\textbf{\textsc{Aocl}}: 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

\begin{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 on relational algebra. A\textsubscript{ocl} can replace OCL: it can be used to write OCL-like constraints and model transformations in Java. The theoretical foundation for A\textsubscript{ocl} is allegories, a unification of category theory and relational algebra. A simple MDE tool generates an A\textsubscript{ocl} Java plug-in from an input class diagram.
\end{abstract}

\section{INTRODUCTION}
A central issue in \textit{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 \textit{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 \textit{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., \textit{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 \textit{Object Oriented (00)} 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 00 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 \textit{allegories} [25]. In 2013, Ziebański, Mašlan-ka, 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 A\textsubscript{ocl}, is pure-Java and is implemented by a Java framework that relies on Java streams, generics, and lambda expressions. Using A\textsubscript{ocl} to write and evaluate model constraints requires an MDE tool to generate the metamodel’s A\textsubscript{ocl} Java plug-in for this framework. A\textsubscript{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 A\textsubscript{ocl} tool and framework is \textasciitilde{}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 A\textsubscript{ocl};
- Illustrating A\textsubscript{ocl} queries and constraints;
- Describing how the A\textsubscript{ocl} plug-in generator was built;
- Explaining the potential of A\textsubscript{ocl} in future MDE tooling; and
- Listing open problems for the MDE community to explore.

\section{A\textsubscript{ocl}}

\subsection{Insights Behind A\textsubscript{ocl}}

Fig. 1 is a UML class diagram. It says there are \textit{Employees}, \textit{Departments}, and \textit{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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{emp-dep-div-class-diagram.png}
\caption{The Emp-Dep-Div (EDD) Class Diagram.}
\end{figure}

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

\texttt{\textbf{SELECT Emp.\textit{name}, Emp.\textit{age} FROM Emp WHERE Emp.\textit{department} = \{Dep.\textit{name} FROM Dep WHERE Dep.\textit{division} = Div\textit{name} AND Dep.\textit{tool} = \textit{true}\}}}
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 OOl style/syntax:

\[
\text{Div.select(name.equals("tool")).hasDept.employs}(0)
\]

- `Div` is the table of all `Div` tuples;
- `select(name.equals("tool")).hasDept.employs` produces 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 (8) using only relational algebra operations, making explicit the semijoin argument — here an association name — for each right-semijoin:

\[
\text{Div.select(name='tool').rightSemijoin(hasDept).rightSemijoin(employs)}
\]

This is ugly. However, by lifting association role names to their expression making explicit the semijoin argument — here an association name — for each right-semijoin:

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

**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 `JobEmp` 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()`;
- `employs()` 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 OOl syntax with customized names for right-semijoins**. We call this language `A_{oo}`.

**Foundations and Concessions.** Our presentation is admittedly backwards in that the theory that inspired `A_{oo}`, and which existed long before `A_{oo}` 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()`. In a nutshell, `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()`.

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

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 MDElite [8], the MDE platform used in this paper. There is a table for `Emp` and `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:n` ancestry information among `Emp`s. The first column of every table is a manufactured identifier `id` required by MDElite.

**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 (e.g., "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 an `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`.

**Example.** Although this example does not have class inheritance hierarchies, `A_{oo}` supports subclasses/subtables as expected.

---

Don Batory and Najd Altoyan
We can immediately write Aedl 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 Aedl code to compute the solution to (Q1):

```java
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:

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

(Q2) finds divisions that have departments in Austin:

```java
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:

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

(Q3) lists employees that work in multiple departments:

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

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

```java
Emp.allInstances.select(workins->size>1)
```

(Q4) finds divisions in which don works:

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

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

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

(Q5) prints the division colleagues of priscila:

```java
edd.Emp.select(t->t.name.equals("priscila")).workIn().hasDept().employ().print();
```

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

```java
Emp.allInstances->select(name='priscila').workIn.inDiv.hasDept.employ
```

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

```java
DTable Q6 = new DTable("Q6","EmpId","DivId");
edd.Emp.select(e->e.parOf().exists()).forEach(em->Q6.add(em.id,em.div.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` computes Q6 tuples (ordered pairs). The last line prints table Q6. Its USE OCL counterpart is:

