$\mathbb{A}_{\mathbb{ocl}}$: A Pure-Java Constraint and Transformation Language for MDE

Don Batory and Najd Altoyan
Department of Computer Science, The University of Texas at Austin, Austin, Texas, U.S.A.

Keywords: Java MDE Constraint Language, Java MDE Transformation Language.

Abstract: OCL is a standard MDE language to express constraints. OCL has been criticized for being too complicated, over-engineered, and difficult to learn. But beneath OCL's complicated exterior is an elegant language based on relational algebra. We call this language $\mathbb{A}_{\mathbb{ocl}}$, which has a straightforward implementation in Java. $\mathbb{A}_{\mathbb{ocl}}$ can be used to write OCL-like constraints and model transformations in Java. A simple MDE tool generates an $\mathbb{A}_{\mathbb{ocl}}$ Java 8.0 package from an input class diagram for $\mathbb{A}_{\mathbb{ocl}}$ to be used.

1 INTRODUCTION

A central issue in *Model Driven Engineering* MDE is tooling: How can MDE tools be easier to learn, use, and maintain? This is not new: a visionary 2004 paper by (Favre, 2004) raised similar concerns by advocating a rethinking of MDE basics from the ground-up. The *Object Constraint Language* OCL (OMG, 2019) has not gone unscathed (Cabot and Gogolla, 2019; Avila et al., 2010; Wilke and Demuth, 2011; Bauerdick et al., 2004; Brucker and et al, 2014; Fuentes et al., 2003; Cadavid et al., 2011).

Unease about OCL's complexity transcends MDE where a simple constraint language for UML class diagrams is needed. For years, researchers in *Software Product Lines* SPLs explored generalizations of feature models to admit replicated features, feature attributes, and numerical features (Eichelberger and Schmid, 2015; Czarnecki et al., 2006). Doing so generalizes trees of features (a.k.a., *feature models*) where propositional logic was sufficient to express constraints (Apel et al., 2013), to class diagrams where first-order logic and languages like OCL are required (Czarnecki et al., 2006). Of course, there has been resistance in adopting OCL outright by SPL researchers for the reasons in the first paragraph.

There is also the intellectual challenge to find an alternative to OCL that matches its power but is simple and elegant. Imagine the damage COBOL would have inflicted on programming and Computer Science if we all were required to use it into the 1980s. Any early language is not, nor should be, an absolute endpoint.

Against this backdrop, today's *Object Oriented* OO programming languages have made great strides in the last 25 years; Java 8.0 is vastly different than Java 1.0. We demonstrate in this paper that contemporary OO languages now have the functionality to replace specialized languages used in MDE, like OCL and ATL. Our work is simply a next step in the evolution of MDE concepts and tooling.

Where might a replacement or simplification of OCL be found? Researchers with a graduate understanding of classical databases have long recognized the connections between MDE and relational algebra (Karsai et al., 2006). Independently, category theory is a mathematical foundation for MDE; categorical concepts are finding their way into today's MDE tools and texts (Ehrig and et al., 2006; Diskin and Maibaum, 2012; Mabrok and Ryan, 2015). What would result if these foundational lines of thought were unified?

(Freyd and Scedrov, 1990) studied categories with power set domains called *allegories*. (Zieliński et al., 2013) showed how allegories were closely connected to database modeling and query processing. Allegories were noticed by mathematicians but not so by the database and MDE communities.

This paper is not an immediate response to reading these pioneering works on allegories; it took years to understand and integrate these ideas and realize their implications and utility.

To our delight, allegories offer a clean way to express constraints from a relational algebra perspective. Our language, called $\mathbb{A}_{\mathbb{ocl}}$, is pure-Java and is implemented by a Java framework that relies on Java streams, generics, and lambda expressions. Using $\mathbb{A}_{\mathbb{ocl}}$ to write and evaluate model constraints requires

Batory, D. and Altoyan, N.
Aocl : A Pure-Java Constraint and Transformation Language for MDE.
DOI: 10.5220/0008942803190327
In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 319-327
ISBN: 978-989-758-400-8; ISSN: 2184-4348

an MDE tool to generate a Java 8.0 package for a given class diagram of the target metamodel.

$\mathbb{A}_{ocl}$ is a pragmatic response to the motivations of this paper. It is a simple, extensible (meaning new operations can be added), pure-Java replacement for OCL. The $\mathbb{A}_{ocl}$ codebase is ∼9K Java LOC and can be prototyped in a few months on any MDE platform.

2 $\mathbb{A}_{ocl}$

2.1 Insights Behind $\mathbb{A}_{ocl}$

Below is the `EDD` class diagram. It says there are `Employees`, `Departments`, and `Divisions`. Each `Emp` works in any number of `Deps` and each `Dep` employs any number of `Emps`. Each `Dep` belongs to a single `Div`.

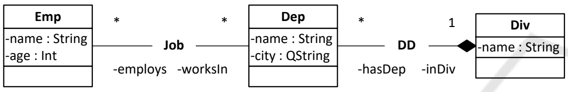


Figure 1: The `Emp-Dep-Div` or `EDD` Class Diagram.

