. In an effort to address this issue, we report on ongoing work on defining a rigorous approach for aligning the OCL querying and navigation facilities, with arbitrary Domain Specific Languages to support instance-level queries. We present a case-study that demonstrates the usefulness and practicality of this approach.

1 Introduction

The MOF metamodelling architecture is a four-level integrated architecture for defining, persisting and managing modelling languages and models. At its meta-meta-model level (M3), lies the Meta Object Facility (MOF) [13], a self-defined language for building modelling languages (metamodels). At the metamodel-level (M2) exist languages defined using MOF. The most prominent example of an M2 metamodel is the Unified Modeling Language (UML) [16]. Models expressed in M2-languages are considered to belong to the model-level (M1) while instances of M1 models are placed at the instance-level (M0) (or system-level according to [11]).

The Object Constraint Language (OCL) [15] is a language originally developed to support capturing constraints in models of the MOF metamodelling architecture. However, due to its expressive and efficient model navigation and querying facilities, OCL has also been used extensively as a query language both for expressing stand-alone queries [5], and in the context of model management languages for tasks such as model transformation (e.g. QVT [14], ATL [9], YATL [17]), code generation (e.g. MOFScript [2]) and model merging (e.g. EML [6]). The navigation and querying facilities of OCL operate at two levels: at the metamodel-level (M2), it can be used to define queries in the context of the abstract syntax of a modelling language. Metamodel-level queries can then be evaluated on M1 models. Similarly, at the model-level (M1), it can be used to define queries in terms of model-specific constructs that can then be evaluated on M0 instances.

OCL is currently aligned with MOF and UML. Due to the MOF-OCL alignment, OCL queries can be expressed at the metamodel level and evaluated at the model-level for all MOF-based languages. By contrast, instance-level queries are supported only for UML models, since OCL is not aligned with any other MOF-based languages. The reason for this is the absence, to our knowledge, of appropriate techniques in the literature and the tool-market, for aligning OCL with arbitrary DSLs to support instance-level queries. As a result, in practice, alignment needs to be implemented manually for each DSL within the context of a specific OCL execution engine. This is certainly not a trivial task, as it requires significant expertise with the internals of the engine. Moreover, even if the alignment is successfully implemented for a specific engine, the alignment specification will be highly coupled with the architecture and platform of the engine and thus hard to port or reuse in a different context. In our view, the absence of a generic high-level technique for using OCL to express instance-level queries in DSL models limits the expressive power of DSLs and consequently their usefulness as viable alternatives to UML in a practical software development environment.

To address this issue, in this paper we introduce a generic technique for aligning the OCL navigational and querying facilities with arbitrary modelling languages to support instance-level queries. The remainder of the paper is organized as follows. In Section 2 we discuss the problem of aligning OCL with arbitrary DSLs in detail and identify the key-challenges. In Section 3 we introduce our technique and discuss its rationale as well as the architecture of the infrastructure that allows us to realize it in practice. In Section 4 we provide a case study that demonstrates a working example of aligning a DSL with OCL. Finally, in Section 5 we conclude and discuss interesting issues for further research.

2 Background and Motivation

The principal difficulty in aligning OCL with arbitrary DSLs lies in the two different instantiation mechanisms used in the context of the MOF architecture, as also discussed in [10]. To illustrate this problem we discuss the two different instantiation mechanisms involved in UML 1.5. As illustrated in Figure 1, an object (e.g. $:Customer$) in a UML model is an instance of the $Object$ metaclass defined in the UML metamodel. Similarly, a class (e.g. $Customer$) is an instance of the $Class$ metaclass. Moreover, although both instances are contained in the same (M1) model, the $:Customer$ object is conceptually an instance of $Customer$ class. By convention, instances produced with that *implicit* instantiation mechanism belong to the M0 level but from a strict technical perspective, both Objects and Classes are M1 instances (instances of metaclasses defined in the M2 level). While the $M2 \rightarrow M1$ instantiation mechanism is well-defined in the MOF specification [13], there is no consensus on the semantics of the $M1 \rightarrow M0$ mechanism[12].