```java
Emp.allInstances->select(parOf->notEmpty).iterate(e->Emp;
edd.Set(Tuple{Firste,second: Set<Div}}>Set());
ed->including(Tuple{
Firste,
second=e.workIn.inDiv->asSet});
```

1If an Emp works in multiple Divs, more tuples would be output.
(Q7) finds the max number of employees in any Dep:

```java
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 Dept tuple with its 1st number of employees, max returns the maximum of these numbers (but in the case that there are no Dept 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:

```oclo
Department d; // a possibly null department
Department m = collection.stream()
   .map(d->d.size())
   .max();
if (m != null) out.println("Ans: "+m); // USE OCL
```

Observations. Fig. 5 is the output of `Aed` and USE OCL. Their solutions are identical, albeit different syntax. In general, `Aed` 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 Java stream operations. `edd.Dept.stream()` produces a stream of Dept tuples. The `mapToInt` replaces each Dept tuple with its 1st number of employees, max returns the maximum of these numbers (but in the case that there are no Dept 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:

```oclo
Department d; // a possibly null department
Department m = collection.stream()
   .map(d->d.size())
   .max();
if (m != null) out.println("Ans: "+m); // USE OCL
```

Figure 5: `Aed` and USE OCL Solutions to Queries (Q1)–(Q7).

<table>
<thead>
<tr>
<th>Aed Solutions</th>
<th>USE OCL Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Q1</code> Find all employees whose name begins with 'd' or 'p'</td>
<td><code>Q1</code> Find all employees whose name begins with 'd' or 'p'</td>
</tr>
<tr>
<td><code>table(Emp, [id, name, age: int])</code></td>
<td><code>table(Emp, [id, name, age: int])</code></td>
</tr>
<tr>
<td><code>Set{p1, p6, p9} : Set(Emp)</code></td>
<td><code>Set{p1, p6, p9} : Set(Emp)</code></td>
</tr>
<tr>
<td><code>Q2</code> Find all employees whose name begins with 'd' or 'p'</td>
<td><code>Q2</code> Find all employees whose name begins with 'd' or 'p'</td>
</tr>
<tr>
<td><code>table(Div, [id, name])</code></td>
<td><code>table(Div, [id, name])</code></td>
</tr>
<tr>
<td><code>Set{v1} : Set(Div)</code></td>
<td><code>Set{v1} : Set(Div)</code></td>
</tr>
<tr>
<td><code>Q3</code> List employees that work in multiple departments</td>
<td><code>Q3</code> List employees that work in multiple departments</td>
</tr>
<tr>
<td><code>table(Emp, [id, name, age: int])</code></td>
<td><code>table(Emp, [id, name, age: int])</code></td>
</tr>
<tr>
<td><code>Set{p1} : Set(Emp)</code></td>
<td><code>Set{p1} : Set(Emp)</code></td>
</tr>
<tr>
<td><code>Q4</code> Find all employees whose name begins with 'd' or 'p'</td>
<td><code>Q4</code> Find all employees whose name begins with 'd' or 'p'</td>
</tr>
<tr>
<td><code>table(Div, [id, name])</code></td>
<td><code>table(Div, [id, name])</code></td>
</tr>
<tr>
<td><code>Bag{v1, v2} : Bag(Div)</code></td>
<td><code>Bag{v1, v2} : Bag(Div)</code></td>
</tr>
<tr>
<td><code>Q5</code> Find unique employees</td>
<td><code>Q5</code> Find unique employees</td>
</tr>
<tr>
<td><code>table(Emp, [id, name, age: int])</code></td>
<td><code>table(Emp, [id, name, age: int])</code></td>
</tr>
<tr>
<td><code>Set{p1, p2, p3, p6, p7} : Set(Emp)</code></td>
<td><code>Set{p1, p2, p3, p6, p7} : Set(Emp)</code></td>
</tr>
<tr>
<td><code>Q6</code> Find all employees whose name begins with 'd' or 'p'</td>
<td><code>Q6</code> Find all employees whose name begins with 'd' or 'p'</td>
</tr>
<tr>
<td><code>table(Div, [id, name])</code></td>
<td><code>table(Div, [id, name])</code></td>
</tr>
<tr>
<td><code>Bag{v1, v2} : Bag(Div)</code></td>
<td><code>Bag{v1, v2} : Bag(Div)</code></td>
</tr>
<tr>
<td><code>Q7</code> Find the max number of employees in any department</td>
<td><code>Q7</code> Find the max number of employees in any department</td>
</tr>
<tr>
<td><code>Ans: 4</code></td>
<td><code>Ans: 4</code></td>
</tr>
</tbody>
</table>