Here is a query written in USE OCL (USE, 2019) to find employees in the `tool` division:

```
Div.allInstances.select(name='tool').hasDep.employs
```

Its meaning is straightforward:

- `Div.allInstances` produces all `Div` objects;
- `select(name='tool')` eliminates `Div` objects whose name is not `tool`;
- `hasDep` produces `Dep` objects that belong to `tool` divisions; and
- `employs` produces `Emp` objects that work in `tool` divisions, which is the result of the query.

Written in this way, the connection between relational databases and OCL emerges when a relational algebra analog to this query is written in OO style/syntax:

```
Div.select(name.equals("tool"))
   .hasDep().employs()                              (1)
```

- `Div` is the table of all `Div` tuples;
- `select(name.equals("tool"))` eliminates `Div` tuples whose name is not `tool`;
- `hasDep()` produces the table of `Dep` tuples that are referenced by qualified `Div` tuples. In database parlance, this operation is a *right-semijoin* of qualified `Div` tuples with the entire `Dep` table (Silberschatz et al., 2006; Wikipedia, 2017); and
- `employs()` is another right-semijoin that produces the table of `Emp` tuples that work in qualified departments.

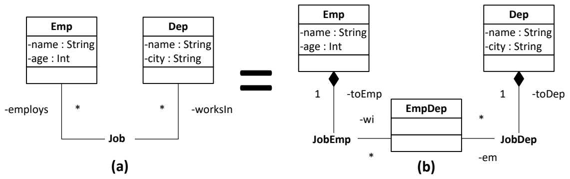


Figure 2: Database Normalization of the `Job` Association.

Query `(1)` could have been written using only relational algebra operations, making explicit the semijoin argument — here an association name — for each right-semijoin:

```
Div.select(name.equals("tool"))
   .rightSemiJoin(hasDep)
   .rightSemiJoin(employs)
```

This is ugly. However, by lifting association role names to their corresponding semijoin operations yields the compact expression `(1)`.

Note: A bit of database sugaring was used in this example. `Job` is a many-to-many association between `Emp` and `Dep` (Fig. 2 a). Classical relational database design, called *normalization*, replaces association `Job` with an association class `Job` and two one-to-many associations `JobEmp` and `JobDep` (Fig. 2 b) (Elmasri and Navathe, 1999; Silberschatz et al., 2006). In MDE parlance, the transformation of Fig. 2 a to Fig. 2 b is a model refactoring (France et al., 2003).

Association traversals in `(1)` and Fig. 2 a are cascading right-semijoins in Fig. 2 b. Written as Java composed methods where `A().B()` means evaluate `A()` first, then `B()`:

```
worksIn()  =  wi().toDep()
employs()  =  em().toEmp()
```

That is, `worksIn()` is a traversal (right-semijoin) from `Emp` to `Dep` in Fig. 2 a. In Fig. 2 b, `worksIn()` is a right-semijoin from `Emp` to `Job` via association `wi()` and then another right-semijoin from `Job` to `Dep` via `toDep()`. Of course, these details can be hidden from end-users.

In a nutshell, the essence of OCL is relational algebra written in OO syntax with customized names for right-semijoins. We call this language $\mathbb{A}_{ocl}$.

Foundations and Concessions. Our presentation is incomplete in that the theory that inspired $\mathbb{A}_{ocl}$, and which existed long before $\mathbb{A}_{ocl}$ itself, should be presented next. For lack of space and as few in MDE appreciate category theory (and far fewer allegories), the usual theory-then-implementation order is presented in a technical report (Batory and Altoyan, 2019).

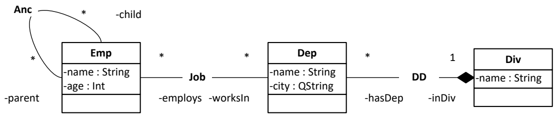


Figure 3: `EDD` with a Recursive, Lineage Association `Anc`.

```
dbase(EDD,[Emp,Dep,Div,Anc,Job]).

table(Emp,[id,name,age:int]).
Emp(p1,don,64).
Emp(p2,karen,57).
Emp(p3,hanna,23).
Emp(p4,alex,18).
Emp(p5,steve,53).
Emp(p6,priscila,28).
Emp(p7,hanna,73).
Emp(p8,kelly,58).
Emp(p9,phyllis,56).

table(Dep,[id,name,"city",inDiv:Div]).
Dep(d1,mens,"Austin",v1).
Dep(d2,womens,"Austin",v1).
Dep(d3,appliances,"Toronto",v2).
Dep(d4,hardware,"Toronto",v2).
Dep(d5,book,"Hamilton",v2).
```

```
table(Div,[id,name]).
Div(v1,clothing).
Div(v2,goods).

table(Anc,[id,parent:Emp,child:Emp]).
Anc(c1,p1,p3).
Anc(c2,p2,p3).
Anc(c3,p1,p4).
Anc(c4,p2,p4).
Anc(c5,p5,p1).

table(Job,[id,employs:Emp,worksIn:Dep]).
Job(w1,p1,d1).
Job(w2,p2,d2).
Job(w3,p3,d2).
Job(w4,p4,d4).
Job(w5,p5,d3).
Job(w6,p6,d2).
Job(w7,p7,d2).
Job(w8,p1,d3).
Job(w9,p8,d5).
Job(w10,p9,d5).
```

Figure 4: An `EDD` Database Instance.