The presence of a loosely-defined $M1 \rightarrow M0$ instantiation mechanism renders alignment of OCL with custom DSLs to support instance-level queries particularly challenging. The reason is that an OCL engine needs to be aware of the instantiation mechanism to support built-in OCL features such as $allInstances$, $oclIsTypeOf()$ and $oclIsKindOf()$. A work-around for this problem is to use OCL expressions at the M2 level (where the instantiation mechanism is well-defined) to query M0 instances like any other M1 model elements. In this way, if we wanted to query all adult customers

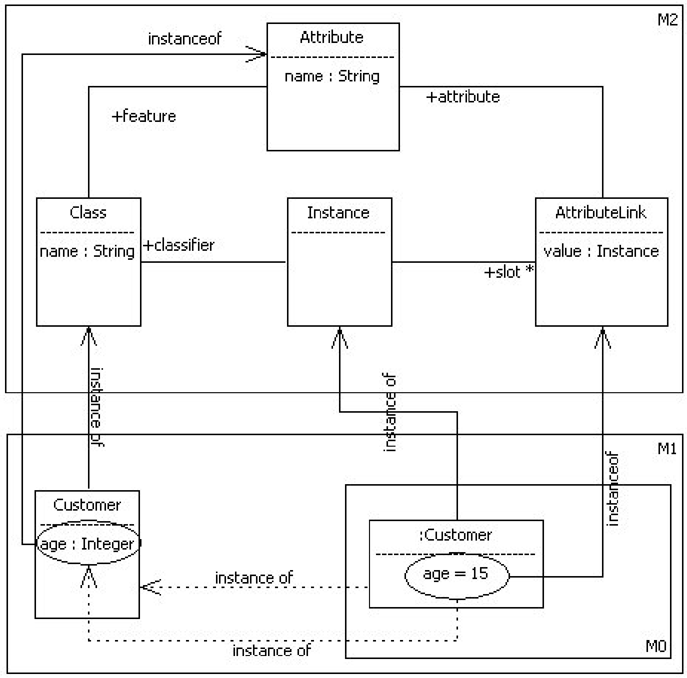

Fig. 1. Demonstration of explicit and implicit instantiation relationships in the MOF architecture

in our UML model of Figure 1, we would have to write the OCL query displayed in Listing 1.1 (or a similar one). The complexity of the OCL expression needed for such a simple query illustrates that while this approach makes querying models at the instance-level feasible, it does not scale for complex queries. By contrast, an OCL engine that is aware of the UML $M1 \rightarrow M0$ instantiation mechanism allows us to specify the same query in a much more compact and meaningful manner, as displayed in Listing 1.2.

Listing 1.1. Querying an M1-level UML model with M2-level OCL

```
Object.allInstances->
  select(o :Object | o.classifier.name->includes('Customer'))->
  select(o :Object | o.slot->exists(aL :AttributeLink |
  aL.attribute.name = 'age' and aL.value.toInteger() > 18))
```

Listing 1.2. Quering an M1-level UML model with M1-level OCL

```
Customer.allInstances->select(c:Customer|c.age > 18)
```

Apart from the $M1 \rightarrow M0$ instantiation mechanism, a UML-OCL execution engine needs to be aware of the semantics of the *point* (.) navigational operator to calculate the result of expressions such as the $c.age$ in Listing 1.2. The semantics of the point operator consist of three parts; the navigation path that must be followed (in terms of M2), the multiplicity of the returned value (single value or collection) and the type of the returned value (Integer, String, Boolean, some other user-defined type etc). Consider the M2-level query in Listing 1.1. The navigation path is defined in lines 1-4 (Object -¿ slot -¿value). The multiplicity is defined by accessing a single-valued feature (aL.value). This indicates that the result should be a single value rather than a collection. The return type is defined via explicit cast of the value of the slot to an Integer. This is done via the built-in toInteger() operation in line 4.

In summary, in order for an OCL engine to support instance-level queries for a new DSL, it must be aware of at least: the semantics of the $M1 \rightarrow M0$ instantiation mechanism and the semantics of the point navigation mechanism for the instance-level. Currently these semantics can be specified using the programming language in which the OCL engine is implemented (e.g. Java) and this is how UML-aware OCL engines, such as $[3, 4, 19, 1]$, have been implemented so far. However, as discussed in [7], 3rd generation languages (3GL) are not particularly efficient for model navigation. Moreover, by adopting this approach, the specification of the semantics becomes bound to the proprietary architecture and platform of the OCL engine. Finally, from a technical perspective, modifying an OCL engine to support a new DSL is by far not a trivial task and this is partly justified by the fact that there is no published work, to our knowledge, on aligning an OCL engine with languages other than UML and MOF.

To address this issue in the following section we propose a generic and platform independent mechanism for specifying the required semantics: OCL itself.

