Noname manuscript No.
(will be inserted by the editor)

Teaching Model Driven Engineering from a Relational Database Perspective

Don Batory · **Maider Azanza**

the date of receipt and acceptance should be inserted later

Abstract We reinterpret MDE from the viewpoint of relational databases to provide an alternative way to understand, demonstrate, and teach MDE using concepts and technologies that should be familiar to undergraduates. We use (1) relational database schemas to express metamodels, (2) relational databases to express models, (3) Prolog to express constraints and M2M transformations, (4) Java tools to implement M2T and T2M transformations, and (5) Java to execute transformations. Application case studies and a user study illuminate the viability and benefits of our approach.

1 Introduction

Model Driven Engineering (MDE) is an approach to software development that uses models to specify complex systems at multiple levels of abstraction. MDE transformations map these models to other models, including source code, documentation, and inputs to external tools, for purposes of software construction and analysis.

We believe it is essential to expose undergraduates to MDE concepts so that they will have an appreciation for MDE when they encounter them in industry. Specifically these concepts are: models, metamodels, model constraints, *model-to-model (M2M)* transformations, *model-to-text (M2T)* transformations, and *tool-to-model (T2M)* transformations. Our motivation is from experience: unless students encounter an idea (however immature) in school, they are less likely to embrace it in the future. *Further,*

Don Batory
Department of Computer Science
University of Texas at Austin
Austin, Texas, USA
E-mail: batory@cs.utexas.edu

Maider Azanza
University of the Basque Country (UPV/EHU)
San Sebastian, Spain
E-mail: maider.azanza@ehu.es

teaching MDE is intimately related, if not inseparable, to the tools and languages that make MDE ideas concrete.

Our initial attempt to do this in Fall 2011 was a failure. We used the Eclipse Modeling Tools[1] and spent quite some time creating videos for students to watch on installation and tool usage. Despite our efforts, installation for students was a problem. A version of Eclipse was eventually posted that had all tools installed. The results were no better when students used the tools. A simple assignment was given to draw a metamodel for state diagrams (largely something presented in class) using Eclipse, let Eclipse generate a tool for drawing state diagrams, and to use the generated tool to draw particular state diagrams. This turned into a very frustrating experience for most students. 25% of our upper-division undergraduate class got it right; 50% had mediocre submissions, and the remaining just gave up. Another week was given (with tutorial help) to allow 80% to 'get it right', but that still left too many behind. The whole experience left a bitter taste for us, and worse, our students. *We do not know if this is a typical situation or an aberration, but we vowed not to repeat it again.*

In retrospect we found many reasons. In a nutshell, Eclipse MDE tools were the culprit.

1. The tools were unappealing—they were difficult to use even for simple applications.
2. The tools fostered a medieval mentality in students to use incantations to solve problems. Point here, click that, something happens. From a student's perspective, this is gibberish. Although we could tell them what was happening, this mode of interaction leaves a vacuum where a deep understanding should reside.[2]
3. With benefit of years of hindsight, we concluded that the entry cost of using, teaching, and understanding these tools was too high for our comfort.

We sought an alternative and light-weight way to understand and demonstrate MDE, *leveraging tools and concepts undergraduates should already know.*

In this paper, we present an evolutionary rather than revolutionary approach to understand and teach core MDE concepts (models, metamodels, M2M, M2T, T2M transformations, constraints). We tried this approach – which we henceforth call *MDELite* – with a new class of undergraduates in Fall 2012 and every year since, and experienced many fewer problems. This paper extends our initial work on MDELite at MODELS'13 [9]. The contributions of that paper were:

- a description of the database/Prolog/Velocity technology behind MDELite that we now use to teach MDE concepts;
- interesting applications to illustrate the viability of MDELite;

Here we round out that work by presenting an evaluation of MDELite consisting of two quasi-experiments:

- an informal cohort study with undergraduate students in Fall 2011 (Eclipse MDE tools) and Fall 2012 (MDELite) to evaluate its benefits and limitations, and

[1] Specifically EMT, Graphical Modeling Tooling Framework Plug-in, OCL Tools Plug-in, and Eugenia for Eclipse 3.6.2.

[2] Admittedly, this statement holds for IDEs in general, and Eclipse in particular.

– a quasi-experiment with graduate students in Fall 2013 to evaluate student perception of the tools.

We report that students largely preferred MDELite over Eclipse MDE tools. And finally,

– we summarize our (now) multi-year perspective using MDELite with directions on future work.

2 MDE Models and MetaModels

MDE can be understood in terms of relational databases. Although MDE is usually presented in terms of graphs (as visual representations of models or metamodels), all graphs have simple encodings as a set of normalized tables.

Consider a metamodel for *finite state machines (FSMs)* in Figure 1a, consisting of nodes and edges. The schema for the underlying relational tables (using manufactured identifiers, denoted by `node#` and `edge#`) is shown in Figure 1b. A particular FSM populates these tables with tuples. The FSM of the first author's eating habits and its tuples are given in Figure 1c-d.

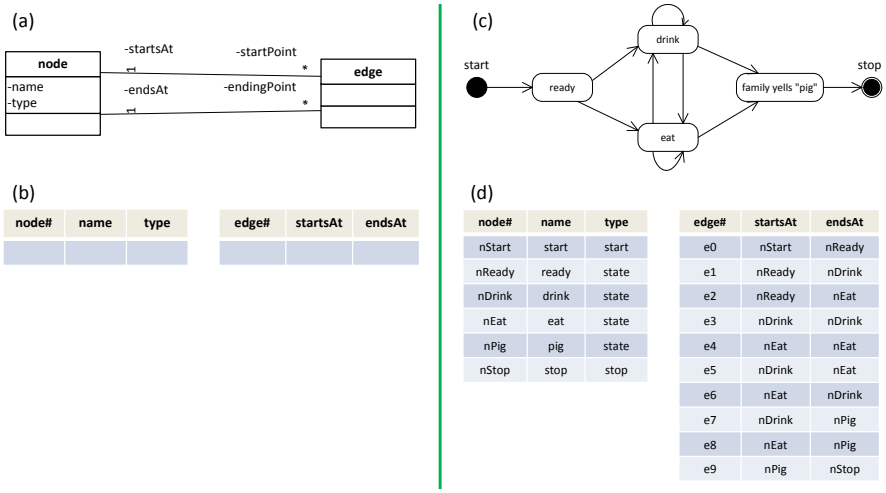


Fig. 1 A State Machine and its Tables.

Manufactured tuple identifiers eliminate virtually all of the complexities of relational table design, e.g., functional dependencies and forming compound primary keys [17,23]. There are only five simple rules to map metamodels to table definitions and one rule for tuple instantiation:

1. Every metaclass maps to a distinct table. If a metaclass has `k` attributes, the table will contain *at least* $1+k$ columns: one for a manufactured identifier and one for each attribute.

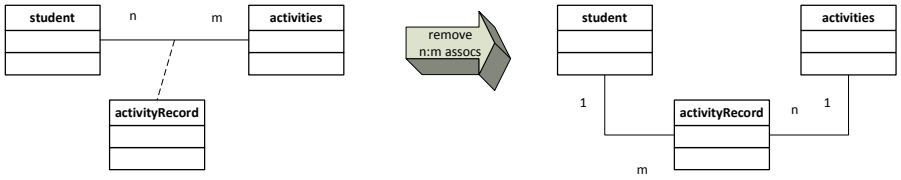


Fig. 2 Transformation That Removes `n : m` Associations.

2. `n : m` associations are valid in metamodels [36], but not in normalized tables. Every association must have one end with a `0..1` or `1` cardinality. Figure 2 shows how `n : m` associations are transformed into a pair of `1 : n` and `1 : m` associations with an explicit association class. The reason for this is the next rule.
3. Each association is represented by a single attribute on the '`0 : 1`' or '`1`' side of the association. Usually an association adds an attribute to both tables that it relates. The '`n`' side would have a set-valued attribute which is disallowed in normalized tables. The '`0 : 1`' or '`1`' side has a unary-valued attribute (a tuple identifier) which is permitted. As both attributes encode exactly the same information, we simply omit the set-valued attribute.

   Figure 3a shows an application of the last three rules: the `dept` table has two columns { #, `name` } and the `student` table has three { #, `utid`, `enrolledIn` }. Column `enrolledIn`, which contains a `dept#` value, encodes the `student` − `dept` association. The mapping of Figure 1a to 1b is another example of these ideas.

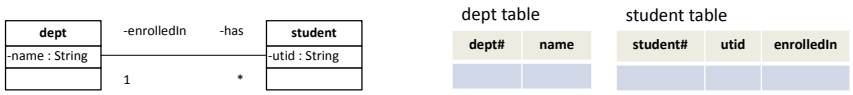


Fig. 3 Diagram-to-Table Mapping.

4. For classes that are related by inheritance, the attributes of each superclass table are included as attributes in each of its subclass tables. The identifier of the root class is shared by all subclasses. See Figure 4.

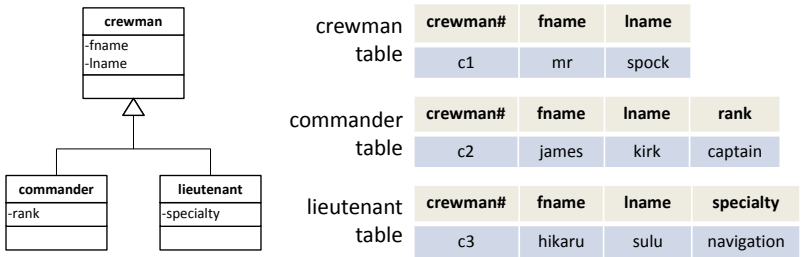


Fig. 4 Inheritance Diagram-to-Table Mapping.

5. In Java, objects are created for a specific class and implicitly become objects of superclasses. The same holds for tuples and tables. In Figure 4, the `crewman` table has three tuples (`spock`, `kirk`, `sulu`), although only one tuple (`spock`) was

created specifically for `crewman`. The other two tuples are inferred: the `commander` table has one tuple (`kirk`) and the `lieutenant` table has one tuple (`sulu`). This example is discussed further in Section 3.

6. Tuple identifiers can be manufactured (e.g., `e1` and `e3` in Figure 1d) or they can be readable single-column keys (e.g., `nReady` and `nDrink` in Figure 1d). Readable keys are preferred in hand-written assignments; tools use manufactured identifiers.

Observation 1. Relational tables have always been able to encode data hierarchies. The elegance of normalized or 'flat' tables highlights the conceptual simplicity of our approach. Appendix B illustrates a mapping of an aggregation hierarchy to a set of MDELite tables.

3 Model Constraints

OCL is the standard language for expressing model constraints in MDE. Given the connection to relational databases, there is an obvious alternative. Prolog is a fundamental language in *Computer Science (CS)* for writing declarative database constraints. It is Turing-complete and is a language that all CS students should have exposure. Figure 5 shows how we express the database of Figure 1. The first line of Figure 5a defines the schema of the `node` table of Figure 1b; it consists of three attributes { `id`, `name`, `type` }. Tuples of this table are Prolog facts. Figure 5b gives the encoding of the `edge` table.

(a)

```
table(node,[id,name,type]).

node(nstart,start,start).
node(nReady,ready,state).
node(nDrink,drink,state).
node(nEat,eat,state).
node(nPig,pig,state).
node(nstop,stop,stop).
```

(b)

```
table(edge,[id,startsAt,endsAt]).

edge(e0,nStart,nReady).
edge(e1,nReady,nDrink).
edge(e2,nReady,nEat).
edge(e3,nDrink,nDrink).
edge(e4,nEat,nEat).
edge(e5,nDrink,nEat).
edge(e6,nEat,nDrink).
edge(e7,nDrink,nPig).
edge(e8,nEat,nPig).
edge(e9,nPig,nStop).
```

Fig. 5 Prolog Tables for Figure 1d.