2.2 Running Example

We add a recursive association `Anc` to our `EDD` diagram, Fig. 3. Now each `Emp` has a lineage: descendants (children) and ancestors (parents). Traversing the `Anc` association computes `Emps` that are grandparents, by expression `Emp.parent().parent()`, and `Emps` that are grandchildren, by `Emp.child().child()`.

Class Diagram to Relational Schema Mapping. It is well-known that UML class diagrams can be translated into normalized relational schemas (Elmasri and Navathe, 1999). The **`blue`** statements in Fig. 4 are `EDD` schema declarations in **`MDElite`** (Batory et al., 2013), the MDE platform used in this paper. The `dbase` statement declares the `EDD` database to consist of five tables: `Emp`, `Dep`, and `Div`, along with an association table `Job` that encodes n:m relationships among `Emp` and `Dep` tuples, and an association table `Anc` that encodes n:m ancestry information among `Emp`.

The first column of every table is a manufactured identifier `id` required by **`MDElite`**. The primary key of tables `Emp`, `Dep`, and `Div` is their `name` attribute. The manufactured tuple identifier `id` always serves as a tuple key. All facts, `dbase`, `table`, and tuple declarations are written in a Prolog-fact notation.

The `Emp` table of Fig. 4 has 3 columns: `id`, `name`, and `age`. Column `age` is of type `int`; the others default to `String`. Table `Dep` has 4 columns: `id`, `name`, `"city"`, and `inDiv`. The first three columns are of type String. `"city"` means that `city` values are quoted because they may have blanks (e.g., `"New York"`). Attribute `name` has unquoted String values. Column `inDiv` has legal identifiers of `Div` tuples as its values.

Object Model to Database Mapping. An `EDD` model (object diagram) is needed to evaluate queries and constraints. Any `EDD` model can be translated into a database of tuples for the computed `EDD` schema, such as Fig. 4. Again, tuples are written as Prolog facts: `Emp(p1,don,64)` is a `Emp` tuple where `id=p1`, `name=don`, and `age=64`. The `Anc(c1,p1,p3)` tuple has `id=c1`, `parent=p1`, and `child=p3`, meaning `don` is the parent of `hanna`.

Although this example lacks inheritance hierarchies, $\mathbb{A}_{\mathbb{ocl}}$ supports subclasses/subtables as expected.

Constraints and Queries on `EDD`. Here are four constraints to enforce on `EDD`:

`(C1)` Every `Emp` has a unique name.
`(C2)` Every `Dep` in `Toronto` employs `Emps` 19 and older.
`(C3)` No `Div` can employ more than 20 `Emps`.
`(C4)` Grandparents of workers can not be employed.

And here are five non-trivial and progressively more complicated queries that could be used in constraints or in model-to-model transformations:

`(Q1)` Find `Emps` whose name begins with `d` or `p`.
`(Q2)` Find the `Divs` that have `Deps` in Austin.
`(Q3)` List `Emps` that work in multiple `Divs`.
`(Q4)` Print the `Div` colleagues of `priscila`.
`(Q5)` List the `id` of each `Emp` (whose parent is also an `Emp`) with the `id` of division(s) in which he/she works.

We consider queries in the next section and constraints afterwords.

2.3 $\mathbb{A}_{\mathbb{ocl}}$ Queries

An $\mathbb{A}_{\mathbb{ocl}}$ program is a pure-Java program that imports its allegory package and starts by reading a database, here the `EDD` model of Fig. 4:

```
import Allegory.EDD.*;
...
Database edd = new Database("EDD.edd.pl");
```

We can immediately write $\mathbb{A}_{\mathbb{ocl}}$ expressions for each query in Section 2.2. Query outputs are posted in Fig. 5.

`(Q1)` finds employees whose name begins with `d` or `p`. Here is Java ($\mathbb{A}_{\mathbb{ocl}}$) code to compute **`(Q1)`**'s solution:

```
edd.Emp
   .select(e->e.name.startsWith("d") ||
              e.name.startsWith("p"))
   .print();
```

The expression `edd.Emp` yields the `Emp` table. The `select` takes a Java **`Predicate`** as input, which selects `Emp` tuples whose `name` starts with `d` or `p`. Then `print()` displays the `select`-produced table. Its USE OCL counterpart is:[1]

```
Emp.allInstances
   .select(name.at(1)=’d’ or
           name.at(1)=’p’)
```

(Q2) finds divisions that have departments in Austin:

```
edd.Dep
   .select(d->d.city.equals("Austin"))
   .inDiv()
   .print();
```

`Deps` that are in `Austin` are identified by `select()`. Austin `Deps` are mapped by `inDiv()` to their `Divs`, and then printed. Its USE OCL counterpart:

```
Dep.allInstances
   .select(city=‘Austin’)
   .inDiv->asSet
```

(Q3) lists `Emps` that work in multiple `Deps`:

```
edd.Emp
   .select(e->e.worksIn().count()>1)
   .print();
```

The `select()` finds employees that work in more than one department. Its USE OCL counterpart:

```
Emp.allInstances.select(worksIn->size>1)
```

