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

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

Najd Altoyan
University of Texas at Austin
naltoyan@cs.utexas.edu

ABSTRACT

OCL is a standard language in MDE to express metamodel constraints. Since its inception, OCL has been criticized for being too complicated, over-engineered, and difficult to learn. We have discovered that underneath OCL's design is a streamlined design based pm relational algebra. $\mathbb{A}_{ocl}$ can replace OCL; it can be used to write OCL-like constraints and model transformations in Java. The theoretical foundation for $\mathbb{A}_{ocl}$ is allegories, a unification of category theory and relational algebra. A simple MDE tool generates an $\mathbb{A}_{ocl}$ Java plug-in from an input class diagram.

ACM Reference Format:
Don Batory and Najd Altoyan. 2017. $\mathbb{A}_{ocl}$: A Pure-Java Constraint Language for MDE. In *Proceedings of conference.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3106195.3106201

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 2005 paper by Favre [23] raised similar concerns by advocating a rethinking of MDE basics from the ground-up. The *Object Constraint Language* (OCL) has not gone unscathed [2, 9, 10, 12, 13, 26, 52].

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 [17, 21]. Doing so generalizes trees of features (*aka*., *feature models*) where propositional logic was sufficient to express constraints [1, 4], to class diagrams where first-order logic and languages like OCL are required [17]. Of course, there has been resistance in adopting OCL outright by SPL researchers for the reasons in the first paragraph.

And then there is 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 [8, 32]. Independently, category theory is a mathematical foundation for MDE; categorical concepts are finding their way into today's MDE tools and texts [7, 18, 20, 35]. But what would be the result if these foundational lines of thought were unified?

In 1990, Freyd and Scedrov studied categories with power set domains which they called *allegories* [25]. In 2013, Zieliński, Maślanka, and Sobieski explained how allegories were closely connected to database modeling and query processing [56]. 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 of rumination to understand and integrate these ideas and realize their implications and utility.

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

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

The contributions of this paper are:

- Closing the intellectual gap between OCL, allegories, and $\mathbb{A}_{ocl}$;
- Illustrating $\mathbb{A}_{ocl}$ queries and constraints;
- Describing how the $\mathbb{A}_{ocl}$ plug-in generator was built;
- Explaining the potential of $\mathbb{A}_{ocl}$ in future MDE tooling; and
- Listing open problems for the MDE community to explore.

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

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

Fig. 1 is a UML 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. Also each Dep belongs to a single Div.

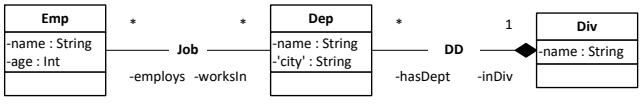

Figure 1: The Emp-Dep-Div (EDD) Class Diagram.

Here is a query written in USE OCL [46]: Find employees in the "tool" division:

conference, month-day-year, place

ACM ISBN 978-1-4503-5221-5/17/09…$15.00
https://doi.org/10.1145/3106195.3106201

```
Div.allInstances->select(name='tool').hasDept.employs
```

Its meaning is straightforward:

- `Div.allInstances` produces the set of all `Div` objects;
- `select(name='tool')` eliminates `Div` objects whose name is not `"tool"`;
- `hasDept` produces the set of `Dep` objects that belong to `"tool"` divisions; and
- `employs` produces the set of `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")).hasDept().employs()     (0)
```

- `Div` is the table of all `Div` tuples;
- `select(name.equals("tool"))` eliminates `Div` tuples whose name is not `"tool"`;
- `hasDept()` produces the table of `Dep` tuples that are referenced by qualified `Div` tuples. In database parlance, this operation is a *right-semijoin* of qualified `Dep` tuples with the entire `Div` table [44, 49]; and
- `employs()` is another right-semijoin that produces the table of `Emp` tuples that work in qualified departments.

We could have written `(0)` using only relational algebra operations, making explicit the semijoin argument — here an association name — for each right-semijoin:

```
Div.select(name=`tool')
   .rightSemiJoin(hasDept).rightSemiJoin(employs)
```

This is ugly. However, by lifting association role names to their corresponding semijoin operations yields the compact and elegant expression `(0)`.

> **Note:** A bit of database sugaring was used in this example. `Job` is a many-to-many association between `Emp` and `Dep` (Fig. 2a). Classical relational database design, called *normalization*, replaces association `Job` with an association class `Job` and two one-to-many associations `JobEmp` and `JobDep` (Fig. 2b) [22, 44]. In the parlance of MDE, the normalization mapping of Fig. 2a→b is a model-to-model transformation.

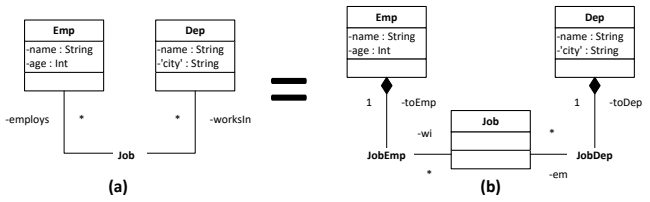

Figure 2: Database Normalization of the `Job` Association.

> **Note:** Association traversals in the unnormalized diagram (Fig. 2a) are implemented by cascading right-semijoins in Fig. 2b. Written as composed methods in Java:
>
> ```
> worksin() = wi().toDep()
> employs() = em().toEmp()
> ```
>
> *ie.,* `worksin()` is a traversal (read: right-semijoin) from `Emp` to `Dep` in Fig. 2a. In Fig. 2b, `worksin()` is a right-semijoin from `Emp` to `Job` via association `wi()` and then from `Job` to `Dep` via `toDep()`. Of course, these details can be hidden from end-users.

In a nutshell, ***the core 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 admittedly backwards in that the theory that inspired $\mathbb{A}_{ocl}$, and which existed long before $\mathbb{A}_{ocl}$ itself, should be presented next. As few practitioners in MDE appreciate category theory and far fewer allegories, the usual theory-then-implementation order is a obstacle for contemporary readability. For this reason, the important sections on relational algebra, category theory and the need for allegories are explained in Appendix A, which we hope in the future all will be able to read and appreciate.

2.2 Running Example

We add a recursive association, `Anc`, to our `EDD` diagram (see Fig. 3). Now each `Emp` has a lineage: descendants (children) and ancestors (parents). Traversing the `ANC` association computes, for example, `Emp`s that are grandparents, by expression `Emp.parOf().parOf()`, and `Emp`s that are grandchildren, by `Emp.chldOf().chldOf()`.

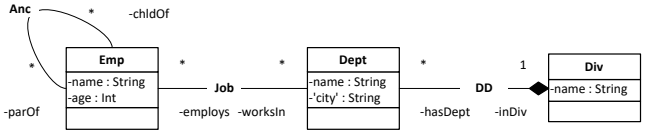

Figure 3: `EDD` with a Recursive, Lineage Association `ANC`.

Class Diagram to Relational Schema Mapping. It is well-known that `UML` class diagrams can be translated into normalized relational schemas [8, 22, 44]. The **`blue`** statements in Fig. 4 are `EDD` schema declarations in MDE_{lite} [8], the MDE platform used in this paper. There is a table for `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`s. The first column of every table is a manufactured identifier `id` required by MDE_{lite}.

> **Example.** 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`, where `city` values are quoted because they may have blanks (*eg.,* `"New York"`). 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 [8], such as Fig. 4. 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`, `parOf=p1`, and `chldOf=p3`, meaning `don` is the parent of `hanna`.

Although this example does not have class inheritance hierarchies, $\mathbb{A}_{ocl}$ supports subclasses/subtables as expected.

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

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

```
table(Anc,[id,parOf:Emp,chldOf: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:Dept]).
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.