Here are three constraints to enforce on a FSM:

- `c1` All states have unique names,
- `c2` All transitions must start and end at a defined state, and
- `c3` There must be precisely one start state.

Their expression in SWI-Prolog [38] is given below; `error(Msg)` is a library call that reports an error. `allConstraints` is true if there are no violations of each constraint.

```
c1  :- node(Id1,Name,_),node(Id2,Name,_),not(Id1=Id2),
       error('non-unique names').
c2  :- edge(_,Starts,Ends),
       ( not(node(_,Starts,_)) ; not(node(_,Ends,_)) ),
       error('edge refers to non-existent node').
c3a :- not(node(_,_,start)), error('no start state').
c3b :- node(Id1,_,start),node(Id2,_,start),not(Id1=Id2),
       error('too many start states').
allConstraints :- not(c1),not(c2),not(c3a),not(c3b).
```

As a last example, Figure 6a replicates the `crewman` table inheritance hierarchy of Figure 4 and also shows our Prolog encoding of these tables, their individual tuples, and their inheritance constraint: every tuple of a subtable is also a tuple of its supertable.[3]

(a)

crewman table

crewman#	fname	lname
c1	mr	spock

commander table

crewman#	fname	lname	rank
c2	james	kirk	captain

lieutenant table

crewman#	fname	lname	specialty
c3	hikaru	sulu	navigation

(b)

```
table(crewman,[id,fname,lname]).

crewman(c1,mr,spock).

crewman(I,F,L):-commander(I,F,L,_).
crewman(I,F,L):-lieutenant(I,F,L,_).

table(commander,[id,fname,lname,rank]).

commander(c2,james,kirk,captain).

table(lieutenant,[id,fname,lname,specialty]).

lieutenant(c3,hikaru,sulu,navigation).
```

Fig. 6 Inheritance Constraints.

4 Model-to-Model Transformations

The semantics of an MDE application are captured by the data that is stored, model constraints, and in transformations, such as *model-to-model (M2M)* transformations. MDELite does not alter this, except instead of using languages that were specifically invented for MDE, we use Prolog both for writing declarative constraints and declarative database-to-database (M2M) transformations.

Suppose we want to perform the M2M mapping of Figure 7, where we shade abstract classes to make them easier to recognize. That is, we want to translate a database that conforms to Figure 7a (which is identical to Figure 1a) to a database that conforms to Figure 7b. The Prolog rules that express this transformation are:

```
start(Id,Name)  :- node(Id,Name,start).
stop(Id,Name)   :- node(Id,Name,stop).
normalState(Id,Name) :- node(Id,Name,state).
transition(Id,Sid,Eid,SName,EName) :- edge(Id,Sid,Eid),
                    node(Sid,SName,_),node(Eid,EName,_).
```

As Prolog is Turing-complete, database transformations can be arbitrarily complex. Appendix B illustrates a more complex M2M mapping.

[3] Prolog tuples can have unquoted atoms (`james`) and quoted atoms (`'James Tiberious'`).

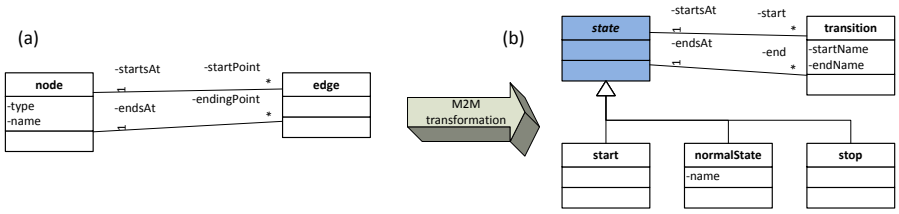


Fig. 7 A Model-to-Model Transformation.

Observation 2. There is an intimate connection between database design and metamodel design. Presenting MDE in the above manner reinforces this connection. Students *do not* have to be familiar with databases to understand the above ideas. Normalized tables are a fundamental and simple conceptual structure in CS. Undergraduates may already have been exposed to Prolog in an introductory course on programming languages. (When one deals with normalized tuples and almost no lists, Prolog is indeed a simple language). We chose Prolog for its database connection, but suspect that Datalog, Haskell, Scala, or other functional languages might be just as effective.

5 Model-to-Text Transformations

A key strength of MDE is that it mechanizes the production of boiler-plate artifacts (e.g., code). This is accomplished by *Model-to-Text (M2T)* transformations. There are many text template engines used in industry. Apache Velocity is an easy-to-learn example [4]. We made two modifications to Velocity to cleanly integrate it with Prolog databases. Our tool is called *Velocity Model-2-Text (VM2T)*.

First, we defined a Velocity variable for each table. If the name of a table is "`table`" then the Velocity table variable is "`tableS`" (appending an "S" to "`table`"). This enables a Velocity `foreach` statement to iterate over all tuples of a table, where a particular attribute `A` of a tuple is referenced by `$tuple.A`:

```
#foreach($tuple in $tableS)
  ...  $tuple.A
#end
```

Second, a Velocity template directs its output to standard out. We introduced markers to redirect output to different files during template execution. The value of the `MARKER` variable defines the name of the file to which output is directed; reassigning its value redirects output to another file. An example of `MARKER` is presented shortly.

As a running example, Figure 8a shows a metamodel for classes. Two instances of this metamodel, `city` and `account`, are shown in Figure 8b. The database containing both instances is Figure 8c.

Figure 9a is a VM2T template. When non-`MARKER` statements are executed, Figure 9b is the output. Preferably, the definition of each class should be in its own file. When all Velocity statements are executed, the desired two files are produced (Figure 9c).

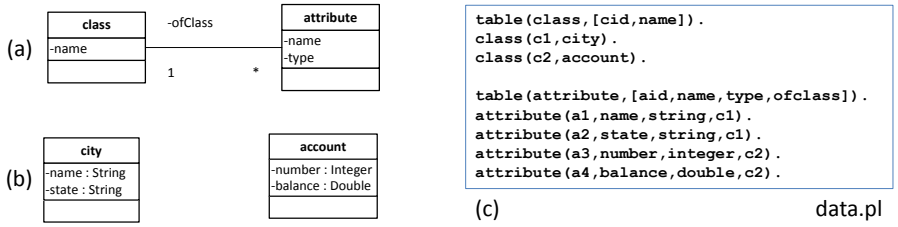


Fig. 8 A Class Metamodel, a Model Instance, and a Prolog Database.

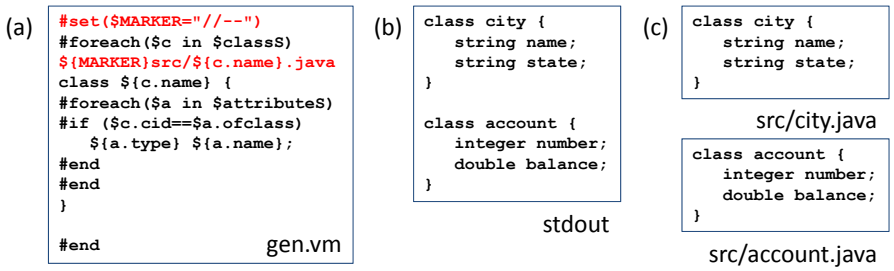


Fig. 9 A VM2T Template and Two Outputs.

Given VM2T, it is an interesting and straightforward assignment to translate the FSM database of Figure 1d to the code represented by the class diagram of Figure 10. The VM2T script for this example is given in Appendix C.

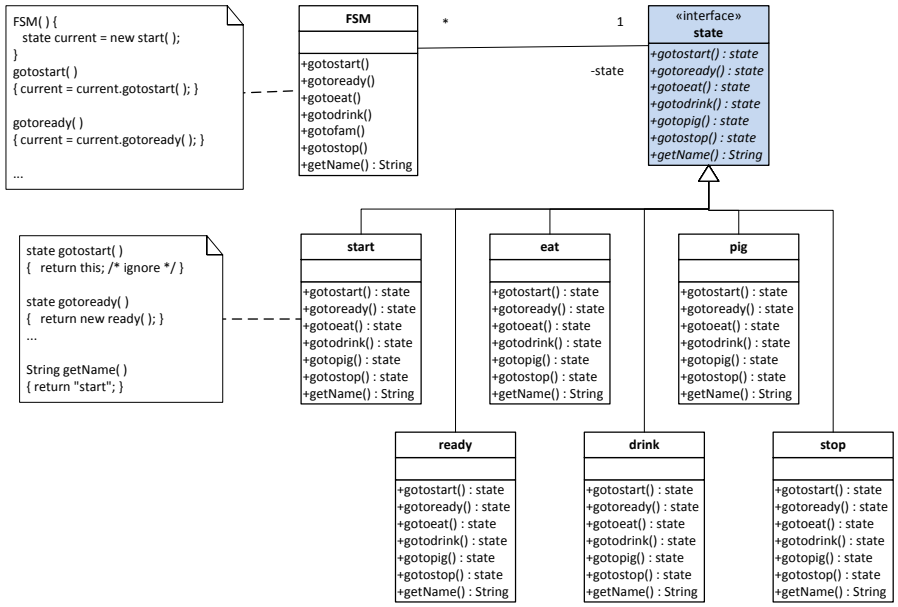


Fig. 10 Class Diagram of FSM Code Output.

Observation 3. The benefits of Velocity seem clear: students use an industrial tool that is not-MDE or Eclipse-specific; it is stable, reasonably bug-free, and has decent documentation. Velocity is by no means our only choice; Handlebars is another tool that looks promising [24]. More on this later.

6 Tool-to-Model Transformations

Given the discussion in previous sections on models, constraints, and transformations, it is not difficult for students to understand Figure 11: an application engineer specifies a FSM using a graphical tool, the tool produces a set of tables, the tables are transformed into a more appropriate tabular organization, and VM2T produces the source code for the FSM. All the engineer sees is a single tool that takes his/her FSM diagram and produces application source at the click of a button.

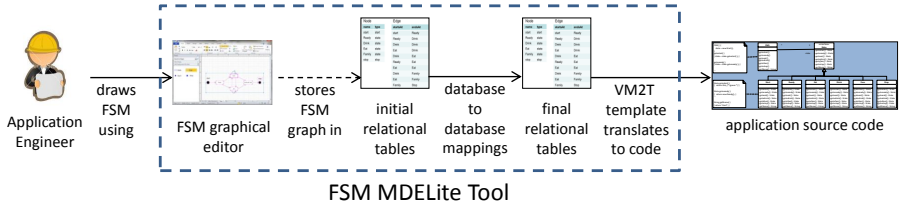


Fig. 11 FSM Application Engineering in MDE.

There are three missing pieces to complete this design. The first is a *Tool-to-Model (T2M)* transformation (the dashed arrow in Figure 11) that converts grossly-verbose XML output of a graphics tool into a clean set of Prolog tables. We wrote a Java program to read the tool's XML file, parse it, and output a text file containing a Prolog database. Frankly, of all the tasks in MDELite, writing the T2M program is the most painful – parsing XML, harvesting its data, and outputting Prolog facts is unpleasant. If the XML is simple, it turns out to be a good exercise for students who haven't parsed XML before to get this experience. For more complex graphical tools, writing an XML-to-Prolog transformation would require some expertise, but once done, it would become yet another tool in a library of tools that could be integrated into an MDELite application.[4] We note that such translations need to be written anyways to integrate foreign tools and their output into Eclipse MDE tools, so MDELite is not disadvantaged in this regard. Further, we will see later in Section 9 that students preferred MDELite over Eclipse.