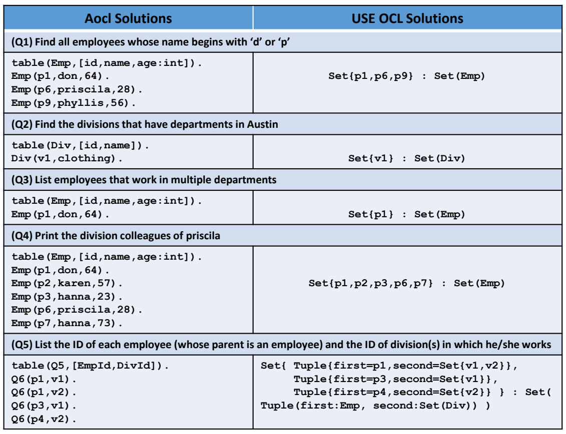
(Q4) prints the division colleagues of `priscila`:

```
edd.Emp
   .select(e->e.name.equals("priscila"))
   .worksIn().inDiv()
   .hasDep().employs()
   .print();
```

`Emp.select()` produces an `Emp` table of `priscila` tuples. `worksIn().inDiv()` produces a table of `Divs` in which `priscila` works. `hasDep().employs()` computes the table of `Emps` that work in those `Divs`. Its USE OCL counterpart is:

```
Emp.allInstances
   .select(name=‘priscila’)
   .worksIn.inDiv.hasDep.employs->asSet
```

(Q5) lists the `id` of each employee (whose parent is an employee) with the `id` of division(s) in which he/she works:



Aocl Solutions	USE OCL Solutions
(Q1) Find all employees whose name begins with ‘d’ or ‘p’	
table(Emp,[id,name,age:int]). Emp(p1,don,64). Emp(p6,priscila,28). Emp(p9,phyllis,56).	Set{p1,p6,p9} : Set(Emp)
(Q2) Find the divisions that have departments in Austin	
table(Div,[id,name]). Div(v1,clothing).	Set{v1} : Set(Div)
(Q3) List employees that work in multiple departments	
table(Emp,[id,name,age:int]). Emp(p1,don,64).	Set{p1} : Set(Emp)
(Q4) Print the division colleagues of priscila	
table(Emp,[id,name,age:int]). Emp(p1,don,64). Emp(p2,karen,57). Emp(p3,hanna,23). Emp(p6,priscila,28). Emp(p7,hanna,73).	Set{p1,p2,p3,p6,p7} : Set(Emp)
(Q5) List the ID of each employee (whose parent is an employee) and the ID of division(s) in which he/she works	
table(Q5,[EmpId,DivId]). Q6(p1,v1). Q6(p1,v2). Q6(p3,v1). Q6(p4,v2).	Set{ Tuple{first=p1,second=Set{v1,v2}}, Tuple{first=p3,second=Set{v1}}, Tuple{first=p4,second=Set{v2}} } : Set(Tuple(first:Emp, second:Set(Div)))

Figure 5: $\mathbb{A}_{ocl}$ and USE OCL Solutions to **(Q1)**–**(Q5)**.

```
DTable Q5 = new DTable("Q5","EmpId","DivId");
edd.Emp
   .select(e->e.parent().exists())
   .forEach(em->em.worksIn().inDiv()
                 .forEach(d->Q5.add(em.id,d.id)));
Q5.print();
```

The first line creates a temporary table `Q5` with column names `EmpId` and `DivId`. The second line selects eligible `Emps`. The `forEach` lines compute `Q5` tuples (ordered pairs). The last line prints table `Q5`. Its USE OCL counterpart is:

```
Emp.allInstances
    .select(parent->notEmpty)->
  iterate(e:Emp;
  ed:Set(Tuple(first:Emp,
               second:Set(Div)))=Set{}
  | ed->including(Tuple{first=e,
         second=e.worksIn.inDiv->asSet}))
```

Observations

- Fig. 5 is the output of $\mathbb{A}_{ocl}$ and USE OCL. Their solutions are identical, albeit different syntax. As these examples showed, $\mathbb{A}_{ocl}$ and OCL expressions are syntactically similar. This is to be expected as both are stream processing languages.
- The methods invoked in the above examples on `EDD` tuples and tables belong to the generated `EDD` Java package. The same holds for the `EDD` constraints we consider next.
- $\mathbb{A}_{ocl}$ follows relational database tradition as tables (sets of tuples) are produced. Tables with duplicates can indeed be produced in $\mathbb{A}_{ocl}$ and by applying a `unique()` operation, duplicates can be removed. Again, this is standard relational database technology (Elmasri and Navathe, 1999). The `Bag`, `Sequence`, `Orderedset`, and `Collection` con-

---

[1] Following `allInstances` in some versions of OCL require `->` or `()->`; our USE OCL is correct.

structs in OCL are clutter from a classical relational database perspective.

- The simplicity of 𝔸ocl syntax relies heavily on Java 8.0 syntax and Java streams. For example, all 𝔸ocl `select` statements require an iteration variable (e.g., `t->`) – this is part of Java 8.0 and absent in earlier versions of Java.

2.4 𝔸ocl Constraints

A special Java class and table operation are used in 𝔸ocl to log constraint violations. `ErrorReport` is a Java class whose stateful objects log errors. Table method `error(er,...)` takes an `ErrorReport` object (`er`) and logs a customized error for each tuple of `error()`'s input table. Constraint outputs are posted in Fig. 6.