3 Proposed Approach

In this section we demonstrate how we can specify the semantics of the $M1 \rightarrow M0$ instantiation mechanism and the instance-level point operators using OCL itself as the specification mechanism.

For practical reasons, in this work instead of using pure OCL we are using the Epsilon Object Language (EOL) [7], an OCL-based model management language. The reason we use EOL and not pure OCL is that from our experiments, we have found that the OCL expressions needed to specify the semantics of the instantiation mechanism and the point operator tend to be rather complex and consequently difficult to test and debug when expressed in pure OCL. This is because OCL does not support statement sequencing and therefore expressing complex queries requires deep nesting of expressions (including *if-else* expressions and variable declarations using *let* expressions) in a single statement. Instead, in EOL, complex expressions can be decomposed into sequences of simpler expressions that are both easier to read, understand and debug. However, we should stress that in principle, exactly the same functionality can be implemented in pure OCL.

3.1 Relationship between EOL and OCL

EOL supports almost all the navigational and querying facilities of OCL. However, it also supports additional features and also deviates from OCL in some aspects. Therefore, in this section we provide a brief discussion of the additional or deviant features we are using in the EOL listings that follow, for readers that are already familiar with

OCL. For a detailed discussion on EOL and its differences with OCL, readers can refer to [7].

Statement sequencing: In OCL, there is no notion of statement sequencing and, as already discussed, this can lead to extremely complex expressions that are difficult to read and debug. By contrast, in EOL statements can be separated using the semi-column (;) delimiter (similarly to Java,C++ and C#). In our view, this feature greatly enhances readability and renders it easier to debug a fragment of code.

Variable definition: The latest version of OCL (2.0) provides the let expression for creating temporary variables in the context of a single query. Similarly, EOL supports the def statement for defining variables in the context of a block of statements.

Helpers: OCL supports definition of custom operations (*helpers* according to the OCL specification) on meta-classes. Since OCL does not support statement sequencing, the body of an OCL helper is a single OCL expression. By contrast, in EOL, the body of a helper operation is a sequence of statements and values are returned using the $return$ statement.

Style: In EOL, the Ocl prefix has been removed from the names of features such as $OclAny$, $oclIsTypeOf$ or $oclIsKindOf$ (in EOL they are called Any, $isTypeOf$, $isKindOf$). Moreover, built-in operations such as $select()$ and $size()$ that are accessible using the $\rightarrow$ operation in OCL, are also accessible using the point operator in EOL.

3.2 Contents and Structure of an Alignment Specification

To align EOL with a DSL, we need to construct an *alignment specification*. Such a specification consists of the following operations (or *helpers* in OCL terms) that operate at the meta-model level and define the required semantics:

operation String allOfType() : Sequence The $allOfType$ operation applies to a String that specifies the name of the type under question and returns all the model elements that are direct instances of the type. This is needed both to be able to calculate the result of the $isTypeOf$ operation at the instance-level.

operation String allOfKind() : Sequence The $allOfKind$ operation applies on a String that specifies the name of the type under question and returns all the model elements that are either direct or indirect (through some kind of inheritance in the M1 level) instances of the type. The $allOfKind$ method is needed to be able to calculate the result of the $isKindOf$ and the $allInstances$ operations at the instance-level. The existence of both the $allOfKind$ and the $allOfType$ operations allows us to support inheritance in the model-level (if the DSL supports such a feature).

operation Type getProperty(property : String) : Any For each $Type$ of instance at the instance-level, a $getProperty$ operation must be defined that specifies the semantics of the point navigational operator in the model-level. As discussed in Section 2, a $getProperty$ operation must define: the navigation path for retrieving the value of the $property$, the multiplicity and the type of the returned value.

3.3 Implementation Architecture

A basic design principle of EOL was that it should be able to manage models of diverse metamodels and technologies. This principle is implemented in the underlying Epsilon Model Connectivity (EMC) layer. The basic concept of EMC is the $EolModel$ interface to which all EOL-compatible models must conform. Implementations of $EolModel$ include the $MdrModel$, $EmfModel$ and $XmlDocument$ that allow EOL to manage MDR [18] and EMF-based [8] models as well as XML documents. In the aforementioned implementations of $EolModel$, the required methods (e.g. $allOfType$, $allOfKind$) are specified using Java.