2.4 Aocl Constraints

A special Java class and table operation are used in Aed to log constraint violations. `ErrorReport` is a Java class whose stateful objects log errors. Table method `error(…)` 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. `Aed` constraint programs begin with the reading of a database and the creation of an `ErrorReport` object:

```java
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:

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

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` dupes of duplicated names and uses dupes 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:

```oclo
class Env {
  _tuple: Tuple()
   Table t1;
}
context Emp inv UniqueName:
   Env env = new Env();
   forAll(e1, e2 | e1.name = e2.name implies e1=e2)
```

Figure 6: Aocl Solutions USE OCL Solutions

Figure 7: USE OCL Solutions

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

```oclo
context Emp inv UnderAge:
   self.select(city="Toronto")
   .forAll(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
```

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

```oclo
context Emp inv UnderAge:
   self.select(city="Toronto")
   .forAll(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
```

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

```oclo
def Dept.
   .select(d | d.city.equals("Toronto"))
   .forEach(d | d.employs().select(e | e.age<19))
   .forEach(d | d.employs().select(e | e.age<19))
   .forEach(d | d.employs().select(e | e.age<19))
```

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

```oclo
context Emp inv UnderAge:
   self.select(city="Toronto")
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
```

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

```oclo
def Dept.
   .select(d | d.city.equals("Toronto"))
   .forEach(d | d.employs().select(e | e.age<19))
   .forEach(d | d.employs().select(e | e.age<19))
   .forEach(d | d.employs().select(e | e.age<19))
```

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

```oclo
context Emp inv UnderAge:
   self.select(city="Toronto")
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
   .forEach(e | e.age<19)
```

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

```oclo
def Div.
   .select(d | d.hasDept().employs().count()<=20)
   .error(="%s has more than 20 workers", d->d.name)
```

Its USE OCL counterpart:

```oclo
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:

```oclo
String fmt = \"%s has more than 20 workers\";
Db.Emp.select(e | e.childOf().childOf().exists())
   .error(="%s has more than 20 workers", e->e.name)
```

Its USE OCL counterpart:

```oclo
context Div inv EmpCount:
   self.hasDept().employs().size()<=20
```
A constraint program ends by printing accumulated errors:

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

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

**Observations.** Fig. 6 shows the output of Aocl and USE OCL. The solutions are identical, although Aocl’s are more detailed. The constraints themselves are comparable in structure with Aocl expressions a bit longer due to customized error logging.

<table>
<thead>
<tr>
<th>Aocl Solutions</th>
<th>USE OCL Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(C1) All Emps must have unique names</strong>&lt;br&gt;Solution 1: multiple employees have name=hanna&lt;br&gt;Solution 2: Emp(p3..) has non-unique name=hanna&lt;br&gt;Emp(p7..) has non-unique name=hanna</td>
<td>false</td>
</tr>
<tr>
<td><strong>(C2) All Depts in Toronto cannot employ Emps younger than 19</strong>&lt;br&gt;hardware illegally hired alex</td>
<td>false</td>
</tr>
<tr>
<td><strong>(C3) No Div can employ more than 20 Emps</strong>&lt;br&gt;true</td>
<td></td>
</tr>
<tr>
<td><strong>(C4) No more than two generations of workers can be employed</strong>&lt;br&gt;steve has &gt;2 generations of family members employed</td>
<td>false</td>
</tr>
</tbody>
</table>

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, Aocl could be the same. By allowing updates, Aocl 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 Aocl packages for the Families and Persons metamodels, and the rest was easy. We do not foresee problems scaling Aocl to large M2M transformations.

### 3 MDElite Implementation of Meta4

[1] MDElite is a set of Java tools to explore the synergy of MDE and 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 MDElite, a metamodel is a database schema and a model is a database.

Meta4 is an MDElite tool that generates Aocl 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:

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

Figure 7: Aocl Families-to-Persons M2M Transformation.
d is a tuple of type Dep. The expression d.employs() computes the right semijoin of table (d) with table emp.