The second missing piece is to find a suitable *graphical editor ($\mathcal{GE}$)* for Figure 11. This is a three-fold challenge:

(a) its XML is stable, meaning its XML format is unlikely to change,
(b) its XML must be simple to understand, and
(c) its palette[5] is customizable.

[4] The code for MDELite, including all of our T2M transformations, is free at http://www.cs.utexas.edu/users/schwartz/MDELite/index.html

[5] The set of icons that one can drag and drop onto a canvas to create instances.

MS Visio is easy to use and its palette is easily customizable, but its XML files are incomprehensible and periodically MS completely modifies the format of these files. Simpler $\mathcal{GE}$s, such as Violet [40], yUML [42], UMLFactory [39], satisfy (a) and (b); it is not difficult to write T2M tools for them.

We have yet to find a $\mathcal{GE}$ that satisfies all three constraints. Violet is typical: all palettes are hardwired – there is one non-extendible palette per UML diagram. It is impossible in Violet (without serious hacking) to define a set of icons with graphic properties to draw customized graphs. All that is permitted is to translate XML files that were specifically designed for a given UML diagram to Prolog tables. This is not bad; it just is not ideal.

> **Observation 4.** MDE is moving away from UML toward modeling environments that are domain-specific with icons and other abstractions that map to the domain. MDELite is agnostic to the specific modeling language. We used UML tools because they were available and are familiar to students, but any graphical specification tool can be used. One still has to convert its output into a form (Prolog tables) that can be subsequently processed.

The third and last missing piece is to add a button to the $\mathcal{GE}$ to initiate the computations of Figure 11. Alternatively, the engineer could supply the name of the XML file that is output by the tool to a command line program to invoke the computations.

> **Observation 5.** MDE tools, such as the FSM tool, could be structure editors. That is, a tool should immediately label incorrect drawings or prevent users from creating incorrect drawings. $\mathcal{GE}$s are typically stupid – they let you draw anything (such as edges that connect no nodes). To provide immediate feedback would require saving a design to an XML file, translating the file into Prolog tables, evaluating Prolog constraints, and displaying the errors encountered. As mentioned earlier, modifying existing tools with a button to activate this analysis and present this feedback can indeed be done, but is not high-priority for our goals.

7 MDELite and its Applications

MDELite is a small set of tools (SWI Prolog, VM2T) that are connected by a tiny Java framework that implements the ideas of the previous sections. An *MDELite application* uses this framework and is expressed as a category [7,34].

A *category* is a directed multigraph; nodes are *domains* and *arrows* are total functions (transformations) drawn from the function's domain to its codomain. Many of the interesting ideas about categories, like functors and natural transformations, are absent in the MDE applications of this paper, so there is nothing to frighten students. Nonetheless, it is useful to remind students that categories are a fundamental structure of mathematics, they are a core part of MDE formalisms [18] and they define the organization of an MDE application [7].[6]

[6] Also known as *megamodels* [12] and *tool chain diagrams* [32].

Recall the FSM MDELite tool of Figure 11. It embodies a tool chain whose category has four domains (Figure 12a): the domain of XML files that are produced by the FSM $\mathcal{GE}$, a domain of database instances that a T2M tool creates, another domain of database instances that results from a restructuring of T2M-produced databases, and a domain of Java Directories whose elements (directory instances) are FSM programs (a set of generated Java files).

When this category is implemented in Java, each domain is a manually-written Java class and each arrow is a method as indicated in Figure 12b. (We talk about the automated construction of these classes and methods in Section 11.) Underneath each domain in Figure 12a is a Java class in Figure 12b. Every class instance has a pointer to a file (XML, Prolog database, etc.) that instance represents. Every Prolog database class has a `conform()` method to validate database instances. The remaining methods correspond to category arrows that exit from that domain (e.g., `M2T` exits from the `FinalPrologTable` domain and thus is a method of class `FinalPrologTables`, `T2M` exits from the `FSMXML` domain and thus is a method of class `FSMXML`). Unlike most UML class diagrams, categories typically have no associations but can have inheritance relationships.

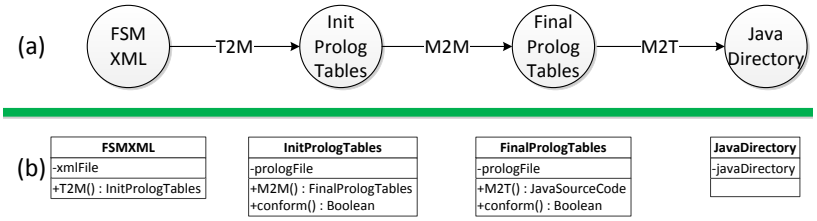


Fig. 12 Category of a FSM MDELite Tool.

These classes are then hooked into the tiny MDELite framework; framework classes are shaded in Figure 13. All database classes are subclasses of `GProlog` (a class that has utility methods to invoke Prolog transformation and constraint computations); the remaining classes are subclasses of `MDELiteObject` (a class that has utility methods for executing arbitrary programs, including `VM2T`). In Figure 13, `InitPrologTables` and `FinalPrologTables` are subclasses of `GProlog`; `FSMXML` and `JavaDirectory` are subclasses of `MDELiteObject`.

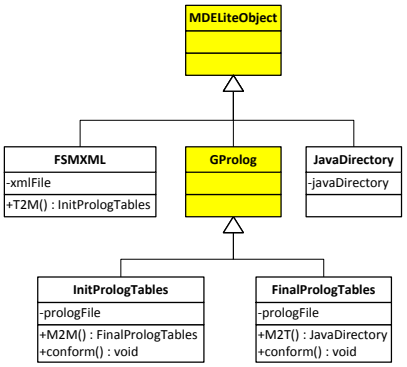


Fig. 13 MDELite Encoding of the Category of Figure 12.

A method of an MDELite class is a small piece of code that follows any one of a number of standardizable forms. The `conform` method of the `InitPrologTables` class is shown below:

```
void conform() {
   String[] list = ... , this.prologFile, ...
          "libpl/InitPrologTables.conform.pl" ;
   InitPrologTables t = new InitPrologTables(list);
   t.executeProlog();
   t.delete();
}
```

Here's what it does: a list of Prolog files are concatenated. One of the files corresponds to `FinalPrologTables` database in question (namely `"this.prologFile"`) and another the Prolog file that encodes model constraints to evaluate (file `"InitPrologTables.conform.pl"`) which is stored in a standard MDELite directory (`"libpl/"`). The concatenated file is executed. If no errors are discovered (which would halt the computation), the concatenated file is deleted. The only difference between the `conform` methods of class `InitPrologTables` and `FinalPrologTables` is literally the name of the class. In effect, the above code is a class-generic Java method.

To perform the computation(s) of the tool, a programmer writes a Java method for each activity. The only computation in our example is to translate an FSMXML file – an XML file produced by the FSM drawing tool – into a directory of Java files:

```
JavaDirectory tool(FSMXML x)
   throws RunTimeException {
   ipt = x.T2M();
   ipt.conform();
   fpt = ipt.M2M();
   fpt.conform();
   jd = fpt.M2T();
   return jd;
}
```

Here's what the above method does: the input XML file is mapped to a initial prolog database (`ipt`) whose conformance is checked. Then this database is restructured into a final prolog database (`fpt`) whose conformance is checked. Finally the `fpt` is translated to a directory of Java files by a `VM2T` script. Any error encountered during translation or conformance test simply halts the MDELite application with an explanative message.

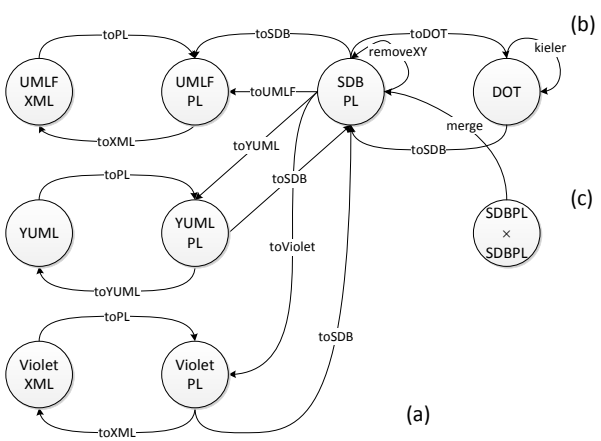
Observation 6. MDE lifts metamodel design to the level of *metaprogramming*—programs that build other programs [6]. The objects of MDE are programs (models) and the methods of MDE are transformations that yield or manipulate other programs (models). The elements of each domain are file system entities—an XML file, a Prolog file that encodes a database, or a directory of Java files—not typical programming language objects. Each MDELite method is literally a distinct executable: a T2M or M2T arrow is a Java program and an M2M arrow (and conformance test) is a Prolog program.

8 A Case Study of MDELite

Our first application of MDELite was instructive. We found several free UML tools that we wanted to:

(i) draw class diagrams,
(ii) verify the diagrams were legal, and
(iii) translate diagrams of one tool into corresponding diagrams of another.

The integration of the Violet [40], UMLFactory [39], and yUML [42] tools (as they existed in June 2012) is expressed by the category of Figure 14a.[7] We could draw UML class diagrams in each of these tools and have them displayed in any other tool.



(b)

```
yUML Violet2yUML(VioletXML v){
  vpl = v.toPL();
  vpl.conform();
  sdb = vpl.toSDB();
  sdb.conform();
  ypl = sdb.toYUML();
  ypl.conform();
  return ypl.toYUML();
}
```

(c)

```
VioletXML yUML2Violet(yUML y){
  ypl = y.toPL();
  ypl.conform();
  sdb = ypl.toSDB();
  sdb.conform();
  d = sdb.toDOT();
  dk = d.kieler();
  sdb1 = dk.toSDB();
  sdb2 = sdb.removeXY();
  sdb3 = sdb2.merge(sdb1);
  upl = sdb3.toUMLF();
  upl.conform();
  return upl.toXML();
}
```

Fig. 14 A Category for an MDELite Application.

We used a simple approach:

1. We created a database for each tool, such that the translation: tool file to Prolog database back to tool file was an equivalence mapping. We visually inspected the display of both files to determine if they were identical.
2. We created a database (`SDBPL`) which was the lowest common denominator across all tool databases; it encoded a FSM database representation that was tool-independent.
3. When positioning information was not exposed by a tool file, we computed it (see below).

Paths in a category represent computations. To convert a VioletXML file to a yUML file is the path `VioletXML.toPL.toSDB.toYUML.toYUML` in Figure 14a and the method of Figure 14b. Look at Figure 15: (a) shows a Violet class diagram, (b) is its computed `SDBPL` representation,[8] and (c) shows its yUML depiction.

[7] The only oddity of Figure 14a is the domain `SDBPL` × `SDBPL`, which is the cross-product of the `SDBPL` domain with itself. The `merge` arrow composes two `SDBPL` databases into a single `SPBPL` database (i.e., `merge` : `SDBPL` × `SDBPL` → `SDBPL`).

[8] `SDBPL` tables `classImplements`, `interfaceExtends`, `interface` have no tuples. The Prolog declaration `:- dynamic T/n.` states that prolog tuples for table `T` with `n` attributes is empty.