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 workers 19 and older.
- **(C3)** No Div can employ more than 20 Emps.
- **(C4)** No more than two generations of workers from the same family can be employed.

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

- **(Q1)** Find employees whose name begins with "d" or "p".
- **(Q2)** Find the divisions that have departments in Austin.
- **(Q3)** List employees that work in multiple departments.
- **(Q4)** Find divisions in which don works.
- **(Q5)** Print the division colleagues of priscila.
- **(Q6)** List the ID of each Emp (whose parent is also an Emp) with the ID of division(s) in which he/she works.
- **(Q7)** Find the max number of employees in any department.

We consider queries in the next section and constraints afterwords.

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

An $\mathbb{A}_{ocl}$ program 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 $\mathbb{A}_{ocl}$ expressions for each query in Section 2.2. Outputs of **(Q1)**–**(Q7)** are posted in Fig. 5.

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

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

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

```
Emp.allInstances->select(name.at(1)='d' or
                         name.at(1)='p')
```

(Q2) finds divisions that have departments in Austin:

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

Departments that are not in "Austin" are removed by select(). Departments that remain are mapped by inDiv() to their divisions, and are then printed. Its USE OCL counterpart:

```
Dept.allInstances.select(city='Austin').inDiv->asSet
```

(Q3) lists employees that work in multiple departments:

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

The select() finds employees that work in more than 1 department. Its USE OCL counterpart:

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

(Q4) finds divisions in which don works:

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

The first line produces a table of Emp tuples with name="don". The table of Dep tuples in which don works is produced by worksIn(), and the table of divisions in which don works is produced by inDiv(), which is printed. Its USE OCL counterpart is:

```
Emp.allInstances->select(name='don').worksIn.inDiv
```

(Q5) prints the division colleagues of priscila:

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

The first line produces an Emp table of priscila tuples. worksIn().inDiv() converts this Emp table into a table of divisions in which priscila works. hasDept().employs() computes the table of Emps that work in those divisions. Its USE OCL counterpart is:

```
Emp.allInstances->select(name='priscila')
                 .worksIn.inDiv.hasDept.employs
```

(Q6) lists the ID of each employee (whose parent is an employee) with the ID of division(s) in which he/she works:

```
DTable Q6 = new DTable("Q6","EmpId","DivId");
edd.Emp.select(e->e.parOf().exists())
       .forEach(em->em.worksIn().inDiv()
              .forEach(d->Q6.add(em.id,d.id)));
Q6.print();
```

The first line creates a temporary table Q6 with column names "EmpId" and "DivId". The second line selects eligible Emps. The forEach lines compute Q6 tuples (ordered pairs). The last line prints table Q6. Its USE OCL counterpart is:[1]

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

[1] If an Emp works in multiple Divs, more tuples would be output.

(Q7) finds the max number of employees in any Dep:

```
OptionalInt m;    // a possibly null integer
m = edd.Dept.stream()
            .mapToInt(d->d.employs().count())
            .max();
if(m.isPresent())
   out.println("Ans: "+m.getAsInt());
```

This query uses existing Java stream operations. `edd.Dept.stream()` produces a stream of Dept tuples. The `mapToInt` replaces each Dep tuple with its Int number of employees. `max` returns the maximum of these numbers (but in the case that there are no Dep tuples, there could be no answer – so an `OptionalInt` is returned). The next line prints the value if there is one. Its USE OCL counterpart is:

```
Dept.allInstances.collect(employs->size)->max
```

Observations. Fig. 5 is the output of $\mathbb{A}_{ocl}$ and USE OCL. Their solutions are identical, albeit different syntax. In general, $\mathbb{A}_{ocl}$ and OCL expressions are similar. This is to be expected as both are stream processing languages.

Before we proceed, note that 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.

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) Find all divisions in which don works	

`table(Div,[id,name]).`
`Div(v1,clothing).`
`Div(v2,goods).`
	`Bag{v1,v2} : Bag(Div)`

(Q5) 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).`
	`Bag{p1,p2,p3,p6,p7} : Bag(Emp)`

(Q6) List the ID of each employee (whose parent is an employee) and the ID of division(s) in which he/she works	

`table(Q6,[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)) )`

(Q7) Find the max number of employees in any department	
`Ans: 4`	`4 : Integer`

Figure 5: $\mathbb{A}_{ocl}$ and USE OCL Solutions to Queries (Q1)–(Q7).

2.4 $\mathbb{A}_{ocl}$ Constraints

A special Java class and table operation are used in $\mathbb{A}_{ocl}$ 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. Outputs for constraints **(C1)**-**(C4)** are posted in Fig. 6.

$\mathbb{A}_{ocl}$ 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. Here are two ways to write **(C1)** that asserts all employees have unique names:

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

STRINGTable dups = edd.Emp.name().duplicates();
edd.Emp
   .select(e->dups.contains(e.name))
   .error(er,"Emp(%s..) has non-unique name=%s"
             r->r.id, r->r.name));
```

The first solution projects the `name` column of the `Emp` table into a single-column `STRINGTable` 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`.

The second solution computes a `STRINGTable` `dups` of duplicated names and uses `dups` to select `Emp` tuples that have a replicated name. The `ids` of these tuples and their replicated `names` are logged, giving a more detailed error report (see Fig. 6). Its USE OCL counterpart:

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

(C2) says every Dep in Toronto hires workers 19 and older:

```
edd.Dept
   .select(d->d.city.equals("Toronto"))
   .forEach(d->d.employs().select(ee->ee.age<19)
               .error(er,"%s illegally hired %s",
                      e->d.name,e->e.name));
```

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

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

(C3) says no division can employ more than 20 workers:

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

Its USE OCL counterpart:

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

(C4) says no more than two generations of workers from the same family can be employed:

```
String fmt = "%s has >2 generations " +
             "of family members employed";
db.Emp.select(e->e.chldOf().chldOf().exists())
      .error(er, fmt, e->e.name);
```

Its USE OCL counterpart:

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

A constraint program ends by printing accumulated errors:

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

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

Observations. Fig. 6 shows the output of $\mathbb{A}_{ocl}$ and USE OCL. The solutions are identical, although $\mathbb{A}_{ocl}$'s are more detailed. The constraints themselves are comparable in structure with $\mathbb{A}_{ocl}$ expressions a bit longer due to customized error logging.

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 Depts in Toronto cannot employ Emps younger than 19	
hardware illegally hired alex	**false**
(C3) No Div can employ more than 20 Emps	
	true
(C4) No more than two generations of workers can be employed	
steve has >2 generations of family members employed	**false**

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

2.5 Model-to-Model Transformations

OCL cannot update the model that it examines. By precluding updates, $\mathbb{A}_{ocl}$ could be the same. By allowing updates, $\mathbb{A}_{ocl}$ 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 [24]. The Java source for this program is in Fig. 7. We generated $\mathbb{A}_{ocl}$ packages for the Families and Persons metamodels, and the rest was easy. We do not foresee problems scaling $\mathbb{A}_{ocl}$ to large M2M transformations.

3 MDE$_{LITE}$ IMPLEMENTATION OF Meta4

MDE$_{lite}$ is a set of Java tools to explore the synergy of MDE and $\mathbb{RA}$ [6, 8]. It supports *model-to-text* (M2T), *model-to-model* (M2M), and (document-to-database parsing) *text-to-model* (T2M) transformations and has been bootstrapped. In MDE$_{lite}$, a metamodel is a database schema and a model is a database.

Meta4 is an MDE$_{lite}$ tool that generates $\mathbb{A}_{ocl}$ Java packages. In this section, we explore **Meta4**'s implementation.

3.1 Meta4 Front-End

Meta4 uses the graphical UML editor called Violet to draw umlCDs [48]. Each Violet umlCD is transformed into a CDSpec file where classes, associations, and inheritance relationships are declared textually. The CDSpec of the EDD umlCD is:

```
classDiagram EDD.