- A TTable.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 TTables, like right semijoins.

**Example. (C3)** uses the expression:

```java
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 tdd.Emp table.

- A Lang.java file that contains a set of static class definitions, including ErrorReport, and ρ.java and TTable.java for each primitive datatype ρ of A. where ρ∈{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 TTable.java files. It begins with the expected header:

```java
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.

Dep.java has a tuple toString() method:

```java
static String fmt = "%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():

```java
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 to 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:

```java
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 (i.e., 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 TTable-.java files for r∈{Job, Dept, Anc, Div} are marginally different.

### 3.4 The A Challenge

The M2T generation of A code is straightforward. The challenge in crafting τ.java and TTable.java files is to reduce redundancy. We presented in the last sections the τ-unique code segments of τ.java files (and TTable.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:

```java
public abstract class Div {
    String id;
    protected Database db;
    public void setDB(Database db) {
        this.db = db;
    }
    public void print(PrintStream out) {
        out.println(this.toString());
    }
    ...
}
```

```java
public abstract class Emp {
    String id;
    protected Database db;
    public void setDB(Database db) {
        this.db = db;
    }
    public void print(PrintStream out) {
        out.println(this.toString());
    }
    ...
}
```

**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.

```java
public abstract class TuPle<Database> {
    String id;
    protected Database db;
    public void setDB(Database db) {
        this.db = db;
    }
    public void print(PrintStream out) {
        System.out.println(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 TTable 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.

---

---

2A 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.
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 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<Dog>.

Every T, Java and TTable.java begin as:

```java
public class T extends TuPle<Database>, TBL extends TaBle<Database,T,TBL>> {
protected Database db;
protected LinkedList<Tuple<Database>> tuples();
public abstract Stream<T> stream();
}
```

which means we are forbidden to write:

```java
public class DogTable extends PetTable (..)
```
as it would require multiple inheritance. The solution is to emulate table inheritance by delegation [16].

3.6 Aocl Statistics

Given a umlCD with t tables and k association declarations that encode n:m associations and require association tables, M2T produces 2·(t+k)+3 files. For each τ, there is a τ.java file and a TTable-.java file. The remaining files are Lang.java, Database.java, 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 M2T framework that these files plug-in is 2800 Java LOC.

The codebase of MDE consists of a set of M2T tools and parsers, totaling 6100 Java LOC. MDElite, the MDE platform on which M2T 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 Aocl. 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 Aocl 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 Aocl. What is the difference?

4.4 A discipline of education is judged by the quality of its teaching material. What does Aocl offer?

4.5 What are reasons for not adopting Aocl?

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!!!
- 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 Aocl powerful. Although the syntax of OCL and Aocl are slightly different, their core language constructs align.

If you agree, Aocl 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 (they be 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 A_etc 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 Metaseq, 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, A_etc 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 [33].

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 A_etc is ideal.

At the time of this writing, A_etc 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 parameter TBL and partitions its tuples according to a grouper function into subtables. An action is then applied to each subtable:

```java
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 tbl = gb.get(key);
        if (tbl == null) {
            tbl = New();  // create empty
            gb.put(key, tbl); // subtable for key
        }
        tbl.add(t);
    }
    // 2: foreach grouped table ...
    for (TBL pt : gb.values()) {
        action.accept(pt);
    }
}
```