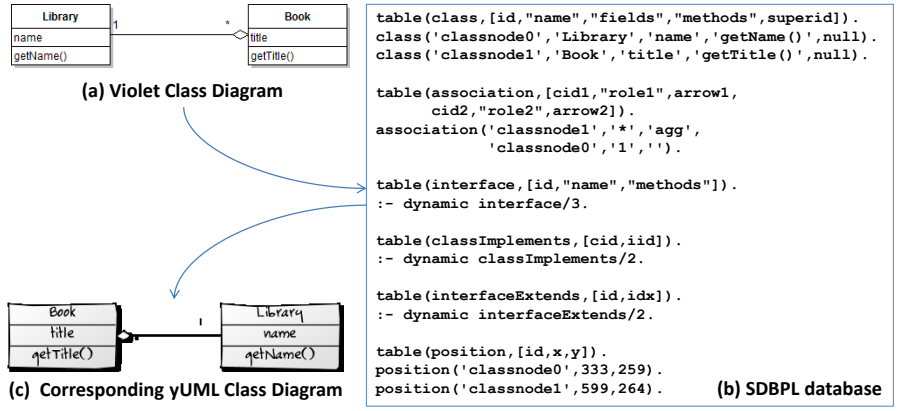


Fig. 15 A Violet Diagram mapped to an SDBPL database mapped to a yUML Diagram.

The reverse – converting a yUML file to a VioletXML file – is a bit more complicated. A yUML spec for the Library—Book class diagram of Figure 15c is:

```
[Library|name|getName()]
[Book|title|getTitle()]
[Library]<>1-*[Book]
```

Translating a yUML spec to the XML document of another tool requires graphical $(\mathtt{x},\mathtt{y})$ positioning information for each class (i.e., where each class is to appear on a canvas). yUML computes this information, but never returns it. Lacking positioning information, Violet simply draws all the classes on top of each other, yielding an unreadable mess.

We looked for tools to compute node positioning information for a graph and found the Kieler graph layout web service [29]. We translated an SDBPL database into a DOT graph [19], transmitted the DOT file to the Kieler server, it returned a new DOT graph with the required positioning information, and we stripped the Kieler response to create a `position` table. This computation is the traversal: `Kieler_path` = SDBPL.toDOT.kieler.toSDB in Figure 14a.

Look at Figure 16. Given the SDBPL database of the above yUML spec, the SDBPL.toDOT transformation produces a DOT specification (a) which defines a graph with two nodes $(\mathtt{c0},\mathtt{c1})$ that are linked $(\mathtt{c1} \rightarrow \mathtt{c0})$. This file is transmitted to the Kieler server, which returns (b): a DOT file with positioning information. The DOT.toSDB transformation parses the DOT file and strips away irrelevant information to yield the $(\mathtt{x},\mathtt{y})$ positioning information for nodes (classes) $\mathtt{c0},\mathtt{c1}$ expressed as tuples in a SDBPL `position` table in (c).

To complete the yUML-to-Violet transformation, we still need some additional computations. We started with an SDBPL database `s` whose `position` table contained useless class positioning information. Using the Kieler server, we created a useful `position` table replacement. The remaining steps are to project (remove) the useless position table from `s` and merge it with the Kieler `position` table replacement to produce a correct SDBPL database. This computation is expressed in the traversal $(\mathtt{SDBPL.removeXY}) \times (\mathtt{Kieler\_path})).\mathtt{merge}$.

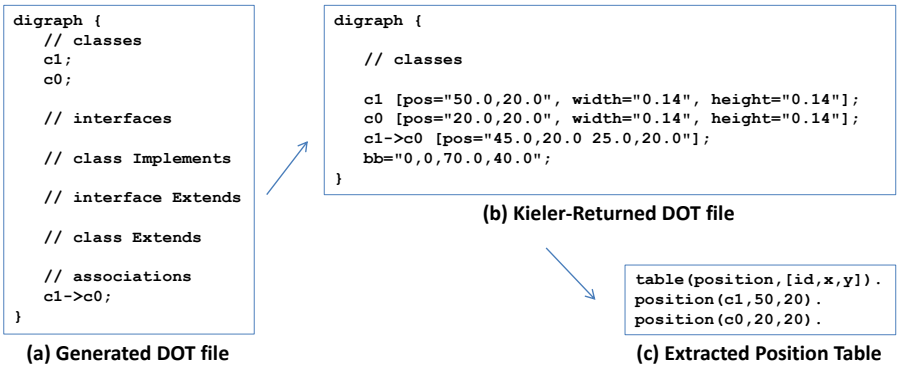


Fig. 16 DOT File Transformations.

This application required all kinds of T2M, M2T, and M2M transformations. Figure 17 lists the size of MDELite framework and this application in lines of Prolog, Velocity, and Java code. As the tables indicate, the framework is tiny; the application numbers indicate the volume of "code" that was needed to write this application.

Concern	LOC Prolog	LOC Velocity	LOC Java Java
MDELite Framework	84	0	581
MDELite Application	506	654	2532
Total	590	654	3093

Fig. 17 Size of MDELite Framework and Application: Lines of Prolog, Velocity, and Java Code

Observation 7. You can try this for any set of tools that satisfies constraints (a) and (b) of Section 6. Doing so, you will likely discover that your set of selected tools were never designed for interoperability. Ideally, interoperability should be transparent to users. Unfortunately, this is not always achievable. We found UMLFactory to be flakey; some tools had cases that we simply couldn't tell if they worked correctly. Hidden dependencies lurked in XML documents about the order in which elements could appear and divining these dependencies to produce decent displays was unpleasant (as there was no documentation). But it is a great lesson about the challenges of tool interoperability, albeit on a small-scale.

9 Student Evaluation

Increased emphasis is placed today on the experimental evaluation of educational programs [41]. By systematically applying different programs (teaching methods), researchers can study their effects and decide on which ones to keep [21]. The goal of MDELite is to provide an 'improved' learning experience and 'better' platform for teaching MDE. For this reason, we created a user study to evaluate MDELite as a teaching tool for MDE and to quantify its effects.

It is well-known that evaluations need to be feasible and appropriate to the realities of school-based outcome evaluation [41]. Although standard experimental designs can

establish causal relations between a new program and its outcomes, such experiments are uncommon in education [13,16]. A cited reason, among others, are ethical issues raised when students are denied access to a potentially better program when assigned to the control group [10,13,21].

A widespread alternative are *quasi-experiments*. Quasi-experiments are similar to experiments in that they compare the new program and baseline. The difference stems from participants not being randomly assigned to groups. Instead, evaluators use a non-equivalent control group for comparison [16]. Among these designs, cohort studies provide a viable option for conducting school-based outcome evaluation [41].

A *cohort* is a group of people with similar characteristics. In education the term is often used to describe successive groups that go through a grade level.[9] Having successive groups in the same environment (e.g., same class, similar group of students with comparable backgrounds, same lecturer, same study program) approximates a tight control of these variables.

Our first experiment was an informal cohort study. We introduced MDE in the Fall 2011 undergraduate CS378 Software Design course at the University of Texas at Austin. The goal of the course was to expose students to fundamental structures and concepts in software development, with an emphasis on automation and middleware. Students were in their 3rd and 4th year and had taken introductory courses on software engineering and programming languages. Topics covered included MDE, refactorings, design patterns, correct-by-construction parallel dataflow applications [8], and service oriented architectures. Within MDE, UML models, meta-modeling, OCL, the MOF architecture, and Eclipse editors were covered in three weeks of lectures, although programming assignments extended the topic by another two weeks. We already described the results.

We replaced Eclipse tools with MDELite in Fall 2012. The MDE module covered the same topics (sans Eclipse) plus transformations and ended with an assignment of comparable difficulty to Fall 2011. Although anecdotal, the results this time seemed better: installation and tool usage problems disappeared. Students had far fewer troubles grasping the idea of a database of tables and programs (be it Prolog or Java) to read and write such databases. Using Prolog to write statechart (metamodel) constraints was doable by students. Nevertheless, we did not have measurable data. In Fall 2011, the MDE programming assignments were removed from the computation of final grades as the instructor (Batory) felt they were not representative of student performance; not so in Fall 2012 where they were included.

This experience encouraged us to evaluate MDELite more systematically in the next class. As research indicates that students are generally good judges of teaching effectiveness [30], we designed an evaluation to gather their perception.

In Fall 2013 we carried out this evaluation in the graduate *CS392F Automated Software Design* course at the University of Texas at Austin. The reason why undergraduates were not used was because the first author (Batory) did not teach the undergraduate course that academic year. There were, however, students in his graduate class who could provide insights and this opportunity offered a broader evaluation

[9] In some research, the term *cohort* is also used to refer to any group that is repeatedly measured over time, as in a longitudinal or panel study; however, this is a different use of the term [21,35]

that included writing M2M transformations. The goal of the evaluation was to gather student opinions on MDELite and to detect deficiencies in MDELite and its tools.

9.1 Evaluation Design and Execution

We followed the guidelines of Jedlitschka [26]. To provide students with enough information to give a balanced opinion on MDELite, students worked with both alternatives (i.e., MDELite and the baseline), resulting in a quasi-experimental design with repeated measures [35]. We state the goal of our evaluation using the *Goal/Question/Metric (GQM)* method [5]:

- *Analyze* the classroom instruction of MDE using MDELite
- *for the purpose of* comparing it with a baseline alternative
- *with respect to* student perception of MDELite
- *from the point of view of* a researcher trying to assess MDELite
- *in the context of* case studies selected from the ATL Zoo.

The baseline was the Eclipse MDE tools from Fall 2011 (albeit updated), namely Eclipse Modeling Tools, Graphical Modeling Tooling Framework Plug-in, OCL Tools Plug-in, and Eugenia for Eclipse 3.6.2.

Variables. The independent variable in our study was the tool: MDELite (using Prolog for writing constraints and M2M transformations) and the baseline (using OCL for constraints and ATL for M2M transformations). The dependent variables measured student perception of the tools, constraint languages, and M2M transformation languages via questionnaires. We also measured the time it took to complete the assigned tasks with each tool – as an indicator of *productivity* [25].

Participants. Twelve students participated. Four weeks were dedicated to MDE, covering metamodeling, M2M transformations, and M2T transformations using MDELite. 30% of the final mark in the course was a five-week project and the MDELite project was a possible topic. 12 of the 25 students chose this project. At the beginning of the course, we surveyed their status.[10] All participants were from Computer Science, 55% in the Masters Program and 45% in the Bachelor Program. Figure 18 surveys their prior knowledge. 67% were unfamiliar with MDE, 78% were unfamiliar with Prolog, and 89% were unfamiliar with categories. Students worked in five groups of two or more, being two graduate student groups, two undergraduate student groups and the remaining one a hybrid group.

Students provided weekly updates of their projects through a Google Groups web page, which was also used by the instructor to post project clarifications.[11] At the end of the semester, a joint presentation was given by all groups with their conclusions. They were given our questionnaire *after* the students had received their project grades to avoid biased responses if they believed their answers could influence their grade. The responses were analyzed by one of the authors (Azanza) who had no direct contact with the students.

[10] Nine out of the twelve students chose to give their background information

[11] https://groups.google.com/forum/#!forum/atl-ocl

Major Or Department		Status at the University of Texas	
Answer Options	Response Percent	Answer Options	Response Percent
CS	87	Bachelor	20
ECE	13	Masters	74
other	0	Ph.D.	6

(a) Participant Status

Answer Options	% Very Familiar	% Somewhat Familiar	% Not Familiar
UML Class Diagrams	33	52	15
Model Driven Engineering	0	22	78
Relational Databases	33	56	11
Program Transformations	4	22	74
Category Theory	0	7	93
BNF Grammars	26	33	41
Parsers	30	48	22
Prolog	7	22	70

(b) Participant Prior Knowledge

Fig. 18 Participant Demographics.

Tasks. ATL is an integral part of the Eclipse Modeling Tools, a modern model transformation language and toolkit for MDE [28]. Among the contributions of ATL and its project is the ATL Zoo, which is a collection of over one hundred MDE applications that have been written in ATL. We found it to be a good source that could be used as examples of MDELite applications. Each group implemented three case studies from the ATL Zoo using both MDELite and the baseline.