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

```
1  package atlexample;
2
3  import MDLUtilities.MarqueelIn_1Out;
4  import Allegory.Person.*;
5
6  public class ATLExample {
7      static Allegory.Family.Database in;
8      static Allegory.Person.Database out;
9
10     public static void main(String... args) {
11         // Step 1: standard marquee processing
12         MarqueelIn_1Out mark = new MarqueelIn_1Out(ATLExample.class,
13                 ".family.pl", ".person.pl", args);
14         String inputFileName = mark.getInputFileName();
15
16         // Step 2: read the families database and init output database
17         in = new Allegory.Family.Database(inputFileName);
18         out = new Allegory.Person.Database();
19
20         // Step 3: m2m transformation + print out
21         addBoys();
22         addFathers();
23         addGirls();
24         addMothers();
25         out.print();
26     }
27
28     static void addBoys() {
29         in.member
30           .select(m->m.sonOf!=null)
31           .forEach(m->{
32               String fullName = m.firstName+" "+m.sonOf.lastName;
33               out.male.add(new male(m.id,fullName)); });
34     }
35
36     static void addGirls() {
37         in.member
38           .select(m->m.daughterOf!=null)
39           .forEach(m->{
40               String fullName = m.firstName+" "+m.daughterOf.lastName;
41               out.female.add(new female(m.id,fullName)); });
42     }
43
44     static void addFathers() {
45         in.family.forEach(f->{
46             String fullName =  f.fatherid.firstName + " " +f.lastName;
47             if (out.male.select(m->m.fullName.equals(fullName)).size()==0)
48                 out.male.add(new male(f.fatherid.id, fullName));});
49     }
50
51     static void addMothers() {
52         in.family.forEach(f->{
53             String fullName =  f.motherid.firstName + " " +f.lastName;
54             if (out.female.select(m->m.fullName.equals(fullName)).size()==0)
55                 out.female.add(new female(f.motherid.id, fullName));});
56     }
57 }
```

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

```
assoc Emp employs NM -- Dept worksIn NM.
assoc Emp parOf NM -- Emp chldOf NM.
assoc Div inDiv BLACK_DIAMOND -- Dept hasDept NM.

// no inheritance decls in this example
```

Meta4 gives its users the option to draw a graphical umlCD using Violet (and have it translated to a CDSpec) or to write the CDSpec directly to produce an allegory package.

3.2 Meta4 Back-End

Here is where the heavy-lifting occurs. **Meta4** generates:

- An MDE$_{lite}$ schema for the input CDSpec. This schema was shown in blue in Fig. 4. The tuples in Fig. 4 were not generated; they were added later by hand.
- A Database.java file whose constructor reads a database text file (such as EDD of Fig. 4) and populates an in-memory database of its tables.
- A τ.java file is produced for each tuple type τ. It defines each field of a tuple and the required set of operations on τ-tuples, like employs() for Dep tuples.

  Example. (C2) uses the expression:

```
forEach(d->d.employs()...
```

d is a tuple of type Dep. The expression d.employs() computes the right semijoin of table {d} with table edd.Emp.

- A τTable.java file is produced for each τ.java file. It maintains a list of all τ-tuples and has the required set of operations on τTables, like right semijoins.

  Example. (C3) uses the expression:

```
d.hasDept().employs()...
```

  The input to employs() is a Dep table and the output is an Emp table. The method employs() performs a right semijoin on its input Dep table with the edd.Emp table.

- A Lang.java file that contains a set of static class definitions, including ErrorReport, and ρ.java and ρTable.java for each primitive datatype ρ of $\mathbb{A}_{\mathbb{ODL}}$, where $\rho \in$ {INT, STRING, FLOAT, DOUBLE, BOOL}.

3.3 A Tour of Dep.java

Fundamentally, Dep.java is no different than other τ.java files and is, for that matter, only marginally different from DepTable.java and other τTable.java files. It begins with the expected header:

```
public class Dept extends ... {
   public String id;
   public String name;
   public String city;
   public Div    inDiv;
```

which defines a Dep tuple and each of its columns. The Dep constructor initializes primitive fields while a helper method assigns a Java object to its inDiv field.[2]

Dep.java has a tuple toString() method:

```
static String fmt ="Dept(%s,%s,`%s',%s).\n";

public String toString() {
 return String.format(fmt,id,name,city,inDiv.id);
}
```

And a method for each semijoin operation. Here is inDiv():

```
public DivTable inDiv() {
   return new DivTable(db).add(this.inDiv);
}
```

Each Dep tuple points to the Div tuple to which it is associated. If d is a tuple of Dep, {d} is the Dep table that contains this tuple. The expression inDiv({d}) = {d.inDiv} yields a Div table that contains the lone tuple d.inDiv. (d.inDiv is the tuple pointed at by tuple d by field inDiv.) The Java expression newDivTable(db) creates an empty Div table and add(this.inDiv) adds the requisite tuple.

The right semijoin method employs() is more complicated as it is a semijoin over an association class:

```
public EmpTable employs() {
   EmpTable result = new EmpTable(db);
   for(job j : db.Job.tuples()) {
      if(j.worksIn==this &&
         !result.contains(j.employs)) {
        result.add(j.employs);
      }
   }
   return result;
}
```

An empty Emp table is assigned to result. We need to know which Job tuples point to this Dep tuple. Consider Job tuple j. If j points to this (*ie.,* j.worksIn==this), then j.employs is an employee of this Dep. The remaining code ensures that the resulting Emp table contains no duplicate Emp tuples.

As said earlier, the generated code for other τ.java and τTable.java files for $\tau \in$ {Job, Dept, Anc, Div} are marginally different.

3.4 The $\mathbb{A}_{\mathbb{ODL}}$ Challenge

The M2T generation of $\mathbb{A}_{\mathbb{ODL}}$ code is straightforward. The challenge in crafting τ.java and τTable.java files is to reduce redundancy. We presented in the last sections the τ-unique code segments of τ.java files (and τTable.java files). These files have a lot in common.

Consider the non-optimized files Div.java and Emp.java of Fig. 8. Their bodies are almost syntactically identical:

```
public abstract class Div {
  String id;
  protected Database db;

  public void setDB(Database db) {
     this.db = db;
  }

  public void print(PrintStream out) {
     out.print(this.toString());
  }
  ...
}
```
(a)

```
public abstract class Emp {
  String id;
  protected Database db;

  public void setDB(Database db) {
     this.db = db;
  }

  public void print(PrintStream out) {
     out.print(this.toString());
  }
  ...
}
```
(b)

Figure 8: Original Div.java and Emp.java.