𝔸ocl constraint programs begin with the reading of a database and the creation of an `ErrorReport` object:

```
import Allegory.EDD.*;
...
Database  edd = new Database("EDD.edd.pl");
ErrorReport er = new ErrorReport();
```

Constraints can now be written. **`(C1)`** asserts all employees have unique names:

```
String fmt = "multiple employees have name=%s";
edd.Emp
   .name()
   .duplicates()
   .error(er,fmt,e->e.value);
```

`Emp.name()` produces a single-column `STRINGTable` of `Emp` names that preserves duplicates. `duplicates()` retains one copy of each duplicated tuple in a table and eliminates non-duplicates. An error is logged for each tuple in `STRINGTable`. Its USE OCL counterpart:

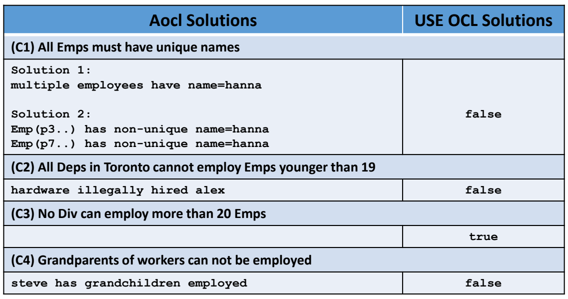
```
context Emp inv UniqueName:
Emp.allInstances
   .forAll(e1,e2 | e1.name=e2.name implies e1=e2)
```

`(C2)` every `Dep` in Toronto hires `Emps` 19 and older:

```
String fmt = ”%s illegally hired %s”;
edd.Dep
   .select(d—>d.city.equals(”Toronto”))
   .forEach(d—>d.employs()
                  .select(ee—>ee.age<19)
                  .error(er,fmt,e—>d.name,e—>e.name));
```

For each `Dep` in Toronto, a table of under-aged `Emps` is computed and each violation is logged. Its USE OCL counterpart:

```
context Dep inv EmpAge:
self.select(city='Toronto')
    .employs->forAll(e|e.age>=19)
```



Aocl Solutions	USE OCL Solutions
(C1) All Emps must have unique names	
Solution 1: multiple employees have name=hanna Solution 2: Emp(p3..) has non-unique name=hanna Emp(p7..) has non-unique name=hanna	false
(C2) All Deps in Toronto cannot employ Emps younger than 19	
hardware illegally hired alex	false
(C3) No Div can employ more than 20 Emps	
	true
(C4) Grandparents of workers can not be employed	
steve has grandchildren employed	false

Figure 6: Error Log of Constraints **`(C1)`**–**`(C4)`**.

`(C3)` says no `Div` can employ more than 20 `Emps`:

```
edd.Div
   .select(d->d.hasDep().employs().count()>20)
   .error(er,"%s has >20 workers",d->d.name);
```

Its USE OCL counterpart:

```
context Div inv EmpCount:
self.hasDep.employs->size()<=20
```

`(C4)` grandparents of workers can not be employed:

```
String fmt = "%s has grandchildren employed";
db.Emp.select(e->e.child().child().exists())
      .error(er, fmt, e->e.name);
```

Its USE OCL counterpart:

```
context Emp inv twoGen:
self.child.child->size() = 0
```

Printing accumulated errors ends a constraint program:

```
er.printEH();
```

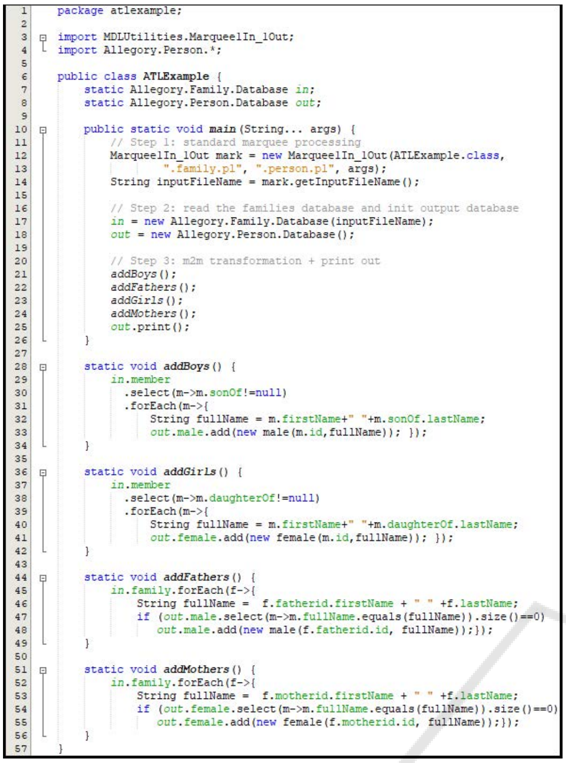
`(C1)`, **`(C2)`** and **`(C4)`** log errors; **`(C3)`** does not.

Observations

- Fig. 6 shows the output of 𝔸ocl and USE OCL queries. The solutions are identical, albeit different syntax. 𝔸ocl errors pin-point their source; USE OCL simply reports `false` when any error is detected, `true` otherwise. The 𝔸ocl reports no errors for **`(C3)`**, and thus is blank.
- USE OCL and 𝔸ocl constraint expressions are comparable in structure with 𝔸ocl expressions a bit longer due to customized error logging.