Zoo Application	*Description*	*LOC*	*Nº Rules*	*Nº Helpers*	Nº Groups	Group Type	Completed in	
							Baseline	MDELite
Families$_2$Persons	maps a database of family to a database of person tables (see Appendix B).	42	2	2	5	All	✓	✓
Tree$_2$List	maps trees with weighted nodes to a list in which nodes are sorted in order of their weights.	33	2	2	2	Grad, Hyb	✓	✓
MySQL$_2$KM3	translates an XMI representation of a MySQL database schema into a KM3 XMI file.	438	11	17	1	UGrad	✓	✓
RSS$_2$ATOM	RSS and ATOM are XML-based languages for easy web syndication. Converting from RSS to ATOM is mostly a matter of renaming attributes – but both specs are massive.	56	3	0	1	UGrad		✓
TT$_2$BDD	transforms a truth table into a binary decision diagram.	168	6	5	1	Grad		✓
Geometric Transformations	given a set of input points in 3-space, perform a translation and scaling to produce output points.	171	3	11	1	Grad	✓	✓
KM3$_2$DOT	transforms a KM3 metamodel into a class diagram drawn with DOT graph layout language.	320	7	18	1	UGrad		✓
Table$_2$TabularHTML	transforms a table model to an HTML file with HTML tables.	151	7	12	1	UGrad		
Make$_2$Ant	transforms a makefile into an Apache ant file.	64	14	0	1	Hybr		
Table$_2$Excel	transforms a table model to an XML Excel file.	93	3	1	1	Grad		✓

Fig. 19 ATL Zoo Applications Used.

All groups implemented the ATL tutorial application ***Families$_2$Persons***, then they chose a second (medium-difficulty) and third (hard-difficulty) application. Figure 19 lists those that were used. To give a flavor of the complexity of the applications, we

list the LOC, number of rules and number of helpers of the solutions provided in the ATL Zoo, together with the number of groups that implemented each of them. The MDELite solution to **Families$_2$Persons** is given in Appendix A.

Instrumentation. Students were given lecture notes for MDELite. For the baseline, they were given the instructional material prepared for the Fall 2011 course.

9.2 Analysis

We wanted answers to the following research questions:

- **RQ1:** Does MDELite reduce student effort to develop applications?
- **RQ2:** How do students perceive MDELite as a tool to learn MDE?
- **RQ3:** How do students perceive Prolog as a constraint language?
- **RQ4:** How do students perceive Prolog as a M2M transformation language?

The quantitative analysis for each question was complemented with a qualitative analysis of open questions. Quantitative answers are analyzed using non-parametric Wilcoxon signed-rank tests to assess significance. The test assumes that there are no differences between the two measurements, MDELite vs. baseline (referred to as the null hypothesis). Then, the conditional probability of having observed the results under the assumption that the null hypothesis is valid is computed. If that probability `p` is low enough (typically $\texttt{p} < \texttt{0.05}$), the null hypothesis is rejected. Unless otherwise noted, results correspond to a five point Likert scale (1-Completely Disagree, 5-Completely Agree).

*9.2.1 **RQ1:** Does MDELite reduce student effort to develop applications?*

As described above, each group implemented three case studies from the ATL Zoo using both MDELite and the baseline. All implemented **Families$_2$Persons** and they had to choose a second (medium-difficulty) and third (hard-difficulty) applications.

The distinction between medium and hard difficulty applications did not prove useful. When rating application difficulty, the medium one scored 3.67 (1-5 scale) while the hard one scored 3.50. When asked about the invested effort, both in MDELite and the baseline the 'medium-difficulty' application took longer than the 'hard' one (4.17 vs 4.08 hours in MDELite and 6.68 vs. 4.29 hours in the baseline). Hence for purposes of presentation, we present measured results of **Familes$_2$Persons** and the accumulative remaining set of **ATLApplications** in Figure 19, together with whether the groups were capable of finishing the application with each alternative and their type (i.e., undergrad, graduate or hybrid). Figure 20a presents the aggregated perceived difficulty of the applications.

Consider the effort students invested in each implementation. Figure 20b presents the average and standard deviations for both tools and calculates the non-parametric Wilcoxon signed-rank test (`z`) and its corresponding `p` value. We found no statistically significant difference using baseline and MDELite for **Familes$_2$Persons** and **ATLApplications**, indicating that MDELite had no impact on student productivity.

Item (1-5 scale)	Families$_2$Persons		ATL Applications	
	Ave	Std	Ave	Std
How would you rate the application difficulty?	2.33	0.79	3.75	1.06
How much did you rely on the ATL answer?	4.17	1.03	4.08	1.16

(a) Reliance on answer and Difficulty

Item (hours)	Baseline		MDELite		z	p
	Ave	Std	Ave	Std		
Approximately, how long did you take to develop Families$_2$Persons in X?	1.94	1.56	2.77	2.24	1.072	0.284
Approximately, how long did you take to develop the ATL Application in X?	3.79	1.85	3.75	2.31	0.360	0.719

(b) Productivity

Fig. 20 Responses on developed applications.

Focusing on the baseline, how much students relied on the ATL answer provided in the Zoo is listed in Figure 20a. Note that, while the perceived difficulty was not very high (2.33 for **Familes$_2$Person** and 3.75 for **ATLApplications**), 60% of the **ATLApplications** were unfinished in baseline (compared to only 20% unfinished in MDELite) and the reliance on the posted ATL solution for students to develop their ATL solution was high (over 4 in all cases). Students admitted having to consult the solution provided in the ATL Zoo to perform the task.

In summary, we found no statistically significant differences between the baseline and MDELite. Consequently, we cannot assert that MDELite improves productivity.

*9.2.2 **RQ2:** How do students perceive MDELite as a tool to learn MDE?*

Figure 21 summarizes student perceptions of MDELite, again using the average and standard deviations for both tools and the non-parametric Wilcoxon signed-rank test (z) and its corresponding p value. The last column shows the effect size ($0 \leq r \leq 1$) in the cases where statistically significant differences are found.

Item (1-5 scale)	Baseline		MDELite		z	p	r
	Ave	Std	Ave	Std			
In general, I found X easy to use.	2.00	0.74	4.08	0.52	3.134	0.002	0.640
In general, I found X adequate to perform the required assignments	3.08	1.24	4.00	0.74	1.876	0.061	
I found that the installation process of X was straightforward	1.83	0.84	4.08	0.67	2.977	0.003	0.608
I found that the messages X provided helped me know what was going on	1.42	0.52	3.17	0.58	3.109	0.002	0.635
In the future, if I apply Model Driven Engineering, I will use X.	2.33	0.99	3.75	0.75	2.812	0.005	0.574

Fig. 21 Responses on the Tools.

Only item 2 of Figure 21 we did not find significant differences between tools. For the rest, the differences between tools *are* statistically significant (i.e., $p < 0.05$) and favor MDELite in all cases. Moreover, the effect size (the impact of using MDELite) is large in all four items ($r > 0.5$). That is, MDELite was perceived to be easier to use, easier to install, its messages were easier to understand, and its use would be preferred in the future over the baseline.

To evaluate MDELite as a teaching tool, we found student responses to items 1 and 4 interesting. Item 1 suggests that MDELite is significantly easier to use and item 4 indicates that its messages help more in understanding what is going on during

application development. We believe this is essential in learning MDE. This was corroborated by their responses to the open question:

- 'From a learning standpoint MDELite was much easier for me to grasp.'
- 'MDELite is a much better tool for teaching MDE... I like that you can see each step of the transformation happening. Prolog is not always intuitive to students since generally we're not taught languages like Prolog, but it's not impossible to learn.'
- 'I felt like MDELite was overall little easier to use and learn (because it was, at least for our group, more of learning languages compared to learning the tool itself).'
- 'I do think that part of the reason why in general I prefer MDELite is that we had more experience in it due to coursework; however, in making comparisons about the ease of learning how to use tool X, I tried to think about how easy it was to learn MDELite in September (which I think was still easier than learning ATL in recent weeks).'
- 'MDELite was generally easier to use. The exception was when intricate recursion was needed to generate the transformation.'
- 'Hard to tell how difficult ATL would have been without the solutions provided. Lack of documentation really hurts. MDELite was generally easier except for the geometric transformations version.'
- 'At this stage I cannot really say confidently whether I would find ATL/OCL as easy if we weren't building up from examples but doing things from scratch. By contrast I do remember that it was relatively easy to pick up on the MDELite end when I started using it for my first class assignment, though we weren't just replicating known examples. So I believe that MDELite is a lot more welcoming to newbies.'

A possible elaboration of the above comments were the differences in how the tools show what is going on:

- 'The idea of having a "1-click" transformation is nice for practical use, but not good for teaching someone about transformations and the various steps between the input file and the output file.'
- 'EMT tools need some work. For one, when they are working, I have no idea what is truly going on.'
- 'In MDELite I like that you can see each step of the transformation happening.'

Summing up, in general students perceived MDELite as a good teaching tool (easy to use, easy to install and its messages were easy to understand).

9.2.3 ***RQ3:*** *How do students perceive Prolog as a constraint language?*

Figure 22 lists the items we posed. Statistically significant differences were found in items 1, 3 and 5. These numbers show that students perceived Prolog to be a better language to write metamodel constraints, that they became familiar enough with Prolog to write the required constraints after a shorter time and found that Prolog

Item (1-5 scale)	Baseline		Prolog		z	p	r
	Ave	Std	Ave	Std			
I found that X is an adequate language to write metamodel constraints	3.08	1.00	4.27	0.47	2.521	0.012	0.580
I found that X is an easy language to learn	2.42	0.67	3.33	1.07	1.813	0.070	
I became familiar enough with X to write the required constraints after a short time	2.58	0.90	3.75	0.87	2.333	0.020	0.476
I found that X eased my grasp of metamodel constraints.	2.92	1.17	3.83	1.03	1.639	0.101	
I found that X improved my ability to write metamodel constraints	2.58	0.51	4.00	0.85	2.850	0.004	0.582

Fig. 22 Responses on constraint languages.

improved more their ability to write metamodel constraints. The effect size of using Prolog is medium for Item 3 ($r > 0.3$) and large for items 1 and 5.

Two students commented that OCL (i.e., the baseline) was easier to learn and three felt that OCL is less powerful than Prolog:

- 'The baseline provided a very convenient way to define metamodels and constraints. However, I felt it was definitely limited compared to Prolog, which was harder to write constraints but was more flexible and extensible.'
- 'After the initial learning curve, I found Eclipse's GUI for creating metamodels quite nice. Constraints however were very difficult to enforce and create.'
- 'Prolog is definitely much more powerful in defining metamodel constraints.'
- 'Even though MDELite initially had a steeper learning curve because of Prolog, it was definitely easier to use overall. Prolog is definitely much more powerful in defining metamodel constraints.'

Considering the qualitative and quantitative data, students perceived that Prolog is a good language to write metamodel constraints and that they became familiar enough with it to write the required constraints quickly. However, no statistically significant differences were found to indicate which language was easier to learn.

9.2.4 ***RQ4:*** *How do students perceive Prolog as a M2M transformation language?*

Responses for this variable are summarized in Figure 23. Statistically significant differences were only found in items 3 and 5. That is, students became familiar enough with Prolog to do the required M2M transformations after a shorter time and found that Prolog improved more their ability to write M2M transformations. The effect size is large in the case of item 3 and medium in the case of item 5.