To align with custom DSLs we have defined a new specialization of $EolModel$ named $EolM0Model$ that delegates calls to its methods to the underlying alignment specification (instead of implementing them in Java). For example, if the instance-level query contains the $X.allInstances$ expression, the EOL engine will invoke the $allOfKind(X)$ Java-method of the $EolM0Model$ that will in its turn delegate the call to the $allOfKind(X)$ EOL operation defined in the alignment specification. This is illustrated in Figure 2.

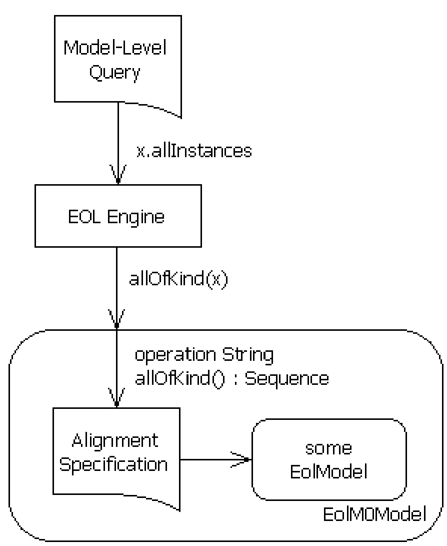

Fig. 2. Architecture of the alignment mechanism

Using this approach, to align with a new DSL, engineers do not need to be aware of the internals of the execution engine, the modelling framework (EMF, MDR etc) or write code in the implementation language of the engine (Java) at all. Instead, they need only provide a high-level alignment specification, in EOL, that implements the required operations.

4 Case Study

Having outlined our approach in Section 3, in this section we present a case-study of aligning OCL with a DSL for modelling Relational Databases. The metamodel of the Relational DSL (constructed using EMF) is presented graphically in Figure 3. There, a $Database$ consists of many tables and each $Table$ consists of a number of $Columns$. All $Database$, $Table$ and $Column$ have a $name$ and $Column$ also has a $type$. Related columns are linked each other using foreign-keys. Each $ForeignKey$ defines a $parent$ and a $child$ column and also if the relationship is one-to-one or one-to-many ($oneToMany$). In the shaded part of the metamodel the *M0* constructs[1] appear. A $TableData$ contains a set of $Rows$ that represent exemplar data of the related $table$. Finally, a Row contains many cells and each $Cell$ corresponds to a $column$ of the table and also has a $value$.

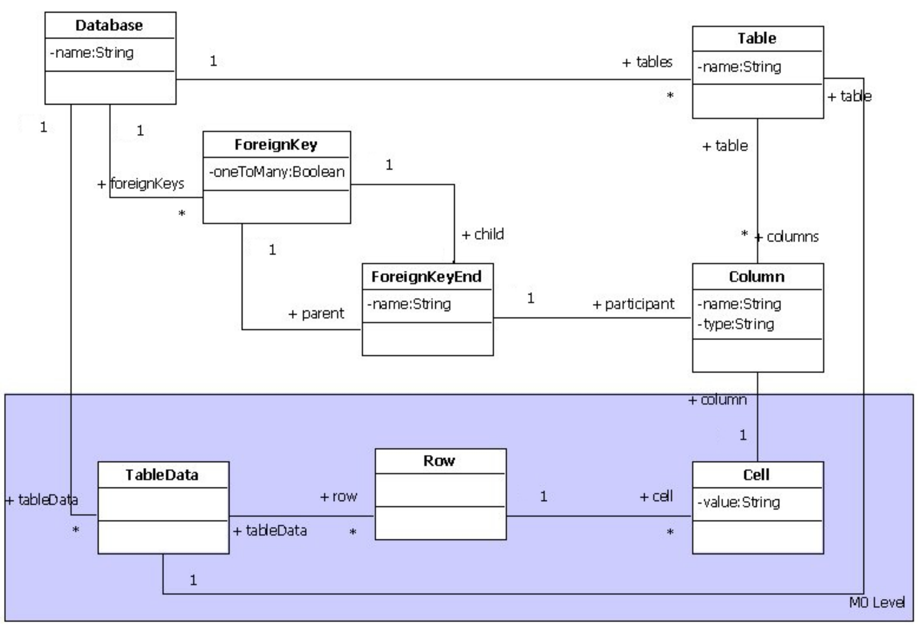

Fig. 3. The abstract syntax of a DSL for Relational Databases

[1] By *M0 constructs* of the metamodel, we refer to metamodel constructs, instances of which belong to the M0 level

A visualized version of an instance of the Relational DSL is illustrated in Figure 4. There, the two shapes on the top represent instances of $Table$ and the two shapes at the bottom represent instances of $TableData$.