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

```java
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)
", 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 Metaseq framework TaBle class, all A_etc tables now have the groupBy operation.

Extensibility can be more invasive. Additional operations might need to be added to T.java and TTable.java files in Metaseq’s M2T transformations. These tasks are straightforward in MDE1to0, as they should be for all MDE platforms.

If you agree, A_etc may be for you.

4.4 What is the Educational and Scientific Value of A_etc?

“Eating your own dog-food” means using your own products internally. Bootstrapping MDE tools is a classic example. A_etc is yet another as it illustrates how theoretical foundations that underlie MDE, namely CT and 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 A_etc 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 A_etc.

If you agree, A_etc may be for you.

4.5 What are reasons for not adopting A_etc?

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 A_etc 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, A_etc 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]. A_etc is
an even closer integration where Java packages express database (or 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 A_{ocl} as the “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. A_{ocl} effectively removes this middleware approach to express OCL-like queries natively, invoking methods of an 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 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. Nevertheless, the authors found that participants working with OCL and Java performed equally well, with an edge to OCL when constraints became complicated. 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”. A_{ocl} does not eliminate this cost, but reduces it to learning a Java package, which is less intimidating.

Rumpel proposed <Java взгляд на 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., OcllAny, Ocl1Type, Ocl1Expression, 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 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 A_{ocl} — a lightweight, elegant, and pure-Java alternative to OCL — for consideration.

We explained the theory and inspiration behind A_{ocl} showed typical A_{ocl} queries and constraints are syntactically similar (with comparable complexity) to OCL, presented enough implementation detail for others to reproduce 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 Altayon are supported by NSF grant CCF1212683. Altayon is also supported by King Abdullah 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

A CATEGORY THEORY, ALLEGORIES, AND RELATIONAL ALGEBRA

Category Theory (CT) is a theory of functions, mathematical structures, and their relationships [37]. 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 CT, showing that standard CT encodings of class diagrams are inappropriate for an OCL replacement, and explaining why generalizing to allegories and relational algebra (RA) is needed.

A.1 Basic Category Theory (CT)

A category is a directed multigraph\(^2\) whose nodes are domains\(^3\) and edges are arrows [37]. A domain \(A\) 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 division of departements to which departments belong.

\[
\begin{align*}
\text{Emp} & \xrightarrow{\text{worksIn}} \text{Dep} \xrightarrow{\text{isIn}} \text{Div} \\
\end{align*}
\]

Figure 12: A Category Diagram.

An arrow \(\alpha : A \rightarrow C\) is a total function that pairs each element in domain \(A\) with an element in co-domain \(C\). Fig. 12 has two arrows: \(\text{worksIn} : \text{Emp} \rightarrow \text{Dep}\) and \(\text{isIn} : \text{Dep} \rightarrow \text{Div}\).

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]. 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:

\[
\begin{align*}
(\beta \circ \alpha) : A \rightarrow C \\
(\gamma \circ (\beta \circ \alpha)) = (\gamma \circ \beta) \circ \alpha
\end{align*}
\]

In addition, every domain \(A\) has an identity arrow \(1_A : A \rightarrow A\) that pairs each element \(a \in A\) to itself. For any arrow \(\alpha : A \rightarrow B\), axiom (3) requires:

\[
1_B \cdot \alpha = \alpha \land \alpha \cdot 1_A = \alpha
\]

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 \(\text{String} \times \text{Int} \times \text{String}\) with three projection arrows: \(\text{Emp} \rightarrow \text{String}\), name : \(\text{Emp} \rightarrow \text{String}\), and age : \(\text{Emp} \rightarrow \text{Int}\). Given an Emp instance \(e\), the name, age and education values of \(e\) are computed by the expressions \(\text{name}(e), \text{age}(e)\) and \(\text{education}(e)\).

\(^2\)A multigraph allows any number of edges between two nodes.

\(^3\)Domains are called objects in CT. We use ‘domains’ to avoid the obvious confusion in the context of Java and other OO languages.
A.2 Metamodels and Models

A metamodel is a UML Class Diagram (\texttt{uml}CD) plus constraints. We focus on \texttt{uml}CDs now and consider constraints later in Section 2.2.

Fig. 14a is a \texttt{uml}CD 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.