Item (1-5 scale)	Baseline		Prolog		z	p	r
	Ave	Std	Ave	Std			
I found X useful to create M2M transformations	3.42	0.67	3.82	0.72	1.406	0.160	
I found X is an adequate language to write M2M transformations	3.17	0.94	3.67	0.49	1.613	0.107	
I became familiar enough with X to do the required M2M transformations after a short time	2.50	1.00	3.58	0.52	2.739	0.006	0.559
I found that X eased my grasp of M2M transformations.	3.00	0.94	3.33	0.99	1.006	0.314	
I found that X improved my ability to write M2M transformations	2.92	0.79	3.83	0.72	2.156	0.031	0.440

Fig. 23 Responses on M2M transformation languages.

In open questions, different students commented on ATL being easier to understand, but more difficult to write from scratch:

- ' ATL's syntax was a major barrier to understand what we were doing as we were composing the code. It is expressive in the sense that when you look at it you

can get a very good idea of what it means, but producing it from scratch was troublesome.'
- 'Assuming I am equally versed in Prolog and ATL, I think that ATL code is more readable. However, from a learning standpoint MDELite was much easier for me to grasp (at least for the easy examples).'
- 'I felt like ATL/OCL was easier to understand M2M transformations as it has visual representations of metamodels that are pretty easy to understand, and rules show how one type of class is transformed into another type of class (in another metamodel). Prolog constraints were definitely more powerful in creating different metamodels but also little hard to visualize. For someone new to M2M transformations, it is probably easier to understand what's going on with ATL/OCL (writing one from scratch is completely different story though—MDELite is definitely better in that sense, at least for us).'

Students also commented on the balance between the generality MDELite offers as opposed to the domain specificity of ATL:

- 'MDELite is like a truck: it's capable of getting over any kind of terrain and hauling some gear too, but its gas mileage is inefficient. ATL is like a sports car: it can't handle all terrain, but it's very quick on a race track. I felt that if I had a strange problem to deal with, I could use MDELite's loose combination of Java, Prolog, and Velocity to figure out a solution. ATL doesn't offer that. The Eclipse GUI keeps you constrained to a very specific way of doing things and doesn't let you fool around to learn what's actually going on. However, when the way ahead is very clear, ATL provides an elegant solution.'
- 'What I thought was interesting was that MDELite's shortcomings were often ATL/OCL's strengths, and vice versa.'
- 'MDELite is a good contradiction (contrast) to the ATL. It uses general purpose languages to provide an acceptable solution to writing M2M transformations. BUT, its use of general purpose languages makes fundamentally limited. A well implemented DSL for M2M transformations has a much higher ceiling than MDELite will ever be able to achieve.'
- 'ATL definitely has the advantage of being purpose built for M2M transformations, and it shows. However the required time to understand MDELite was considerably less.'
- 'For transformations, Prolog was sometimes easy to use (for simple examples) but as examples got more complex in the M2M conversions needed, it got to be a pain.'

and how choosing one or the other depends on the domain at hand:

- 'I think using ATL vs Prolog depends a lot on the use case. For instance if the transformations were simple – like in families to persons where you were just converting members to individuals – Prolog would definitely be a lot easier to use. But I think in more complicated scenarios Prolog would be less desirable mainly because ATL seems a lot closer to the imperative programming paradigm, we are used to. So I think ATL would help us get the tasks done a lot quicker.'

Some students commented on the difficulties of handling recursive transformations using Prolog:

- 'MDELite seems poorly suited for tasks involving the definition of a recursive model or recursively-based transformations.'
- 'MDELite was generally easier except for the geometric transformations version.'
- 'Recursive nature of geometric transformations was difficult to get working in MDELite.'

In general students perceived that M2M transformations were easier to write in Prolog but that ATL is more suited for complex transformations.

9.3 Summary

Below we briefly summarize the answers to our research questions:

RQ1: *Does MDELite reduce student effort to develop applications?* We found no evidence that MDELite improves student productivity when writing MDE applications.

RQ2: *How do students perceive MDELite as a tool to learn MDE?* Students found that MDELite was significantly easier to use, easier to install and its messages being easier to understand than the baseline. In general they perceived it to be a good teaching tool.

RQ3: *How do students perceive Prolog as a constraint language?* Students found that Prolog is a good language to write metamodel constraints and that they became familiar enough with it to write the required constraints quickly. However, no evidence was found to indicate that it is easier to learn than the baseline.

RQ4: *How do students perceive Prolog as a M2M transformation language?* Students found that they became familiar enough with Prolog to do the required M2M transformations after a short time and that Prolog improved their ability to write M2M transformations (except in the case of geometric transformations). They commented on Prolog being easier to learn than the baseline. Nevertheless, they found ATL more suited for complex transformations.

These results, together with student comments and their answers to the open questions gave us valuable feedback on how to improve MDELite in the future.

9.4 Threats to Validity

A first issue relates to sample size. Johnson et al. suggest six participants per group as the minimum required for a controlled experiment [27]. Our experiment used twelve participants divided in five groups. Even though our results show statistically significant differences and large effect sizes, larger groups are needed to corroborate these findings.

Another issue relates to the background of students. Our students were enrolled in the University of Texas at Austin. It would be interesting to replicate the evaluation in another setting to see if the findings are corroborated.

A third concern is the evaluation design. The course taught MDE concepts using MDELite (4 weeks) and then students carried out the project with both MDELite and the baseline (5 weeks). In this time teams had access to the baseline material, had regular meetings with the instructor, but no classroom discussion was dedicated to the issue. While we believe students provided balanced and coherent feedback, a similar evaluation in a setting that taught the baseline and then did MDELite as a project might reveal a different perspective.

Moreover, the evaluation was performed in a graduate course, when MDELite is firstly aimed at undergraduate students. The reason was opportunistic, the first author (Batory) did not teach the undergraduate course that academic year and we wanted to gather student perception on MDELite and to identify areas for improvement. Nevertheless, 45% of the students that participated in the evaluation were undergraduates and we found no significant differences between graduate and undergraduate students.

Last, while we tried our best not to influence students in any way during the project and asked for their honest opinion, them knowing that MDELite was created by their instructor could have had an impact on their answers.

10 Lessons Learned

Multi-Paradigm Programming. In general, students are Java programmers and novices to Prolog. Prolog and Java have two very different mind-sets, and flipping between paradigms can be confusing. Trivial things like Prolog rules ending in (Java) semicolons instead of (Prolog) periods was a mistake constantly made. Prolog inequalities (=<) are syntactically reversed in Java (<=). In SWI-Prolog, when something is mistyped, a question-mark prompt (?) is produced and the usual Windows/Linux character escapes (e.g., `ctl-C`) to reset to the command prompt fails. Problems like these disappear once familiarity with Prolog sets in—they clearly are not fundamental, but are jolting to students in a first, quick immersion into Prolog. For this reason, we recommend that MDELite be a pair-programming project: one person concentrating on Prolog, the other on Java, to minimize cross-paradigm confusion.

Many-Columned Tables. When there are many columns, it can be daunting in Prolog to correctly reference a table and account for each of its columns in a predicate. In such cases, one can M2M transform such tables into RDF 3-tuple format of (`tupleid`, `columnName`, `value`) or a 4-tuple format (`tableName`, `tupleid`, `columnName`, `value`) for easy attribute referencing.

Transformation Debugging. MDELite provides a microcosm of the challenges of debugging transformations. Even though a transformation takes an object (a model) as input and produces an object (a model) as output, objects are Prolog databases that are anything but simple values and can have complex

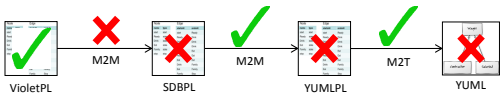


Fig. 24 Debugging Transformation Scripts.

structures. *Writing transformations in any language is not simple*—it is easy to forget a case or incorrectly code a translation. Our hunch is that the simpler a transformation's specification, the easier it will be to track down errors. This remains, however, a conjecture.

A technique that we found useful—perhaps motivated by the "shape" of the category of Figure 14a—was to define a transformation τ and then its inverse τ^{-1}, so that we could test whether $\tau \cdot \tau^{-1}$ was an identity or an equivalence. This helped, but obviously did not eliminate all bugs.[12]

Nonetheless, the fundamental challenge in debugging transformations becomes clearly evident: an error is detected in a database (far right of Figure 24). Upon examination, we discovered that the transformation that produced it was correct, but its input database was incorrect. This unwinds backwards until we discover a correct database that was input to a transformation that produced an incorrect database. Surely results on debugging Prolog programs and debugging database transactions—studied long ago—might be useful to MDELite. This too remains a conjecture.

Velocity Limitations. When Velocity templates have many loops and `if` statements, it is easy to lose track of loop and if-then-else boundaries, thereby creating incorrect templates. One reason why loops and if-statements are used is to join tables. For example, consider the following Class table rows, where class `Customer` is connected to class `Address` via a $^{*} \rightarrow^{1}$ association:

```
class(c1,'Customer','','','').
class(c2,'Address','','','').
association(c1,'*',none,c2,'1',arrow).
```

To correctly compute the set of attributes that each class will have requires a Velocity template that joins these tuples.

Another issue, one that is much more difficult to solve, is that Velocity provides very limited capabilities to manipulate Strings—which ironically is the purpose of Velocity. Simple Java String methods can be invoked within a template, but any significant manipulation requires hacking (i.e., distorting what would otherwise be a simple and straightforward template).

We mentioned earlier that it is possible to use a Prolog M2M transformation to 'prepare' a database that is better suited for (i.e., makes it easier to write) Velocity templates. But a well-known weakness of Prolog is its clumsy handling of strings, which limits the use of Prolog M2M transformations for this purpose.

We have since discovered other M2T (text-template) tools, such as Handlebars [24] that offers Velocity-like functionality with the ability to allow customized Java methods to be invoked from templates. To offer Handlebars (or other text-template tools) will simply make MDELite more of a framework, where programmers can pick-and-choose the text-template tools that they want to use.

[12] Two documents $\mathtt{d}_1$ and $\mathtt{d}_2$ can differ in whitespace, ordering of declarations, etc. and still represent equivalent class diagrams.

Prolog Limitations. Although Prolog is Turing-complete, there are applications for which is is poorly suited. One of the applications selected from the ATL Zoo involved geometric transformations. Students had to recode ATL computations in Prolog which was unpleasant. It is not difficult to extrapolate this to more computation-extensive transformations – where highly-tuned modules were programmed, say, in the `C` language are already available to accomplish this task. This would mean that cross-language calls from Prolog to whatever language is daunting and unattactive. This lead us to realize that picking-and-choosing text-template tools should also lead to the picking-and-choosing of M2M transformation languages, where the reading and writing of readable Prolog databases—rather than obscure XML documents—is the key to generalizing MDELite. While we could not express such generality in MDELite, a next generation of MDELite does permit this [15]. So in our opinion, a practical benefit of our user study was to reveal the direction for our future research.

11 Current Status, Bootstrapping, and Course Lectures

Current Status. The framework of MDELite is tiny because MDELite, as described in this paper, does not have a meta-meta model nor was bootstrapped. It became clear soon enough that classes and methods of an MDELite application that we wrote manually (as described previously) could indeed be generated from a category diagram of an MDE application.

BootStrapping. Since our original publication [9], we have made two efforts to bootstrap MDELite. The first was accomplished by graduate students in Fall 2013 using the existing MDELite framework. Effectively it did the following: given a category spec, it produced a set of Java files, which would then be compiled and executed like a manually-written MDELite application.

A second and more ambitious attempt was made by the first author (Batory) in Summer 2014 to reimplement MDELite not as a code-generator but as an interpreter. The project was called *Catalina*. An entire MDE application was expressed as a category diagram, from which a set of Prolog database tables was derived, and executions (e.g., paths through categories) were interpretively executed.