4.1 Defining the $M1 \rightarrow M0$ instantiation semantics

In our DSL, a Row is conceptually an instance of a $Table$. Therefore, the expression $Customer.allInstances$ should return all the rows in the model that belong to the $TableData$ that has an associated $Table$ with the name *Customer*. This is formally defined by the $allOfKind$ operation of Listing 1.3. In Listing 1.3, the $allOfType$ operation is also defined. The fact that they both return the same result indicates that there is no notion of inheritance in our DSL.

Listing 1.3. Specification of the allOfType and allOfKind operations

```
operation String allOfType() : Sequence(Row) {
      return Row.allInstances().
       select(r|r.tableData.table.name = self);
}

operation String allOfKind() : Sequence(Row) {
      return self.allOfType();
}
```

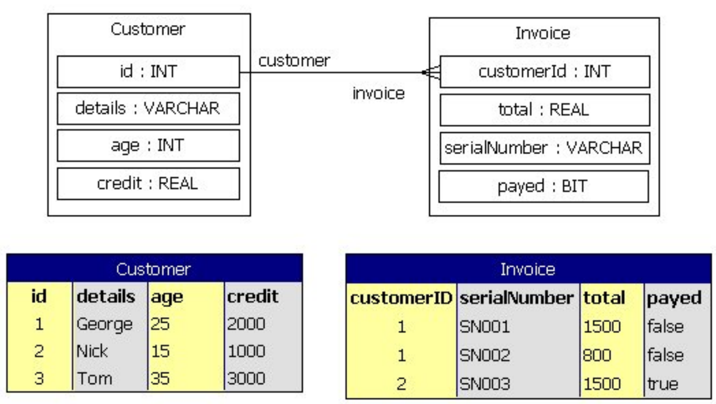

Fig. 4. An instance of the Relational DSL

4.2 Defining the point operator semantics

Having defined the $M1 \rightarrow M0$ instantiation semantics, we must now define the semantics of the point operator for the instance level. To provide better understanding, we first describe the semantics informally through a set of small examples: Let c be the first row of the Customer table-data displayed in Figure 4. In this case, the expression $c.age$ should return an $Integer$ (25). Similarly, $c.details$ should return a $String$ (George). Moreover, $c.invoice$ should return a collection of all the rows of the Invoice table-data where the value of $customerId$ is equal to the value of $c.id$. This is dictated by the foreign key that relates the repsective columns in the model. The complete formal semantics of the point operator are captured in the $getProperty(name : String)$ operation of Listing 1.4. The $getProperty$ operation delegates the task of defining the navigation path and the multiplicity of the returned result to the $getRowsOrCell()$ operation. Finally, the $getValue()$ operation (lines 12-23), casts the string values of cells to the respective OCL data-types (Boolean, String, Integer and Real) according to the the $type$ of the respective $Column$ (BIT, VARCHAR, INT and REAL).

Listing 1.4. Specification of the getProperty operation

```
1  operation Row getProperty(name : String) : Any {
2   def ret : Any;
3   ret := self.getCellOrRows(name);
4   if (ret.isTypeOf(Cell)){
5    return ret.getValue();
6   }
7   else {
8    return ret;
9   }
10 }
11
12 operation Cell getValue() : Any {
13  if (cell.column.type = 'INT'){
14   return cell.value.asInteger();
15  }
16  if (cell.column.type = 'BIT'){
17   return cell.value.asBoolean();
18  }
19  if (cell.column.type = 'REAL'){
20   return cell.value.asReal();
21  }
22  return cell.value.asString();
23 }
24
25 operation Row getCellOrRows(name : String) : Any {
26
27  def cell : Cell;
28
29  -- First try to find a cell with that name
30  cell := self.cell.select(c|c.column.name = name).first();
31
32  if (cell.isDefined()){
33   -- If a cell with that name exists, return it
34   return cell;
35  }
```

```
  else {
    -- Try to find a foreign child-key with that name
    def childKeyCell : Cell;

    childKeyCell := self.cell.select
      (c|ForeignKey.allInstances().
        exists(fk|fk.child.participant =
          c.column and fk.parent.name = name)).first();

    if (childKeyCell.isDefined()) {
      def ck : ForeignKey;
      ck := ForeignKey.allInstances().
        select(fk|fk.child.participant = childKeyCell.column)
          .first();
      return Row.allInstances().
        select(r|r.cell.exists(c|c.column = ck.parent.participant
          and c.value = childKeyCell.value)).first();
    }
    else {
      -- Try to find a foreign parent-key with that name
      def parentKeyCell : Cell;
      parentKeyCell := self.cell.select
        (c|ForeignKey.allInstances()
          .exists(fk|fk.parent.participant = c.column
            and fk.child.name = name)).first();

      if (parentKeyCell.isDefined()) {
        def pk : ForeignKey;
        pk := ForeignKey.allInstances().
          select(fk|fk.parent.participant = parentKeyCell.column)
            .first();
        def rows : Sequence(Row);
        rows := Row.allInstances().
          select(r|r.cell.exists(c|c.column = pk.child.participant and
            c.value=parentKeyCell.value));
        if (pk.oneToMany){
          return rows;
        }
        else{
          return rows.first();
        }
      }

    }

  }
  throw 'Undefined property: ' + name;
}
```