The standard way to eliminate such redundancy is to pull-up members that are shared by τ.java classes into a single class, here called TuPle, that becomes the superclass of τ.java classes like Div and Emp. This works except that every allegory package has its own Database class. This means that TuPle must be a Java generic, with Database as a parameter. Emp and Div become subclasses of TuPle<Allegory.EDD.Database>, as do all other τ.java files in an allegory package, Fig. 9.

```
public abstract class TuPle<Database> {
  String id;
  protected Database db;

  public void setDB(Database db) { this.db = db; }

  public void prints(String x) { System.out.print(x); }
  ...
}
```

Figure 9: TuPle superclass of Div.java and Emp.java.

Now examine the DivTable.java and EmpTable.java files in Fig. 10 which is digitally enlargeable.

We spot a Database reference, so we know the generic-parameter-trick above to handle it. There are, however, syntactic differences: Two terms are highlighted in red: τ the name of the tuple class and τTable the name of its table class. Unfortunately, it is not a simple matter of adding two more generic parameters to have <Database,T,TBL>, where T is the tuple class and TBL is the table class. Why? Because the table class is itself parameterized by T. This makes creating a useable Java multi-parameter generic class challenging, as it pushes the capabilities of Java Generics.

[2] A database is populated with tuples in two passes. Primitive data values for each tuple are loaded on the first pass. Tuple pointers, such as fields created by association roles, are assigned in the second phase, after a Java object per tuple has been created.

```
public class DivTable {
  Database db;

  protected LinkedList<Div> tuples
     = new LinkedList<>();

  public abstract LinkedList<Div> tuples();

  public Stream<Div> stream() {
        return tuples().stream();
  }

  public DivTable unique() {
     DivTable unique = new DivTable();
     for (Div t : tuples()) {
        if (!unique.contains(t))
           unique.add(t);
     }
  }

  public DivTable select(Predicate<Div> p) {
     DivTable filtered = new DivTable();
     tuples().stream().filter(p)
       .forEach(t -> filtered.add(t));
     return filtered;
  } ...
}
```

(a)

```
public class EmpTable {
  Database db;

  protected LinkedList<Emp> tuples
     = new LinkedList<>();

  public abstract LinkedList<Emp> tuples();

  public Stream<Emp> stream() {
        return tuples().stream();
  }

  public EmpTable unique() {
     EmpTable unique = new EmpTable();
     for (Emp t : tuples()) {
        if (!unique.contains(t))
           unique.add(t);
     }
  }

  public EmpTable select(Predicate<Emp> p) {
     EmpTable filtered = new EmpTable();
     tuples().stream().filter(p)
       .forEach(t -> filtered.add(t));
     return filtered;
  } ...
}
```

(b)

Figure 10: Original `DivTable.java` and `EmpTable.java` .

Fig. 11 is our solution. `TaBle` is parameterized by classes `Database`, `T`, and `TBL` where `T` must extend `TuPle<Database>` and `TBL` must extend `TaBle<Database,T,TBL>`, which is decidedly non-obvious.

```
public abstract class TaBle<Database,
                            T extends TuPle<Database>,
                            TBL extends TaBle<Database,T,TBL>> {

 protected Database db;

 protected LinkedList<T> tuples = new LinkedList<>();

 public abstract LinkedList<T> tuples();

 public Stream<T> stream() { return tuples().stream(); }

 public TBL unique() {
   TBL unique = New();
   for (T t : tuples()) {
     if (!unique.contains(t))
       unique.add(t);
   }
 }

 public abstract TBL New();
 ...
}
```

Figure 11: Generic `TaBle` superclass of `τTable.java` files.

Java adds nits of its own. For example, locate the black pointers (⬅) in Fig. 10. One cannot invoke "`new TBL()`" in a Java generic. Instead, an indirection is used: an abstract `New` method is defined in `TaBle`; its concrete method is defined in each `τTable.java` file:

```
τTable New() { return new τTable(); }
```

The black pointers in Fig. 11 show this indirection. This is a case where having both generics and `M2T` capabilities were critical, as both complement each other.

3.5 Table Class Hierarchies

Suppose class `Dog` is a subclass of `Pet`. It is well-documented in Java that `List<Dog>` is *not* a subclass of `List<Pet>` [30].

Yet, it makes perfect sense (to us anyways) that `Table<Dog>` is a subclass or subtable of `Table<Pet>`. Every `Dog` tuple of `Table<Dog>` is a tuple of `Table<Pet>`. All associations of `Pet` become semijoin methods in `Table<Pet>`. These same associations are inherited by `Dog`, and therefore should be semijoin methods of table `Table<Dog>`.

Every `τ.java` and `τTable.java` begin as:

```
public class τ extends TuPle<Database> {..}
public class τTable extends TaBle<Database,τ,τTable> {..}
```

which means we are forbidden to write:

```
public class DogTable extends PetTable {..}
```

as it would require multiple inheritance. The solution is to emulate table inheritance by delegation [16].

3.6 $\mathbb{A}_{ocl}$ Statistics

Given a `umlCD` with `t` tables and `k` association declarations that encode `n:m` associations and require association tables, **Meta4** produces `2·(t+k)+3` files. For each `τ`, there is a `τ.java` file and a `τTable.java` file. The remaining files are `Lang.java`, `Database.java`, and an `MDElite` schema file. Empirically we found each table file is approximately 100 Java LOC and each tuple file is 170 Java LOC. The remaining files total 1330 Java LOC. The **Meta4** framework that these files plug-in is 2800 Java LOC.

The codebase of **Meta4** consists of a set of `M2T` tools and parsers, totaling 6100 Java LOC. `MDElite`, the `MDE` platform on which **Meta4** was built, is 18K Java LOC.

4 EVALUATION

We said in Section 1 that this is an idea paper. We assert any `MDE` application that we have written with `MDElite`, or with Eclipse `MDE` for that matter, we could have used $\mathbb{A}_{ocl}$. We explained how our prototype worked and believe it could be replicated a couple months on any `MDE` platform. What is important at this stage are assessments of the potential of $\mathbb{A}_{ocl}$ in `MDE` tooling.

We pose the following research questions for this evaluation. The number of each question is the following sub-section where it is addressed:

4.1 What is wrong with `OCL`?
4.2 Why use a general-purpose programming language, as opposed to specialized languages for `MDE`?
4.3 `OCL` is operation extensible; so too is $\mathbb{A}_{ocl}$. What is the difference?
4.4 A discipline of education is judged by the quality of its teaching material. What does $\mathbb{A}_{ocl}$ offer?
4.5 What are reasons for not adopting $\mathbb{A}_{ocl}$?

4.1 What is wrong with `OCL`?

Cabot and Gogolla have a thorough on-line tutorial about `OCL` [12] and say it best:

- There is no serious use of UML without `OCL`!!!, and
- You may not like it but right now there is nothing better than `OCL`!!

Such statements have been encouraging to us ☺. The point here is that such sentiments are not isolated [23, 29, 40, 47].

`OCL` does indeed have many good points. At its core is a stream-based language — which is exactly the same reason that makes $\mathbb{A}_{ocl}$ powerful. Although the syntax of `OCL` and $\mathbb{A}_{ocl}$ are slightly different, their core language constructs align.

If you agree, $\mathbb{A}_{ocl}$ may be for you.