Catalina went further, as it prototyped an IDE plug-in. For example, to write or debug a VM2T transformation, a triplet of text windows would be presented to a programmer: one showing the current Prolog database, another the VM2T template, and a third the source that the VM2T template generated given that Prolog database. In this way, a programmer was given a cocoon-like environment in which VM2T templates could be dynamically and incrementally developed. The programmer would edit the VM2T template in one window, push a button, and see the result of its execution in the third window. Similar environments were created to develop Prolog constraints and Prolog M2M transformations. In principle, this triad of windows worked extremely well.

Catalina was used in a graduate class on Automated Software Design in Fall 2014. The first author believes that Catalina's ideas sketched above are correct. However, it is not clear that Catalina was successful. In a nutshell, consider at Figure 25.

Tool	LOC Prolog	LOC Velocity	LOC Java Java
MDELite	590	654	3093
Catalina	1130	383	8237

Fig. 25 Size of MDELite and Catalina: Lines of Prolog, Velocity, and Java Code

Catalina is at least $3\times$ the size of MDELite, but it feels $10\times$. It was a nightmare to build the specialized window management that Catalina needed, and even though it worked, Catalina's implementation didn't feel "integrated". The three windows described above were realized as separately spawned text editors that were not managed within a single window frame; a programmer had to manage these windows manually as a group.

The bottom-line is that we are still exploring the simplest way to bootstrap and present MDELite ideas. We have gained far greater appreciation of the difficulties of creating a usable IDE environment for MDE in general, and MDELite in particular; it was far more difficult than we imagined. A fuller account of Catalina, with its lessons learned and trade-offs, may be described in some future paper.

Course Lectures. In the meantime, our basic lectures on MDELite have not fundamentally changed. In a graduate class, our first lecture covers the contents of this paper up through and including T2M transformations (Section 6). A second lecture briefly reviews Prolog and how to use Prolog to write model constraints and M2M transformations within the MDELite/Catalina environment. The third lecture explains how MDELite can be bootstrapped. As this section reveals, we have not yet settled on the best way to bootstrap and what constitutes "best".

Our undergraduate curriculum follows a similar trajectory, but at a *much* slower pace. The first lecture familiarizes students with UML class diagrams. The second explains how meta-models are mapped to database schemas, how models (and UML Object models) are mapped to databases. The third explains how Prolog can be used to express model constraints and a fourth reinforces these ideas with in-class exercises, hinting at Prolog M2M transformations.[13] The idea that an MDE engineer draws a program specification and generates code from it, as in Figure 11, are clear by then. We have not observed any advantage for students who had prior database knowledge, or any disadvantage for students not knowing database concepts prior to this course.

In both classes, there is a small number of lectures on MDE concepts. Assignments to reinforce these ideas, both written and programming, extend weeks past the end of these formal lectures. This matches our instincts: the ideas behind MDE are elegant, simple and powerful. To appreciate them requires hands-on experience which takes longer.

12 Related Work

At the presentation of MDELite at the MODELS13 conference, Pantel observed that the STOOD, ADELE, and AADL-inpector tool sets rely on Prolog for model analysis

[13] The undergraduate class is not MDE specific, but is intended to cover and explain a wide spectrum of software development and design approaches.

and transformations; these toolsets have been used in the Airbus static architecture plane model [33]. So the basic ideas of MDELite (using Prolog) do indeed scale to large systems.

On a panel discussion held during the Educators Symposium at MODELS 2009, Bézivin presented a statement entitled "We Need an Army of Engineers to Implement the MDE Vision". He described how teaching MDE to future engineers is key to realizing the vision [11]. The same belief is behind this work. However, our experience coincides with France's insights at the same symposium in 2011, i.e., that many existing modeling tools introduce significant accidental complexity and that dissatisfaction with current toolset is sometimes the basis for students dismissing modeling techniques [22]. Hence, the seed for MDELite was born.

A paper by Favre inspired our work [20]. He warned against adding complex technologies on top of already complex technologies, and advocated a back-to-basics approach, specifically suggesting that MDE be identified with set theory and the use of Prolog to express MDE relationships among models and their meta-model counterparts.

In searching the literature, we found many papers advocating Prolog-database interpretations of MDE. For lack of space, we concentrate on the most significant, although we feel none are quite as compact or as clean as MDELite. Almendros-Jiménez and Iribarne advocated Prolog to write model transformations and model constraints [1,3]. The difference between our work and theirs is orientation: our goal is to find a simple way to demonstrate and teach MDE to undergraduates. Their goal is to explore the use of logic programming languages in MDE applications. For example, PTL is a hybrid of the Atlas Transformation Language and Prolog for writing model transformations [2]. In another paper, OWL files encode MDE databases and OWL RL specifies constraints in terms of Description Logics. For teaching undergraduates, the use of OWL and Description Logic is overkill and obscures the simplicity of MDELite. How M2T transformations are handled and MDE applications (categories) are encoded are not discussed.

Störrle's Model Manipulation Toolkit uses unnormalized (set-valued) relational tables as the basic Prolog data representation and uses Prolog to query these tables [37]. Although M2M transformations seem not to be discussed, the obvious implication is present. MDELite goes beyond this work also integrating M2T and T2M transformations, as well as exposing the bigger picture of MDE applications as categories.

Oetsch et. al. advocate *Answer-Set Programming (ASP)* to express a limited form of MDE [31]. Entity-Relationship models represent meta-models (drawn using Eclipse MDE tools); and their tool allows one to enter ASP facts (similar to Prolog facts) manually that conform to the input meta-models; ASP queries are used to validate meta-model constraints expressed in the ER model.[14] MDELite is more general than this: M2M, M2T, and T2M mappings need to be defined in addition to model constraints. Further, how MDE applications are defined (as in MDELite categories) is not considered.

[14] The Eclipse OCL tool plugin is similar in that one has to manually enter tuples beforehand before OCL queries can be executed. This is impractical, even for classroom settings.

To our knowledge, few papers have reported rigorous evaluations of MDE teaching experiences. Brosch et al. report their experience teaching an advanced modeling course called Model Engineering at the Vienna University of Technology [14]. We had the same experience with student complaints concerning the maturity of MDE technologies and lack of documentation. The authors state that they would like to develop a dedicated model engineering framework for teaching purposes. This is exactly the role of MDELite.

13 Conclusions

MDELite reinterprets MDE from the viewpoint of relational databases. A metamodel is a database schema with declarative constraints written in Prolog. A model is a database of tuple-populated-tables that satisfy metamodel constraints. Declarative M2M transformations are written in Prolog. M2T and T2M transformations rely on simple Java programs or off-the-shelf tools. Categories, a fundamental structure in mathematics, integrates these concepts to define MDE applications. MDELite leverages (and maybe introduces or refreshes) core undergraduate CS knowledge to explain, illustrate, and build MDE applications without the overhead and complexity of Eclipse MDE tools. Our case studies indicate MDELite is feasible; our initial user study supports our observations on MDELite as a good platform for teaching MDE concepts.

The next phases of our research, which we have already begun, is to bootstrap MDELite. We have already explored several possibilities, but have not yet settled on a way that exhibits both simplicity and usability within an IDE framework. Improving MDE teaching material/descriptions and building MDE tools are separable concerns: MDELite has a conceptual compactness and elegance that MDE should have. Finding a correspondingly pleasing IDE framework remains our top open problem.

We believe MDELite is a clarion way to explain MDE to undergraduate students. It is our hope that others may benefit, and indeed improve, our ideas. MDELite is available at http://www.cs.utexas.edu/users/schwartz/MDELite/index.html.

Acknowledgements. We thank the SoSyM referees for very constructive reviews of our initial draft; in a time where writing unconstructive, uninformed negative reviews has become a sport, these referees took their jobs seriously, for which we are grateful.

We are indebted to S. Trujillo (Ikerlan), O. Diaz (UPV/EHU), P. Stevens (Edinburgh), J. Siegmund (Passau) and L. Vozmediano (UPV/EHU) for their insightful comments on earlier drafts of this paper. We also appreciate the help given by M. Spönemann on the Kieler graph layout tools and R. Lämmel his invaluable help answering questions about Prolog. We also thank R. Berg, E. Huneke, A. Shali, and J. Ho for creating VM2T and thank J. Croy, D. Ilijev, B. Koo, E. Liu, T. McCandless, M. Parikh, C. Orchard, C. Salisbury, A. Sharp, J. Siu, C. Stewart, and M. Teng for participating in the user evaluation. We gratefully acknowledge support for this work by NSF grants CCF-1421211, CCF-0724979, and OCI-1148125, the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839.

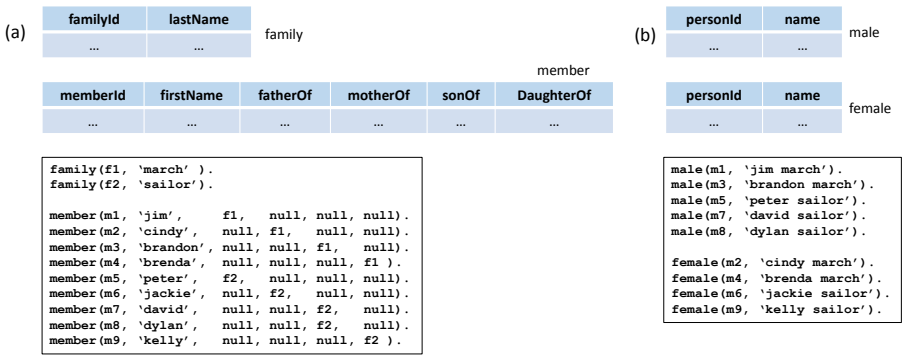
References

1. Almendros-Jiménez, J.M., Iribarne, L.: A Framework for Model Transformation in Logic Programming. In: VIII Jornadas sobre Programación y Lenguajes (PROLE 2008), Gijón, Spain. pp. 29–39 (2008)
2. Almendros-Jiménez, J.M., Iribarne, L.: A Model Transformation Language Based on Logic Programming. In: 39th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2013), Špindlerův Mlýn, Czech Republic. Lecture Notes in Computer Science, vol. 7741, pp. 382–394. Springer (2013)
3. Almendros-Jiménez, J.M., Iribarne, L.: ODM-based UML Model Transformations using Prolog. In: International Workshop on Model-Driven Engineering, Logic and Optimization: friends or foes? (MELO 2011), in conjunction with the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011), Birmingham, UK. Lecture Notes in Computer Science, vol. 7741, pp. 382–394. Springer (2013)
4. Apache Velocity Project. `http://velocity.apache.org/`. Last accessed: April 2015
5. Basili, V.R.: Software Modeling and Measurement: the Goal/Question/Metric Paradigm. Tech. rep., University of Maryland (1992)
6. Batory, D.: Multilevel Models in Model-driven Engineering, Product Lines, and Metaprogramming. IBM Systems Journal 45(3), 527–540 (2006)
7. Batory, D., Azanza, M., Saraiva, J.: The Objects and Arrows of Computational Design. In: 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008), Toulouse, France. Lecture Notes in Computer Science, vol. 5301, pp. 1–20. Springer (2008)
8. Batory, D., Gonçalves, R., Marker, B., Siegmund, J.: Dark Knowledge and Graph Grammars in Automated Software Design. In: 6th International Conference on Software Language Engineering (SLE 2013), Indianapolis, IN, USA. Lecture Notes in Computer Science, vol. 8225, pp. 1–18. Springer (2013)
9. Batory, D., Latimer, E., Azanza, M.: Teaching Model Driven Engineering from a Relational Database Perspective. In: 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2013), Miami, FL, USA. Lecture Notes in Computer Science, vol. 8107, pp. 121–137. Springer (2013)
10. Baughman, M.: The Influence of Scientific Research and Evaluation on Publishing Educational Curriculum. New Directions for Evaluation 117, 85–94 (2008)
11. Bézivin, J., France, R.B., Gogolla, M., Haugen, Ø., Taentzer, G., Varró, D.: Teaching Modeling: Why, When, What? In: Workshops and Symposia at the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, CO, USA, Reports and Revised Selected Papers. Lecture Notes in Computer Science, vol. 6002, pp. 55–62. Springer (2009)
12. Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: OOPSLA/GPCE Workshop on Best Practices for Model-Driven Software Development (MDSD 2004) in conjuction with the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, Canada (2004)
13. Boruch, R.: Encouraging the Flight of Error: Ethical Standards, Evidence Standards, and Randomized Trials. New Directions for Evaluation 2007, 55–73 (2007)
14. Brosch, P., Kappel, G., Seidl, M., Wimmer, M.: Teaching Model Engineering in the Large. In: 5th Educators' Symposium in conjunction with the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, CO, USA (2009)
15. Catalina: A Next-Generation of MDELite. `http://www.cs.utexas.edu/users/schwartz/MDELite/index.html`. Last accessed: April 2015
16. Cook, T.D.: Why have Educational Evaluators Chosen Not to Do Randomized Experiments? The ANNALS of the American Academy of Political and Social Science 589(1), 114–149 (2003)
17. Dehayni, M., Féraud, L.: An Approach of Model Transformation Based on Attribute Grammars. In: 9th International Conference on Object-Oriented Information Systems (OOIS 2003), Geneva, Switzerland. Lecture Notes in Computer Science, vol. 2817, pp. 412–423. Springer (2003)
18. Diskin, Z.: Algebraic Models for Bidirectional Model Synchronization. In: 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008), Toulouse, France. Lecture Notes in Computer Science, vol. 5301, pp. 21–36. Springer (2008)
19. Dot Language. `http://www.graphviz.org/content/dot-language`. Last accessed: April 2015
20. Favre, J.M.: Towards a Basic Theory to Model Driven Engineering. In: 3rd Workshop in Software Model Engineering (WISME 2004) in conjuction with the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal (2004)