4.3 Running instance-level queries on the model

Having defined the alignment specification, we can now express and evaluate OCL instance-level queries on our model. The OCL expression of Listing 1.5 returns a $Collection$ of the $details$ of all the customers in our model that have an age under 18 ('Nick'). In a more complex query, Listing 1.6 prints a message for every customer that has unpayed invoices, the sum of which exceed his/her credit.

Listing 1.5. Instance-level query for retrieving under-aged customers

```
Customer.allInstances.select(c|c.age < 18).collect(c|c.details);
```

Listing 1.6. Instance-level query for retrieving customers in debt

```
for (c in Customer.allInstances){

 def balance : Real;

 balance := c.invoice.select(i|i.payed = false)
  .collect(i|i.total).sum();

 if (balance > c.credit){
  ('Customer ' + c.details + ' has a negative balance').println();
 }
}
```

5 Conclusions and Further Work

In this paper we have presented a novel technique for aligning OCL with custom Domain Specific Languages to support instance-level queries. Moreover, we have presented a working example of applying this technique in a DSL for modelling Relational Databases that demonstrates its practicality and usefulness. However, we plan to continue our experiments with diverse metamodels to further validate or refine (where this is required) our approach.

As discussed in Section 3, we are using EOL instead of pure OCL for defining the alignment specification. This is primarily due to the practical difficulties involved in capturing complex expressions such as this displayed in the $getRowOrCells$ operation of Listing 1.4 using pure OCL. However, we realize that expressing the alignment specification in that way renders re-use from plain OCL engines impossible. Therefore, we are considering developing a transformation from EOL to pure OCL that will be able to translate sequential EOL statements into a single OCL-compatible statement.

6 Acknowledgements

The work in this paper was supported by the European Commission via the MODELWARE project. The MODELWARE project is co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006).

References

1. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net.
2. MOFScript. Official Web-Site: http://www.modelbased.net/mofscript/.
3. OCLE: Object Constraint Language Environment, official web-site. http://lci.cs.ubbcluj.ro/ocle/.
4. Octopus: OCL Tool for Precise Uml Specifications, official web-site. http://www.klasse.nl/ocl/octopus-intro.html.
5. D. H. Akehurst and Behzad Bordbar. On Querying UML Data Models with OCL. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 91–103, London, UK, 2001. Springer-Verlag.
6. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, Genova, Italy, October 2006. LNCS. (to appear).
7. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
8. Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf.
9. Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.
10. Ivan Kurtev, Klaas van den Berg. Unifying Approach for Model Transformations in the MOF Metamodeling Architecture. Technical Report TR-CTIT-04-12, University of Twente, 2004. CTIT Technical Report, ISSN 1381-3625.
11. Jean Bezivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
12. Jean Bézivin and Richard Lemesle. Ontology-Based Layered Semantics for Precise OA&D Modeling. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 151–154, London, UK, 1998. Springer-Verlag.
13. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
14. Object Management Group. MOF QVT Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf.
15. Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.
16. Object Management Group. UML official web-site. http://www.uml.org.
17. Octavian Patrascoiu and Peter Rodgers. Embedding OCL Expressions in YATL. In *Proc. OCL and Model Driven Engineering wokshop, UML'04*, October 2004.
18. Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.
19. University of Bremen, Database Systems Group. USE - A UML-based Specification Environment. http://www.db.informatik.uni-bremen.de/projects/USE/.