4.2 The Value of General-Purpose Languages

`MDE` has opened the market and software engineering technologies to domain-specific languages, with a focus on metamodels and models (be they graphical or not). But does `MDE` really need to

use a special-purpose programming and constraint languages like OCL, ATL, and QVT that require their own compiler and IDE-like infrastructure, when a standard and richer infrastructure that Java 11 provides might suffice?

Maintaining a specialized language, its 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 $\mathbb{A}_{\mathbb{OCL}}$ even more appealing.

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

If you agree, $\mathbb{A}_{\mathbb{OCL}}$ may be for you.

4.3 The Value of Extensibility

OCL is now operation extensible; it requires a heavy-weight solution to be back-compatible with earlier heavy-weight OCL platforms [53].

The need for extensibility reflects a common experience of the MDE community where models arise whose constraints are not expressible in OCL. Example: a model with matrices may require them to be non-singular (invertible). No MDE user wants to recode singularity checks or other standard domain operations. For Java, there is a host of matrix packages that could be imported as-is. So it is for other domains. Adding such operations to OCL or $\mathbb{A}_{\mathbb{OCL}}$ is ideal.

At the time of this writing, $\mathbb{A}_{\mathbb{OCL}}$ did not have database groupBy operation. Adding such a capability would, we thought, be a good test of extensibility. The following generic groupBy function was designed, added to TaBle, and tested within a half hour. It takes a table paramater TBL and partitions its tuples according to a grouper function into subtables. An action is then applied to each subtable:

```
public void groupBy(Function<T,String> grouper,
                    Consumer<TBL> action) {

  // 1: partition `this' tbl by grouper value
  HashMap<String, TBL> gb = new HashMap<>();
  for (T t : tuples()) {
    String key = grouper.apply(t);
    TBL tl = gb.get(key);
    if (tl == null) {
      tl = New();      // create empty
      gb.put(key, tl); // subtable for key
    }
    tl.add(t);
  }

  // 2: foreach grouped table...
  for (TBL pt : gb.values()) {
    action.accept(pt);
  }
}
```

A use of groupBy partitions the Dep table into subtables by city and counting the number of departments in each city:

```
db.Dept.groupBy(t->t.city, tbl->displayAgg(tbl));

//... helper function ...
static void displayAgg(DeptTable t) {
    Dept d = t.getFirst(); // get 1st tuple of t
    out.format("%10s has %d Dept(s)\n",
        d.city, t.size());
}

//... result output ...
Hamilton has 1 Depts(s)
  Austin has 2 Depts(s)
 Toronto has 2 Depts(s)
```

As it was added to the **Meta4** framework TaBle class, all $\mathbb{A}_{\mathbb{OCL}}$ tables now have the groupBy operation.

Extensibility can be more invasive. Additional operations might need to be added to τ.java and τTable.java files in **Meta4**'s M2T transformations. These tasks are straightforward in MDE$_{\text{lite}}$, as they should be for all MDE platforms.

If you agree, $\mathbb{A}_{\mathbb{OCL}}$ may be for you.

4.4 What is the Educational and Scientific Value of $\mathbb{A}_{\mathbb{OCL}}$?

"Eating your own dog-food" means using your own products internally. Bootstrapping MDE tools is a classic example. $\mathbb{A}_{\mathbb{OCL}}$ is yet another as it illustrates how theoretical foundations that underlie MDE, namely $\mathbb{CT}$ and $\mathbb{RA}$, can be integrated to address one of its long-time concerns – tooling. It is a novel case-study of how MDE theory might improve MDE practice. And it underscores the belief of many (not all) in MDE about the importance of theory in MDE development.

New research possibilities arise. It is now 40 years since the first practical relational query optimizer was built [41]. There is no reason why the algebra underlying $\mathbb{A}_{\mathbb{OCL}}$ could not be optimized using similar technology. Already there is research on how blocks of pure-Java code that process tuples could be abstracted into SQL statements to achieve greater execution performance [15]. There is every reason to believe such work could be replicated for $\mathbb{A}_{\mathbb{OCL}}$.

If you agree, $\mathbb{A}_{\mathbb{OCL}}$ may be for you.

4.5 What are reasons for not adopting $\mathbb{A}_{\mathbb{OCL}}$?

OCL expressions need to be analyzed or translated to other languages (*eg.*, CSP) [12, 28]. Wouldn't a specialized Java compiler be needed to do this? And if so, wouldn't a large infrastructure be needed, returning us back to square one?

Maybe not. A standard trick is to use an existing Java compiler to parse and typecheck a program. At which point, custom tools can redirect compiler execution to walk Java *Abstract Syntax Trees* (ASTs) to find and extract information about $\mathbb{A}_{\mathbb{OCL}}$ expressions to be subsequently manipulated (see [33] as an example). Again, by leveraging existing Java infrastructure, such tasks won't be easy, but not terrible.

If you agree, $\mathbb{A}_{\mathbb{OCL}}$ may be for you.

5 RELATED WORK

Embedding database queries in Java and other languages is common today [36]. Cheny, Lindley, and Wadler proposed quoting mechanisms for Java to enclose SQL-like queries [14]. Cook and Wiedermann 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 [15]. $\mathbb{A}_{\mathbb{OCL}}$ is

an even closer integration where Java packages express database (or $\mathbb{RA}$) computations.

There are general tools, such as Xtend and Xbase, that integrate DSLs (such as OCL constructs) with Java and other languages [54]. Of course, these tools are necessarily heavier-weight than $\mathbb{A}_{ocl}$ as the "$\mathbb{A}_{ocl}$" DSL is simply a package requiring no language engineering at all.

Another approach implements OCL as a Java package (interpreter) [19]. 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. $\mathbb{A}_{ocl}$ effectively removes this middleware approach to express OCL-like queries natively, invoking methods of an $\mathbb{A}_{ocl}$-generated package for direct execution.

Several projects have translated OCL into Java [31, 43]. These particular projects were completed before Java 8 (2014) was released, where streams and lambda functions first appeared. The translations to Java, as one can imagine without Java 8, are verbose and not as elegant as their OCL and $\mathbb{A}_{ocl}$ counterparts.

Yue and Ali performed a user study to compare OCL and Java when writing constraints [55]. Java 7 was used, meaning that the Java code was (as above) more verbose than that of 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. $\mathbb{A}_{ocl}$ should reduce this advantage. A goal of [55] was to "find a way to ... offset the investment in terms of training and tool support ... for OCL". $\mathbb{A}_{ocl}$ does not eliminate this cost, but reduces it to learning a Java package, which is less intimidating.

Rumpe proposed ≪Java≫OCL to (a) adjust OCL syntax closer to that of Java to make it more familiar to Java developers [40]. He examined the OCL meta operations (*eg.*, 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 [55]: to offset the investment in OCL training.

And finally, Vaziri and Jackson [47] argue that a language like Alloy would be more appropriate than OCL to express constraints, as OCL is "too implementation oriented". We believe $\mathbb{A}_{ocl}$ is a step in the right direction. It is unclear how we can completely avoid using general purpose programming languages for MDE software development (see Section 4.3).