2.5 Model-to-Model Transformations

OCL cannot update the model that it examines. By precluding updates, 𝔸ocl could behave similarly. By



```java
package atlexample;

import MDLUtilities.MarqueeIIn_1Out;
import Allegory.Person.*;

public class ATLExample {
    static Allegory.Family.Database in;
    static Allegory.Person.Database out;

    public static void main(String... args) {
        // Step 1: standard marquee processing
        MarqueeIIn_1Out mark = new MarqueeIIn_1Out(ATLExample.class,
                ".family.pl", ".person.pl", args);
        String inputFileName = mark.getInputFileName();

        // Step 2: read the families database and init output database
        in = new Allegory.Family.Database(inputFileName);
        out = new Allegory.Person.Database();

        // Step 3: m2m transformation + print out
        addBoys();
        addFathers();
        addGirls();
        addMothers();
        out.print();
    }

    static void addBoys() {
        in.member
          .select(m->m.sonOf!=null)
          .forEach(m->{
              String fullName = m.firstName+" "+m.sonOf.lastName;
              out.male.add(new male(m.id,fullName)); });
    }

    static void addGirls() {
        in.member
          .select(m->m.daughterOf!=null)
          .forEach(m->{
              String fullName = m.firstName+" "+m.daughterOf.lastName;
              out.female.add(new female(m.id,fullName)); });
    }

    static void addFathers() {
        in.family.forEach(f->{
            String fullName =  f.fatherid.firstName + " " +f.lastName;
            if (out.male.select(m->m.fullName.equals(fullName)).size()==0)
              out.male.add(new male(f.fatherid.id, fullName));});
    }

    static void addMothers() {
        in.family.forEach(f->{
            String fullName =  f.motherid.firstName + " " +f.lastName;
            if (out.female.select(m->m.fullName.equals(fullName)).size()==0)
              out.female.add(new female(f.motherid.id, fullName));});
    }
}
```

Figure 7: $\mathbb{A}_{\mathbb{ocl}}$ Families-to-Persons M2M Transformation.

allowing updates, $\mathbb{A}_{\mathbb{ocl}}$ could be used to write model-to-model transformations and be more versatile.

As an illustration, in a few minutes, we coded and executed ATL's **"Families-to-Persons"** example (Jouault, 2007). The Java source for this program is in Fig. 7. (This image is digitally enlargable). We generated $\mathbb{A}_{\mathbb{ocl}}$ packages for the `Families` and `Persons` metamodels, and the rest was easy. We do not foresee problems scaling $\mathbb{A}_{\mathbb{ocl}}$ to large M2M transformations.

3 STATUS

$\mathbb{A}_{\mathbb{ocl}}$ is operational and relies on a Java framework that uses Java generics, lambda functions, and Java streams (MDELite, 2020). An **MDElite** tool called **Meta4** converts a textual specification of a class diagram into a plug-in for the $\mathbb{A}_{\mathbb{ocl}}$ framework. This plug-in defines all classes, tables, and database operations to support an $\mathbb{A}_{\mathbb{ocl}}$ package for that class diagram. For example, a textual specification of the `EDD` diagram of Fig. 3 is:

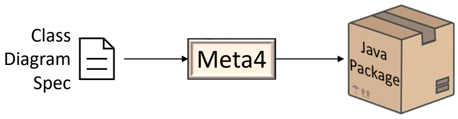
```
classDiagram EDD.

table(Emp,[id,name,age:int]).
table(Dep,[id,name,"city"]).
table(Div,[id,name]).

// continued on next page
assoc Emp employs * -- Dep worksIn *.
assoc Emp parent  * -- Emp  child  *.
assoc Div inDiv   1 -- Dep hasDep  *.

// no inheritance decls in this example
```

From this specification, **Meta4** generates the $\mathbb{A}_{\mathbb{ocl}}$ `EDD` Java 8.0 package with classes for `Emp`, `Dep`, `Div` tuples and tables, hidden association tables and their tuples, a Java class whose instances are `EDD` databases and a **MDElite** database schema:



Meta4 itself is a set of M2T tools and parsers, totaling 6100 Java LOC. Java Generic classes that are shared by all $\mathbb{A}_{\mathbb{ocl}}$ packages is 2800 Java LOC. The size of the generated `EDD` plug-in is 1330 Java LOC. And **MDElite**, the MDE platform on which **Meta4** was built, is 18K Java LOC.

4 THE VALUE PROPOSITION OF $\mathbb{A}_{\mathbb{ocl}}$

Must MDE use special-purpose programming and constraint languages like OCL(OMG, 2019), ATL (ATL, 2005), and QVT (OMG, 2016) that require their own compiler and IDE-like infrastructure, when a standard and richer infrastructure that Java 11 provides might suffice? $\mathbb{A}_{\mathbb{ocl}}$'s existence suggests not. Here are additional pros-and-cons:

Pros. Maintaining a custom language, compiler, debugger, refactoring tools, document tools, etc. is a long-term and costly burden that few research efforts can afford. Modern programming languages have come a long way in the last 20 years. Java 11 (2018) is vastly different than Java 1 (1996). The combination of generics (Java 5), lambda expressions and streams (Java 8), with compiler, debugging, documentation, and refactoring support offers a modern programming environment that makes $\mathbb{A}_{\mathbb{ocl}}$ appealing.

Even if OCL and its infrastructure were perfect today, they must be maintained and extended tomorrow. Extending tools that are Java packages, like **Meta4**, is easier and less costly. And replacing arcane languages with custom packages in modern languages can entice more people to the MDE community. It certainly would reduce the long-term burden of MDE tool support and tool education.