21. Fraenkel, J.R., Wallen, N.E.: How to Design and Evaluate Research in Education. McGraw-Hill (2009)
22. France, R.B.: Teaching Programming Students how to Model: Challenges & Opportunities. In: Keynote Speaker at the 7th Educators' Symposium in conjunction with the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011), Wellington, New Zealand (2011)
23. Hainaut, J.: The Transformational Approach to Database Engineering. In: 1st International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005), Braga, Portugal. Revised Papers. Lecture Notes in Computer Science, vol. 4143, pp. 95–143. Springer (2005)
24. Handlebars Template Project. http://handlebarsjs.com/. Last accessed: April 2015
25. ISO/IEC: Software Engineering - Software Product Quality - Part 1: Quality Model (2001)
26. Jedlitschka, A., Pfahl, D.: Reporting Guidelines for Controlled Experiments in Software Engineering. In: International Symposium on Empirical Software Engineering (ISESE 2005), Noosa Heads, Australia. pp. 95–104. IEEE (2005)
27. Johnson, P.: Human Computer Interaction: Psychology, Task Analysis, and Software Engineering. McGraw-Hill (1992)
28. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming (SCP) 72(1-2), 31–39 (2008)
29. Kieler Web Service Tool. http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Downloads+-+KIELER+Web+Service+Tool. Last accessed: April 2015
30. McKeachie, W.J.: Research on College Teaching: The Historical Background. Journal of Educational Psychology 82, 189–200 (1990)
31. Oetsch, J., Pührer, J., Seidl, M., Tompits, H., Zwickl, P.: VIDEAS: A Development Tool for Answer-Set Programs Based on Model-Driven Engineering Technology. In: 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, Canada. Lecture Notes in Computer Science, vol. 6645, pp. 382–387. Springer (2011)
32. Oldevik, J.: UMT: UML Model Transformation Tool Overview and User Guide Documentation. http://umt-qvt.sourceforge.net/docs/UMT_documentation_v08.pdf. Last accessed: April 2015 (2004)
33. Pantel, M.: Private Email Conversation (2013)
34. Pierce, B.: Basic Category Theory for Computer Scientists. MIT Press (1991)
35. Shadish, W., Cook, T., Campbell, D.: Experimental and Quasi-experimental Designs for Generalized Causal Inference. Cengage Learning (2002)
36. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling: State of the Art and Research Challenges. In: Model-Based Engineering of Embedded Real-Time Systems. International Dagstuhl Workshop, Dagstuhl Castle, Germany, 2007. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6100, pp. 57–76. Springer (2010)
37. Störrle, H.: A prolog-based approach to representing and querying software engineering models. In: Workshop on Visual Languages and Logic (VLL 2007) in conjunction with the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2007), Coeur d'Aléne, Idaho, USA. CEUR Workshop Proceedings, vol. 274, pp. 71–83. CEUR-WS.org (2007)
38. SWI-Prolog. http://www.swi-prolog.org/. Last accessed: April 2015
39. UML Factory. http://www.umlfactory.com/. Last accessed: June 2012
40. Violet UML Editor. http://alexdp.free.fr/violetumleditor/page.php. Last accessed: April 2015
41. Walser, T.M.: Quasi-Experiments in Schools: The Case for Historical Cohort Control Groups. Practical Assessment, Research and Evaluation 19(6), 1–7 (2014)
42. yUML Beta. http://yuml.me/

A Families$_2$Persons Application

Families$_2$Persons is a simple M2M application. A family database, consisting of two tables `family` and `member`, is transformed into a persons database, consisting of the `male` and `female` tables. Figure 26a-b shows the schema and tuples of the family and persons databases, respectively. The M2M rules in Prolog are given in Figure 26c.




```
family(f1, 'march' ).
family(f2, 'sailor').

member(m1, 'jim',     f1,   null, null, null).
member(m2, 'cindy',   null, f1,   null, null).
member(m3, 'brandon', null, null, f1,   null).
member(m4, 'brenda',  null, null, null, f1 ).
member(m5, 'peter',   f2,   null, null, null).
member(m6, 'jackie',  null, f2,   null, null).
member(m7, 'david',   null, null, f2,   null).
member(m8, 'dylan',   null, null, f2,   null).
member(m9, 'kelly',   null, null, null, f2 ).
```

```
male(m1, 'jim march').
male(m3, 'brandon march').
male(m5, 'peter sailor').
male(m7, 'david sailor').
male(m8, 'dylan sailor').

female(m2, 'cindy march').
female(m4, 'brenda march').
female(m6, 'jackie sailor').
female(m9, 'kelly sailor').
```

(c)

```
% utility needed to concatenate atoms with blank in between

concatnames(X,Y,Z) :-
   string_to_atom(Sx,X),string_to_atom(Sy,Y),
   string_concat(Sx,' ',XX),string_concat(XX,Sy,Z).

% here are the database translation rules

male(Id,N)   :- family(F,Ln),member(Id,Fn,F,_,_,_),concatnames(Fn,Ln,N).
male(Id,N)   :- family(F,Ln),member(Id,Fn,_,_,F,_),concatnames(Fn,Ln,N).
female(Id,N) :- family(F,Ln),member(Id,Fn,_,F,_,_),concatnames(Fn,Ln,N).
female(Id,N) :- family(F,Ln),member(Id,Fn,_,_,_,F),concatnames(Fn,Ln,N).
```

Fig. 26 Families to Persons.

B An Illustrative M2M Example

Consider Figure 27. (a) shows an aggregation hierarchy: each `course` has a number of `offerings`, each `offering` has `students` and `instructors`. Using the rules of Section 2, the schema of (b) is produced from (a). The primary or partial keys of each table are indicated in (a) by '`Key`' and are underlined in (b). To map this to a proper set of relational tables, where MDELite internal identifiers are absent and only primary or compound keys are present in tuples as in (c), we need a M2M transformation.

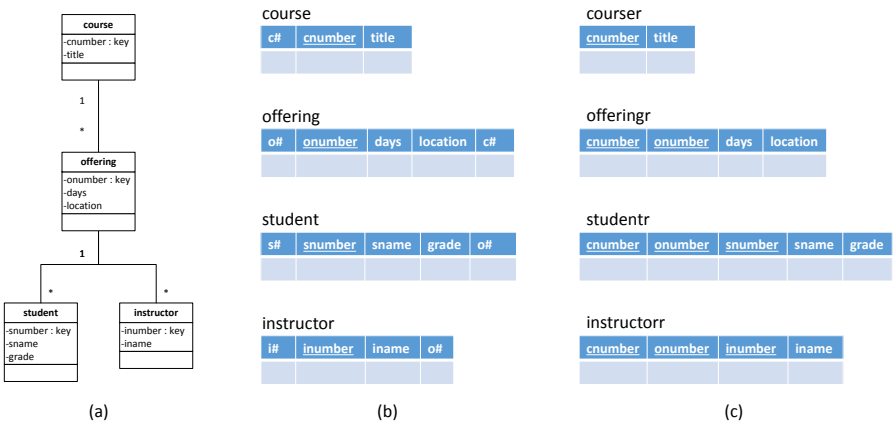


Fig. 27 Database Normalization Transformation.

The Prolog program that translates schema (b) to (c) is listed below:

```
/* MDELite database and schema declarations */

dbase(one,[course,offering,student,instructor]).
dbase(two,[courser,offeringr,studentr,instructorr]).

table(course,[cid,cnumber,title]).
table(offering,[oid,onumber,days,location,cid]).
table(student,[sid,snumber,sname,grade,oid]).
table(instructor,[iid,inumber,iname,oid]).

table(courser,[cnumber,title]).
table(offeringr,[cnumber,onumber,days,location]).
table(studentr,[cnumber,onumber,snumber,sname,grade]).
table(instructorr,[cnumber,onumber,inumber,iname]).


/* table translation rules */

courser(C,T) :- course(_,C,T).

offeringr(C,O,D,L)    :- offering(_,O,D,L,Cid),course(Cid,C,_).

studentr(C,O,S,N,G)   :- student(_,S,N,G,Oid),offering(Oid,O,_,_,Cid),course(Cid,C,_).

instructorr(C,O,I,N)  :- instructor(_,I,N,Oid),offering(Oid,O,_,_,Cid),course(Cid,C,_).
```

C Illustrative VM2T Application

The VM2T script that converts an FSM Prolog database into executable source is given below.

```
#set($MARKER="//----")
${MARKER}src/FSM.java
##
## FSM code
##
class FSM {
   state current = new start();

#foreach( $n in $nodeS)
   void goto$n.name()
   { current = current.goto$n.name(); }

#end
   String getName()
   {   return current.getClass().getName();  }

}
##
## state interface
##
${MARKER}src/state.java
interface state {

#foreach( $n in $nodeS)
   state goto$n.name();
#end
   String getName();

}
##
## node classes
##
#foreach ( $n in $nodeS )
${MARKER}src/$n.name.java
class $n.name implements state {

  #foreach ( $s in $nodeS )
     #set ($legal = false)
     #foreach ( $e in $edgeS )
        #if ($e.startsAt == $n.id && $e.endsAt == $s.id)
           #set ($legal = true )
           #set ($end  = $s.name )
        #end
     #end
     #if ($legal)
  public state goto$s.name() { return new $end(); }
     #else
  public state goto$s.name() {  return this; /* ignore */ }
     #end
  #end

  public String getName() { return "$n.name"; }

}
#end
```

The Java code for class `drink` is shown below:

```
class drink implements state {

  public state gotostart() {  return this; /* ignore */ }
  public state gotoready() {  return this; /* ignore */ }
  public state gotodrink() { return new drink(); }
  public state gotoeat() { return new eat(); }
  public state gotopig() { return new pig(); }
  public state gotostop() {  return this; /* ignore */ }

  public String getName() { return "drink"; }

}
```