6 CONCLUSIONS

We answered a controversial but inevitable question of 'When can general-purpose programming languages replace the specialized languages in MDE?' It is controversial because of enormous investments that have supported *Object Management Group* (OMG) standards, such as OCL. We are uninterested in the politics of this question — although politics may be the ultimate decider. We foresee long-term benefits in a rethinking of early technical decisions in MDE and potential long-term penalties by not doing so. (This is a 14-year-old refrain of Favre [23]). Never-the-less, replacements for many of the first-generation MDE tools will eventually be considered. We presented $\mathbb{A}_{ocl}$ – a lightweight, elegant, and pure-Java alternative to OCL – for consideration.

We explained the theory and inspiration behind $\mathbb{A}_{ocl}$, showed typical $\mathbb{A}_{ocl}$ queries and constraints are syntactically similar (with comparable complexity) to OCL, presented enough implementation detail for others to reproduce $\mathbb{A}_{ocl}$ on their own MDE platforms, and argued in favor of using today's general-purpose languages to replace today's MDE languages.

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 infrastructure (all of which must be maintained) with small Java libraries. We argued that maintaining these 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. 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. We are also indebted to Maider Azanza and Arantza Irastorza (University of Basque Country, Spain) for their helpful comments and insights on OCL.

REFERENCES

[1] S. Apel, D. Batory, C. Kaestner, and G. Saake. *Feature-oriented software product lines: concepts and implementation.* Springer, 2013.
[2] C. Avila, A. Sarcar, Y. Cheon, and C. Yeep. Runtime constraint checking approaches for ocl, a critical comparison. In *SEKE*, 2010.
[3] K. Baclawski, D. Simovici, and W. White. A categorical approach to database semantics. *Mathematical Structures in Computer Science*, 4(2), 1994.
[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, Sept. 2005.
[5] D. Batory. Using Modern Mathematics as an FOSD Modeling Langauge. In *GPCE*, Oct. 2008.
[6] D. Batory. *Automated Software Design: Volume 1.* submitted – available on request, 2019.
[7] D. Batory, M. Azanza, and J. Saraiva. The Objects and Arrows of Computational Design. In *MODELS*, Oct. 2008.
[8] D. Batory, E. Latimer, and M. Azanza. Teaching Model Driven Engineering from a Relational Database Perspective. In *MODELS*, 2013.
[9] H. Bauerdick, M. Gogolla, and F. Gutsche. Detecting ocl traps in the uml 2.0 superstructure: An experience report. In *UML*, 2004.
[10] A. D. Brucker, T. Clark, C. Dania, G. Georg, M. Gogolla, F. Jouault, E. Teniente, and B. Wolff. Panel discussion: Proposals for improving ocl. In *Proceedings of the MODELS 2014 OCL Workshop (OCL 2014)*, 2014.
[11] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 1991.
[12] J. Cabot and M. Gogolla. Object constraint language: A definitive guide. https://www.slideshare.net/jcabot/ocl-tutorial.
[13] J. Cadavid, B. Baudry, and B. Combemale. Empirical evaluation of the conjunct use of mof and ocl. In *Proceedings of EESSMOD workshop at MODELS'11*, 10 2011.
[14] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1976.
[15] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, 2013.
[16] W. R. Cook and B. Wiedermann. Remote batch invocation for SQL databases. In *DBPL*, 2011.
[17] B. I. Corporation. Using the tie mechanism. https://www.albany.edu/dept/csi/csi518/fall03/inprise/vbroker/doc/books/vbj/vbj45/programmers-guide/tie.html.
[18] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature models are views on ontologies. In *SPLC*, 2006.
[19] Z. Diskin and T. S. E. Maibaum. Category theory and model-driven engineering: From formal semantics to design patterns and beyond. In *ACCAT*, 2012.
[20] Eclipse-Kepler-Documentation. Evaluating constraints and queries. https://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FPivotEvaluatingConstraints.html&cp=38_6_5, 2013.

[21] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. *Fundam. Inf.*, Jan. 2006.
[22] H. Eichelberger and K. Schmid. Mapping the design-space of textual variability modeling languages: A refined analysis. *Int. J. Softw. Tools Technol. Transf.*, Oct. 2015.
[23] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 1999.
[24] J.-M. Favre. Towards a Basic Theory to Model Model Driven Engineering. In *Workshop on Software Model Engineering, WISME 2004*, 2004.
[25] Families to persons. https://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf, 2007.
[26] P. J. Freyd and A. Scedrov. *Categories, Allegories*. Elsevier Science Publishers, 1990.
[27] J. Fuentes, V. Quintana, J. Llorens, G. Genova, and R. Prieto-Diaz. Errors in the uml metamodel? *ACM SIGSOFT Software Engineering Notes*, 28:3, 11 2003.
[28] R. C. Gonçalves, D. Batory, J. L. Sobral, and T. L. Riché. From software extensions to product lines of dataflow programs. *Software and Systems Modeling (SoSyM)*, Dec. 2017.
[29] C. A. González, F. Büttner, R. Clarisó, and J. Cabot. Emftocsp: A tool for the lightweight verification of emf models. In *FormSERA*, June 2012.
[30] H. Husmann and S. Zschaler. The object constraint language for uml 2.0 an overview and assessment. 2004.
[31] Java generics, inheritance, and subtypes. https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html.
[32] S. Kallel and et al. Automatic translation of ocl meta-level constraints into java meta-programs. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015*, 2016.
[33] G. Karsai, A. Ledeczi, S. Neema, and J. Sztipanovits. The MIC Toolsuite: Metaprogrammable Tools for Embedded Control System Design. In *IEEE CCACSD*, Oct. 2006.
[34] J. Kim, D. Batory, and D. Dig. X15: A tool for refactoring java software product lines. In *SPLC*, 2017.
[35] E. Lippe and A. H. Ter Hofstede. A category theory approach to conceptual data modeling. *RAIRO-Theoretical Informatics and Applications*, 1996.
[36] M. Mabrok and M. Ryan. Category theory as a formal mathematical foundation for model-based systems engineering. *Applied Mathematics and Information Sciences*, 2015.
[37] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *SIGMOD*, 2006.
[38] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
[39] R. Rosebrugh and R. Wood. Relational databases and indexed categories. In *Proceedings of the International Category Theory Meeting*, 1991.
[40] B. Rossiter, D. Nelson, and M. A. Heather. *The Categorical Product Data Model as a Formalism for Object-Relational Databases*. University of Newcastle upon Tyne. Computing Laboratory, 1995.
[41] B. Rumpe. 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, 2002.
[42] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD*, 1979.
[43] A. J. Shidqie. Compilation of ocl into java for the eclipse ocl implementation. https://www.semanticscholar.org/paper/Compilation-of-OCL-into-Java-for-the-Eclipse-OCL-Shidqie/89ae7c8a840df6cefc35dcb871eabf34ba3e6029, 2007.
[44] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.
[45] D. Spivak. Categorical databases. http://categoricaldata.net/fql/oracle.pdf, 2014.
[46] Use:uml-based specification environment. https://sourceforge.net/projects/useocl/, 2019.
[47] M. Vaziri and D. Jackson. Some shortcomings of ocl, the object constraint language of uml. In *TOOLS*, 2000.
[48] Violet UML Editor. http://alexdp.free.fr/violetumleditor/page.php.
[49] Relational algebra. https://en.wikipedia.org/wiki/Relational_algebra, 2017.
[50] Span (category theory). https://en.wikipedia.org/wiki/Span_(category_theory), 2017.
[51] Sql:2016. https://en.wikipedia.org/wiki/SQL:2016, 2018.
[52] 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*, 01 2011.
[53] E. D. Willink. An extensible ocl virtual machine and code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL, 2012.
[54] Xtext: Language engineering for everyone! https://www.eclipse.org/Xtext/index.html, 2017.
[55] T. Yue and S. Ali. Empirically evaluating ocl and java for specifying constraints on uml models. *Software & Systems Modeling*, Jul 2016.
[56] B. Zieliński, P. Maślanka, and Ś. Sobieski. Allegories for database modeling. In *International Conference on Model and Data Engineering*, 2013.