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. \texttt{MDE} tools evaluate constraints on models and not on infinite-sized domains. We use the \texttt{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 \texttt{uml}CDs 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 \texttt{uml}CD and Fig. 15b is its category diagram. Associations with \texttt{x:1} cardinalities, like Emp\(\rightarrow\)Dep, are encoded by arrow \texttt{worksIn:Emp\rightarrowDep}. In contrast, a \texttt{x:0..1} association is harder to represent. Association \texttt{D\rightarrow\texttt{set}} of \texttt{D} says each instance of \texttt{D} is \texttt{optionally} paired with an instance of \texttt{C} by role \texttt{J}. Arrow (total function) \texttt{J:D\rightarrowC} can be used only if domain \texttt{C} were enlarged by \texttt{null1} to indicate an absence of a pairing for some objects of \texttt{D}.

Associations without a 1 or 0..1 end cardinality cannot be directly encoded [8]. Instead, \texttt{uml}CDs with such associations are normalized by the rewrite/refactoring of Fig. 16, which replaces a single \texttt{n:m} association with a pair of \texttt{1:n} and \texttt{1:m} associations and an association class \texttt{AB} [44, 50]:

---

Figure 13: Domain Emp as String\(\times\)Int\(\times\)String.

Henceforth whenever you encounter a category diagram, take note the computations that it expresses. The category in Fig. 15b computes \texttt{education(e), name(e) and age(e) \forall e\in\texttt{Emp}}.

To express computations and traversals of \texttt{uml}CDs, we translate \texttt{uml}CDs 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 RA operations to these tables. Essentially each class of a \texttt{uml}CD becomes a table and its instances are table rows.

Although the above encoding of \texttt{uml}CDs-as-categories is both general and common, it has serious computational drawbacks:

- How is association \texttt{hasEmps} of Fig. 15a computed? All arrows are functions that map a domain instance to a co-domain instance. \texttt{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 \texttt{uml}CDs as categories are inappropriate for our needs because essential \texttt{OCL} computations are inexpressible.

A.4 Allegories = C\(\mathcal{T}\) + Power Set Domains

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

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

---

\(^1\text{And finally, rather than using manufactured identifiers for tuples, relational databases use tuple fields to form tuple identifiers [44]. In doing so, arrows of a category denote functional dependencies [56]. We use tuples with manufactured identifiers without loss of generality.}\)
Let’s reinterpret category diagram Fig. 15b in this manner. An arrow \( A : 2^0 \rightarrow 2^t \) maps a table of \( 0 \) tuples to a table of \( t \) tuples. So arrow \( \text{worksIn}: 2^\text{Emp} \rightarrow 2^\text{Dep} \) maps an \( \text{Emp} \) table to the table of \( \text{Dep} \) in which these \( \text{Emp} \)s work.

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

Even better, the reverse or dual of \( \text{worksIn} \) is \( \text{hasEmps}: 2^\text{Dep} \rightarrow 2^\text{Emp} \), which can be computed.

**Example.** \( d1 \) is a tuple of \( \text{Dep} \) and \( \{d1\} \) is its singleton table in \( 2^\text{Dep} \). Thus \( \text{hasEmps}(\{d1\}) = \{e1, e2, e3\} \), 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 \( t \in 2^0 \) and arrow \( \alpha: 2^0 \rightarrow 2^t \):

\[
\alpha(t) = \bigcup \{ \alpha(d) \mid d \in t \} \tag{4}
\]

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

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

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

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

### A.5 Relational Algebra (RA)

Allegories admit any functions on tables. Consider the functions select \( (\sigma) \), semijoin \( (\bowtie) \), and sort \( (\pi) \):

- **Select.** Select \( \sigma_p : 2^r \rightarrow 2^r \) eliminates tuples from table \( r \in 2^r \) that do not satisfy predicate \( p \). The result is subtable \( r' \subseteq r \).

  **Example.** Let \( p = \{ \text{name} = "\text{alex}" \text{ and age} > 20 \} \) and table \( e \in 2^\text{Emp} \). The expression \( \sigma_p(e) \) produces subtable \( e' \subseteq e \) 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 \( r \bowtie s \). It takes the cross product of tables \( r \) and \( s \) and applies the selection predicate \( j \) to eliminate unwanted tuples. A right semijoin, denoted \( r \bowtie s \), is ‘half’ a join, yielding the subtable \( s' \subseteq s \) whose tuples join with \( r \) tuples on predicate \( j \) [44, 49].

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

  **Example.** \( \text{worksIn}(\text{Emp}) = \text{Emp} \bowtie \text{worksIn} \bowtie \text{Dep} = \{d1, d3\} \).

  **Example.** \( \text{hasEmps}(\text{Dep}) = \text{Dep} \bowtie \text{hasEmps} \bowtie \text{Emp} = \text{Emp} \).

- **Sort.** Sort \( \pi_r : 2^r \rightarrow 2^r \) 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 RA operation because tuple ordering is not expressible. However, sorting is a core operation on tables in SQL [51].

Other RA operations, like projection \( (\pi) \) and join \( (\bowtie) \), are also possible.