Cons. A perceived important down-side of 𝔸𝕠𝕔𝕝 is that it is a platform dependent language rather than a platform *independent* language. So how could a platform dependent language be advantageous? Ans#1: Recall the history of distributed programming. CORBA initially offered a language-independent front-end for multi-platform programming. It has given way to language annotations that bridge the gap (Oracle, 2019), eliminating arcane CORBA languages entirely and replacing them with Java-derivable WSDLs for platform independence. This could be done for MDE. Ans#2: The tools of an area reflect the quality of teaching material. OCL is about 20 years old; Eclipse MDE tools are about 15. They are first generation and should be celebrated for their success. We can do better now, by reducing ideas that were considered new 20 years ago to well-established Computer Science technologies, making MDE concepts, languages, and tools more elegant, easier to learn, and use.

5 RELATED WORK

Embedding database queries in Java and other languages is common (Meijer et al., 2006). (Cheney et al., 2013) proposed quoting mechanisms for Java to enclose SQL-like queries. (Cook and Wiedermann, 2011) took a broader view, recognizing that quoted blocks of SQL or a subset of Java can provide elegant language support for service oriented architectures and database processing. 𝔸𝕠𝕔𝕝 is an even closer integration where Java packages express database (or relational algebra) computations.

General-purpose tools, like `Xtend` and `Xbase`, integrate DSLs (e.g., OCL constructs) with Java and other languages (XText, 2017). Of course, these tools are necessarily heavier-weight than 𝔸𝕠𝕔𝕝 as 𝔸𝕠𝕔𝕝 is simply a package requiring no language engineering at all.

Another approach implements OCL as a Java package (interpreter) (Eclipse, 2019). OCL queries are submitted as Java Strings to this package (much like SQL strings are submitted to SQL packages for execution). The results are returned as Java objects for subsequent processing. 𝔸𝕠𝕔𝕝 eliminates this middleware approach to express OCL-like queries natively, invoking methods of an 𝔸𝕠𝕔𝕝-generated package for direct execution.

Several projects translated OCL into Java (Shidqie, 2007; Kallel and et al., 2016). These particular projects were completed before Java 8 (2014) was released, where streams and lambda functions first appeared. The translations to Java 7.0 and earlier versions are verbose and not as elegant as their OCL and 𝔸𝕠𝕔𝕝 counterparts. (Yue and Ali, 2016) compared OCL and Java when writing constraints. Java 7 was used, meaning that the Java code was (as above) more verbose than OCL. Never-the-less, the authors found that participants working with OCL and Java performed equally well, with an edge to OCL when constraints became complicated. 𝔸𝕠𝕔𝕝 should reduce this advantage. A goal was to "find a way to ... offset the investment in terms of training and tool support ... for OCL". 𝔸𝕠𝕔𝕝 does not eliminate this cost, but reduces it to learning a Java package, which is less intimidating.

(Rumpe, 2002) proposed ≪`Java`≫`OCL` to (a) adjust OCL syntax closer to that of Java to make it more familiar to Java developers. He examined the OCL meta operations (e.g., `OclAny`, `OclType`, `OclExpression`, `oclAsType`) that we believe are more elegantly handled in Java. His underlying motivation (in our opinion) was similar to that of Yue and Ali (Yue and Ali, 2016): to offset the investment in OCL training.

And finally, (Vaziri and Jackson, 2000) argue that a language like Alloy would be more appropriate than OCL to express constraints, as OCL is "too implementation oriented". Declarative languages always need some escape to code to express certain concepts. 𝔸𝕠𝕔𝕝 is a compromise between too-high and too-low a specification language.

6 CONCLUSIONS

𝔸𝕠𝕔𝕝 is a lightweight, simple, and pure-Java alternative to OCL and special-purpose MDE transformation languages. 𝔸𝕠𝕔𝕝 queries and constraints are syntactically similar (with comparable complexity) to those of OCL. But 𝔸𝕠𝕔𝕝 is more streamlined as it is pure Java, relying on Java syntax and semantics. We believe this will simplify next-generation MDE tooling and teaching – critical problems in their own right.

Modern programming languages are constantly improving. Our experience with Java generics, lambda expressions and streams have convinced us that Java can effectively compete with some of yesterday's special-purpose languages. The trade-off replaces an ecosystem of intertwined special-purpose programming languages with their massive IDE infrastructure (all of which must be maintained) with small Java libraries. We argued that maintaining Java libraries will be more cost effective in the long-run and the maintenance of infrastructure becomes the rightful burden of a small set of language and IDE developers that have the resources for such efforts.

MDE users will also benefit: the cost of entry using well-known modern languages will be lower than it is for out-dated specialized legacy languages.

ACKNOWLEDGMENTS

We thank the referees for their helpful comments, Martin Gogolla, Jean-Marc Jézéquel, and Jeff Gray for additional citations on OCL, and Maider Azanza and Arantza Irastorza for their insights on OCL.

Batory and Altoyan are supported by `NSF` grant `CCF1212683`. Altoyan is also supported by King Abdulaziz City for Science and Technology (`KACST`), Riyadh, Saudi Arabia.

REFERENCES

Apel, S., Batory, D., Kaestner, C., and Saake, G. (2013). *Feature-oriented software product lines: concepts and implementation*. Springer.