A CATEGORY THEORY, ALLEGORIES, AND RELATIONAL ALGEBRA

Category Theory ($\mathbb{CT}$) is a theory of functions, mathematical structures, and their relationships [37]. $\mathbb{CT}$ is foundational to diverse software paradigms such as dataflow architectures, software product lines, and model driven engineering [5, 7, 27].

This section presents a gentle introduction to $\mathbb{CT}$, showing that standard $\mathbb{CT}$ encodings of class diagrams are inappropriate for an `OCL` replacement, and explaining why generalizing to allegories and *relational algebra* ($\mathbb{RA}$) is needed.

A.1 Basic Category Theory ($\mathbb{CT}$)

A *category* is a directed multigraph[3] whose nodes are domains[4] and edges are arrows [37]. A *domain* is a set of elements. Fig. 12 is a *category diagram* with the `Emp`, `Dep`, and `Div` domains. `Emp` is the domain of employees, `Dep` is the domain of departments in which employees work, and `Div` is the domain of divisions to which departments belong.

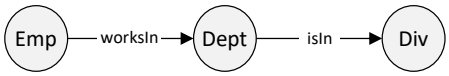

Figure 12: A Category Diagram.

An *arrow* $\theta : D \rightarrow C$ is a total function that pairs each element in domain `D` with an element in co-domain `C`. Fig. 12 has two arrows: $\texttt{worksIn} : \texttt{Emp} \rightarrow \texttt{Dep}$ and $\texttt{isIn} : \texttt{Dep} \rightarrow \texttt{Div}$.

Let $\alpha : A \rightarrow B$, $\beta : B \rightarrow C$, and $\gamma : C \rightarrow D$ be arrows with domains $\{A, B, C, D\}$ not necessarily distinct [37]. $\mathbb{CT}$ has three axioms: Arrows compose (1) and arrow composition is associative (2). Both should be familiar to readers as they are axioms of function composition:

$$(\beta \cdot \alpha) : A \rightarrow C \tag{1}$$

$$(\gamma \cdot \beta) \cdot \alpha \;=\; \gamma \cdot (\beta \cdot \alpha) \tag{2}$$

In addition, every domain `A` has an *identity arrow* $\texttt{I}_A : A \rightarrow A$ that pairs each element $a \in A$ to itself. For any arrow $\alpha : A \rightarrow B$, axiom (3) requires:

$$\texttt{I}_B \cdot \alpha = \alpha \quad \wedge \quad \alpha \cdot \texttt{I}_A = \alpha \tag{3}$$

Identity arrows are implied (not drawn) in our category diagrams.

Lastly, the *product* $A \times B$ of two domains, `A` and `B`, is the set of all ordered pairs `[a,b]` where $a \in A$ and $b \in B$. Each product is defined with projection arrows $\pi_A : A \times B \rightarrow A$ and $\pi_B : A \times B \rightarrow B$ to extract elements of a pair. The product of three or more domains and their projection arrows are logical generalizations.

Example. An `Emp` class is shown in Fig. 13a and Fig. 13b is its category diagram. `Emp` is the product $\texttt{String} \times \texttt{Int} \times \texttt{String}$ with three projection arrows $\texttt{education} : \texttt{Emp} \rightarrow \texttt{String}$, $\texttt{name} : \texttt{Emp} \rightarrow \texttt{String}$, and $\texttt{age} : \texttt{Emp} \rightarrow \texttt{Int}$. Given an `Emp` instance `e`, the `name`, `age` and `education` values of `e` are computed by the expressions `name(e)`, `age(e)` and `education(e)`.

[3] A *multigraph* allows any number of edges between two nodes.
[4] Domains are called *objects* in $\mathbb{CT}$. We use 'domains' to avoid the obvious confusion in the context of Java and other OO languages.

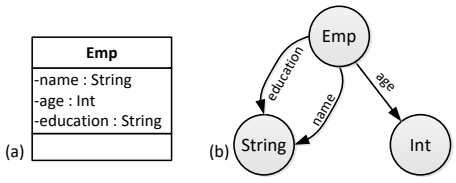

Figure 13: Domain `Emp` as `String`×`Int`×`String`.

Henceforth whenever you encounter a category diagram, take note the computations that it expresses. The category in Fig. 13b computes `education(e)`, `name(e)` and `age(e)` ∀`e`∈`Emp`.

A.2 Metamodels and Models

A *metamodel* is a *UML Class Diagram* (`umlCD`) plus constraints. We focus on `umlCD`s now and consider constraints later in Section 2.2.

Fig. 14a is a `umlCD` and Fig. 14b is a model or instance (*aka.*, object diagram). Fig. 14a states that there are two domains: `Dep` is the domain of departments and `Emp` is the domain of employees, where each `Emp` `worksIn` precisely one `Dep` and the employees of a `Dep` are found via `hasEmps`.

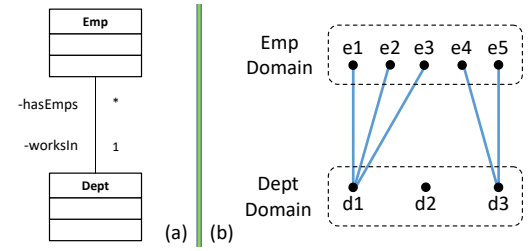

Figure 14: A Class Diagram and an Object Diagram.

Two points: First, `Emp` has two distinct meanings: (1) the domain of *all* employees, which is infinite, and (2) the domain of employees for a model, which is finite. The sizes of the `Emp` and `Dep` domains in the model of Fig. 14b are finite, 5 and 3 respectively. `MDE` tools evaluate constraints on models and not on infinite-sized domains. We use the `MDE` interpretation.

Second, each domain instance is assigned a (manufactured) identifier. In Fig. 14b, employees and departments have no explicit attributes, but they *do* have identifiers: `e1...e5` are identifiers for `Emps` and `d1...d3` are identifiers for `Deps`.

A.3 Category Encodings of Class Diagrams

There are many ways to encode `umlCD`s as categories [3, 11, 34, 38, 39, 45, 56]. The most well-known pairs each class with a domain of its instances and each primitive data type with a domain of its values. Fig. 15a shows a `umlCD` and Fig. 15b is its category diagram. Associations with *:1 cardinalities, like $\texttt{Emp}\xrightarrow[\texttt{worksIn}]{*\quad 1}\texttt{Dep}$, are encoded by arrow `worksIn:Emp→Dep`. In contrast, a `*:(0..1)` association is harder to represent. Association $\texttt{D}\xrightarrow[\texttt{J}]{*\quad 0..1}\texttt{C}$ says each instance of `D` is *optionally* paired with an instance of `C` by role `J`. Arrow (total function) `J:D→C` can be used only if domain `C` were enlarged by `null` to indicate an absence of a pairing for some objects `d`∈`D`.