ATL (2005). Atl: Atlas transformation language. https://www.eclipse.org/atl/documentation/old/ATL_VMSpecification[v00.01].pdf.

Avila, C., Sarcar, A., Cheon, Y., and Yeep, C. (2010). Runtime constraint checking approaches for ocl, a critical comparison. In *SEKE*.

Batory, D. and Altoyan, N. (2019). Aocl: A pure java constraint and transformation language for mde. University of Texas Computer Science TR-19-04.

Batory, D., Latimer, E., and Azanza, M. (2013). Teaching Model Driven Engineering from a Relational Database Perspective. In *MODELS*.

Bauerdick, H., Gogolla, M., and Gutsche, F. (2004). Detecting ocl traps in the uml 2.0 superstructure: An experience report. In *UML*.

Brucker, A. D. and et al (2014). Panel discussion: Proposals for improving ocl. In *MODELS 2014 OCL Workshop (OCL 2014)*.

Cabot, J. and Gogolla, M. (2019). Object constraint language: A definitive guide. https://www.slideshare.net/jcabot/ocl-tutorial.

Cadavid, J., Baudry, B., and Combemale, B. (2011). Empirical evaluation of the conjunct use of mof and ocl. In *EESSMOD workshop at MODELS'11*.

Cheney, J., Lindley, S., and Wadler, P. (2013). A practical theory of language-integrated query. In *ICFP*.

Cook, W. R. and Wiedermann, B. (2011). Remote batch invocation for SQL databases. In *DBPL*.

Czarnecki, K., Kim, C. H. P., and Kalleberg, K. T. (2006). Feature models are views on ontologies. In *SPLC*.

Diskin, Z. and Maibaum, T. S. E. (2012). Category theory and model-driven engineering: From formal semantics to design patterns and beyond. In *ACCAT*.

Eclipse (2019). Evaluating constraints and queries. https://help.eclipse.org/2019-09/index.jsp.

Ehrig, H. and et al. (2006). Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inf.*

Eichelberger, H. and Schmid, K. (2015). Mapping the design-space of textual variability modeling languages: A refined analysis. *Int. J. Softw. Tools Technol. Transf.*

Elmasri, R. A. and Navathe, S. B. (1999). *Fundamentals of Database Systems*. Addison-Wesley.

Favre, J.-M. (2004). Towards a Basic Theory to Model Model Driven Engineering. In *Workshop on Software Model Engineering, WISME 2004*.

France, R., Ghosh, S., Song, E., and Kim, D.-K. (2003). A metamodeling approach to pattern-based model refactoring. *IEEE Softw.*

Freyd, P. J. and Scedrov, A. (1990). *Categories, Allegories*. Elsevier Science Publishers.

Fuentes, J., Quintana, V., Llorens, J., Genova, G., and Prieto-Diaz, R. (2003). Errors in the uml metamodel? *ACM SIGSOFT Software Engineering Notes*, 28:3.

Jouault (2007). Families to persons. https://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf.

Kallel, S. and et al. (2016). Automatic translation of ocl meta-level constraints into java meta-programs. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015*.

Karsai, G., Ledeczi, A., Neema, S., and Sztipanovits, J. (2006). The MIC Toolsuite: Metaprogrammable Tools for Embedded Control System Design. In *IEEE CCACSD*.

Mabrok, M. and Ryan, M. (2015). Category theory as a formal mathematical foundation for model-based systems engineering. *Applied Mathematics and Information Sciences*.

MDELite (2020). Mdelite and aocl. http://www.cs.utexas.edu/users/schwartz/MDELite/index.html.

Meijer, E., Beckman, B., and Bierman, G. (2006). Linq: Reconciling object, relations and xml in the .net framework. In *SIGMOD*.

OMG (2016). Mof 2.0 qvt specification. https://www.omg.org/spec/QVT/1.3/PDF.

OMG (2019). About the object constraint language specification version 2.4. https://www.omg.org/spec/OCL/About-OCL/.

Oracle (2019). Intro to java web services. https://docs.oracle.com/javaee/6/tutorial/doc/gijti.html.

Rumpe, B. (2002). Javaocl based on new presentation of the ocl-syntax. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Springer Berlin Heidelberg.

Shidqie, A. (2007). Compilation of ocl into java for the eclipse ocl implementation. Masters Thesis Tech. University Hamburg-Harburg.

Silberschatz, A., Korth, H., and Sudarshan, S. (2006). *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition.

USE (2019). Use:uml-based specification environment. https://sourceforge.net/projects/useocl/.

Vaziri, M. and Jackson, D. (2000). Some shortcomings of ocl, the object constraint language of uml. In *TOOLS*.

Wikipedia (2017). Relational algebra. https://en.wikipedia.org/wiki/Relational_algebra.

Wilke, C. and Demuth, B. (2011). Uml is still inconsistent! how to improve ocl constraints in the uml 2.3 superstructure. In *Wkshop on OCL and Textual Modelling*.

XText (2017). Xtext: Language engineering for everyone! https://www.eclipse.org/Xtext/index.html.

Yue, T. and Ali, S. (2016). Empirically evaluating ocl and java for specifying constraints on uml models. *Software & Systems Modeling*.

Zieliński, B., Maślanka, P., and Sobieski, Ś. (2013). Allegories for database modeling. In *International Conference on Model and Data Engineering*.