Associations without a `1` or `0..1` end cardinality can not be directly encoded [8]. Instead, `umlCD`s with such associations are *normalized* by the rewrite/refactoring of Fig. 16, which replaces a single `n:m` association with a pair of `1:n` and `1:m` associations and an association class `AB` [44, 50]:

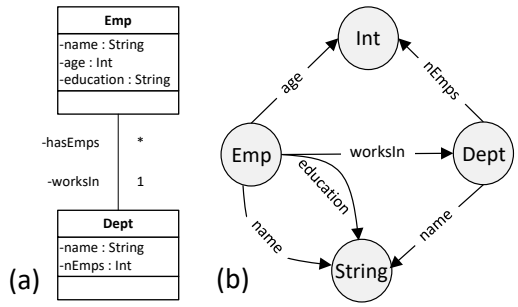

Figure 15: A Class Diagram and its Category Diagram.

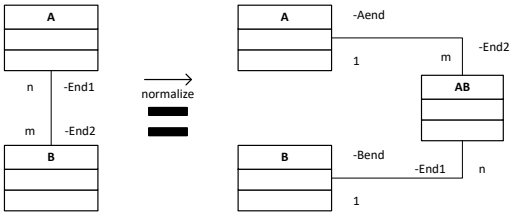

Figure 16: The Association Normalization Refactoring.

To express computations and traversals of `umlCD`s, we translate `umlCD`s to database schemas and their instances to databases — an ancient tradition in database modeling [42, 44]. Doing so allows us to think about relational tables and apply $\mathbb{RA}$ operations to these tables.[5] Essentially each class of a `umlCD` becomes a table and its instances are table rows.

Although the above encoding of `umlCD`-as-categories is both general and common, it has *serious* computational drawbacks:

- How is association `hasEmps` of Fig. 15a computed? All arrows are functions that map a domain instance to a co-domain instance. `hasEmps` maps a `Dep` instance to a *set* of `Emp` instances. So given a `Dep` instance, how does one compute its table of `Emp` instances using the category of Fig. 15b? Answer: you can't. ***There are computations (association traversals) that cannot be expressed.***
- How are tables of employees and departments represented? And operations on tables? At best, tables are expressed indirectly (or as we saw in the last bullet, maybe not at all).

In short, standard encodings of `umlCD`s as categories are inappropriate for our needs because essential `OCL` computations are inexpressible.

A.4 Allegories = $\mathbb{CT}$ + Power Set Domains

Now consider an encoding that removes the difficulties of the last section. Recall that `MDE` domains are of finite size. Let `D` be a domain (read: finite table) of tuples and |`D`| be its cardinality. By $2^{\texttt{D}}$ we denote the *powerset* of `D` — the set of all unique $2^{|\texttt{D}|}$ subtables of `D`.

Given a category diagram, replace each domain `D` with powerset $2^{\texttt{D}}$. That is, $2^{\texttt{D}}$ is the domain that has the empty table ∅, every *singleton table* `{d}` for each tuple `d`∈`D`, every table with distinct pairs `{d1,d2}` `d1`≠`d2` of tuples in `D`, and so on, including `D` itself.

[5] And finally, rather than using manufactured identifiers for tuples, relational databases use tuple fields to form tuple identifiers [44]. In doing so, arrows of a category denote functional dependencies [56]. We use tuples with manufactured identifiers without loss of generality.

Let's reinterpret category diagram Fig. 15b in this manner. An arrow $A : 2^D \to 2^C$ maps a *table* of D tuples to a *table* of C tuples. So arrow $\texttt{worksIn} : 2^{\texttt{Emp}} \to 2^{\texttt{Dep}}$ maps an Emp table to the table of Deps in which these Emps work.

Example. Fig. 14b shows pairings (associations) of Dep and Emp tuples. e1 is a tuple of Emp. The singleton table in $2^{\texttt{Emp}}$ containing e1 is {e1}. So worksIn({e1}) = {d1}.

Even better, the reverse or *dual* of worksIn is $\texttt{hasEmps} : 2^{\texttt{Dep}} \to 2^{\texttt{Emp}}$, *which can be computed.*

Example. d1 is a tuple of Dep and {d1} is its singleton table in $2^{\texttt{Dep}}$. Thus hasEmps({d1}) = {e1,e2,e3}, the table of employees that work in d1.

We know how to map singleton tables via an association — we use the association pairings of individual tuples in an instance diagram. We lift this to compute the mappings for all other tables of a power set. Let table $t \in 2^D$ and arrow $\alpha : 2^D \to 2^C$:

$$\alpha(t) = \bigcup_{d \in t} \alpha(\{d\}) \tag{4}$$

$\alpha(t)$ is the union of the tables formed by applying α to the singleton tables of all tuples of t.

Example. Table $\{e1,e4,e5\} \in 2^{\texttt{Emp}}$. Then worksIn({e1,e4,e5}) ={d1,d3} from Fig. 14b and (4).

Example. Table $\{d1,d3\} \in 2^{\texttt{Dep}}$. Then hasEmps({d1,d3}) = {e1,e2,e3,e4,e5} = Emp from Fig. 14b and (4).

Allegories are categories with power set domains [25, 56].

A.5 Relational Algebra ($\mathbb{RA}$)

Allegories admit any functions on tables. Consider the functions select (σ_p), semijoin ($\rtimes_j$), and sort (τ_o):

Select. Select $\sigma_p : 2^R \to 2^R$ eliminates tuples from table $r \in 2^R$ that do not satisfy predicate p. The result is subtable $r' \subseteq r$.

Example. Let p=(name="alex" and age>20) and table $e \in 2^{\texttt{Emp}}$. The expression $\sigma_p(e)$ produces subtable $e' \subseteq e \ \wedge \ e' \in 2^{\texttt{Emp}}$ that contains only the tuples of e whose name attribute equals "alex" and whose age attribute >20.

Semijoin. Readers may be familiar with the relation join operation $r \bowtie_j s$. It takes the cross product of tables r and s and applies the selection predicate j to eliminate unwanted tuples. A *right semijoin*, denoted $r \rtimes_j s$, is 'half' a join, yielding the subtable $s' \subseteq s$ whose tuples join with r tuples on predicate j [44, 49].

Consider an association $X \overset{B}{\underset{A}{\text{———}}} Y$ with arbitrary cardinalities. Every arrow $A : 2^X \to 2^Y$ in an allegory is a right semijoin (*eg.*, $X \rtimes_A Y$). And so too is its dual, $B : 2^Y \to 2^X$. *This means we can traverse an association both in the forward AND reverse directions in an allegory.* Recall Emp={e1...e5} and Dep={d1,d2,d3}:

Example. $\texttt{worksIn(Emp)=(Emp} \rtimes_{\texttt{worksIn}} \texttt{Dep)=\{d1,d3\}}$.

Example. $\texttt{hasEmps(Dep)=(Dep} \rtimes_{\texttt{hasEmps}} \texttt{Emp)=Emp}$.

Sort. Sort $\tau_o : 2^R \to 2^R$ sorts an input table $r \in 2^R$ on field o in ascending (+o) or descending (–o) order.[6]

[6] sort(r) rearranges but does not modify tuples of table r. For this reason, sort is not a $\mathbb{RA}$ operation because tuple ordering is not expressible. However, sorting ***is*** a core operation on tables in SQL [51].

Other $\mathbb{RA}$ operations, like projection (π) and join ($\bowtie$), are also possible.