Teaching Model Driven Engineering from a Relational Database Perspective

Don Batory[1], Eric Latimer[1], Maider Azanza[2]

[1] University of Texas at Austin, Austin, TX 78712 USA
batory@cs.utexas.edu, e@utexas.edu
[2] University of the Basque Country (UPV/EHU), San Sebastian, Spain
maider.azanza@ehu.es

Abstract. We reinterpret MDE from the viewpoint of relational databases to provide an alternative way to teach, understand, and demonstrate MDE using concepts and technologies that should be familiar to undergraduates. We use (1) relational databases to express models and metamodels, (2) Prolog to express constraints and M2M transformations, (3) Java tools to implement M2T and T2M transformations, and (4) OO shell-scripting languages to compose MDE transformations. Case studies demonstrate the viability of our approach.

1 Introduction

Model Driven Engineering (MDE) is a standard technology for program specification and construction. We believe it is essential to expose undergraduates to MDE concepts (models, metamodels, M2M, M2T, T2M transformations, constraints, and bootstrapping), so that they will have an appreciation for MDE when they encounter it in industry. Our motivation was experience: unless students encounter an idea (however immature) in school, they are less likely to embrace it in the future. *Further, teaching MDE is intimately related, if not inseparable, to the tools and languages that make MDE ideas concrete.*

Our initial attempt to do this (Fall 2011) was a failure. We used the Eclipse Modeling Tools[3] and spent quite some time creating videos for students to watch, both for installation and for tool usage. For whatever reason, installation for students was a problem. A version of Eclipse was eventually posted that had all the 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 this 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 will not try this again.*

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

In retrospect we found many reasons, but basically Eclipse MDE tools are the problem. (1) The tools we used were unappealing—they were difficult to use even for trivial 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. (3) With the benefit of years of hindsight, we concluded that the entry cost of using, teaching, and understanding these tools was too high for our comfort. (Whether students agree with this or not is the subject of an empirical study targeted for this fall). 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, and bootstrapping). We tried this approach with a new class of undergraduates in Fall 2012 with many fewer problems. (Again, we carefully avoid words like "better" or "more successful" until the results of our empirical study are in; the appropriate word to use is "interesting"). This paper concentrates on the technology we used and the case studies in its evaluation). It is our hope that others in MDE may benefit from the simplicity of our approach.

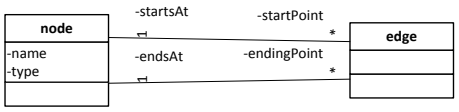
2 MDE Models and MetaModels

MDE can be understood as an application 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 schemas of the underlying relational tables (using manufactured identifiers, denoted by `node#` and `edge#`) are shown in Figure 1b.

An instance of a FSM populates these tables with tuples. The FSM of the first author's eating habits and its tuples are given in Figure 1c-d.

(a)

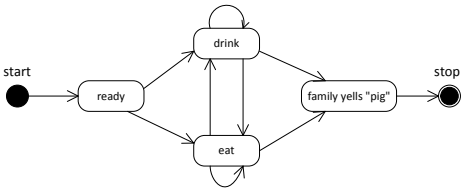


(b)

node#	name	type

edge#	startsAt	endsAt

(c)



(d)

node#	name	type
nStart	start	start
nReady	ready	state
nDrink	drink	state
nEat	eat	state
nPig	pig	state
nStop	stop	stop

edge#	StartsAt	EndsAt
e1	nStart	nReady
e2	nReady	nDrink
e3	nReady	nEat
e4	nDrink	nDrink
e5	nEat	nEat
e6	nDrink	nEat
e7	nEat	nDrink
e8	nDrink	nPig
e9	nEat	nPig
e10	nPig	nStop

Fig. 1. A State Machine and its Tables

Manufactured tuple identifiers eliminate virtually all of the complexities of relational table design (*c.f.* [8,12]). 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 the identifier and one for each attribute.
2. `n : m` associations are valid in metamodels [17], but not in ours. Every association must have an 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.

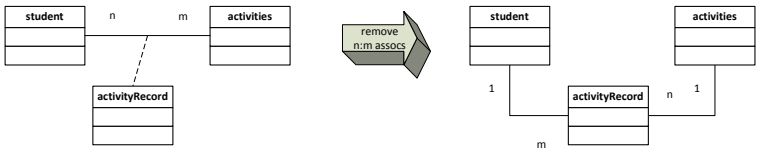


Fig. 2. Transformation That Removes `n : m` Associations.

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 "`1`" side has a unary-valued attribute (a tuple identifier) which is permitted. As both attributes encode exactly the same information, we simply drop the set-valued attribute. Figure 3a illustrates the application of the last three rules: the `dept` table has two columns (# and `name`) and the `student` table has three (#, `utid`, and `enrolledIn`). Column `enrolledIn`, which contains a `dept#` value, represents the `student` − `dept` association. The mapping of Figure 1a to 1b is another example.

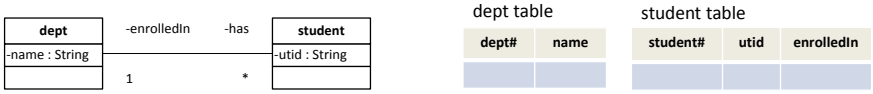


Fig. 3. Diagram-to-Table Mapping.

4. For classes that are related by inheritance, all attributes of superclasses are propagated into the class tables. The identifier of the root class is shared by all subclasses. Tables need not be produced for abstract classes. See Figure 4.

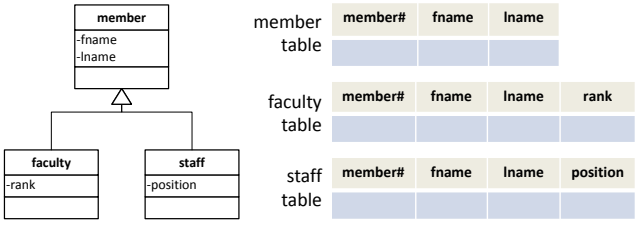


Fig. 4. Inheritance Diagram-to-Table Mapping.

5. Only objects of a class that are not instances of subclasses populate the tuples of a table. This rule is is discussed in more detail in Section 4.

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

Note that relational tables have always been able to encode data hierarchies. We see the elegance of normalized or "flat" tables to be an important conceptual simplicity.

3 Model Constraints

OCL is the standard language for expressing model constraints. Given the connection to relational databases, we can do better. 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 5a shows how to express tuples of a relational table as Prolog facts. The first fact in Figure 5a defines the schema of the `node` table of Figure 1b: it has three columns {`id`, `name`, `type`}.

(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)

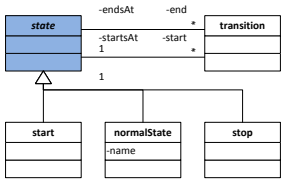


Fig. 5. A Prolog Table and Target MetaModel.

Here are three constraints to enforce on a FSM:

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

Their expression in SWI-Prolog [19] 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(A,N,_),node(B,N,_),not(A=B),error('non-unique names').
c2 :- edge(_,S,E), ( not(node(_,S,_)) ; not(node(_,E,_)) ), error('bad edge').
c3a :- not(node(_,_,start)), error('no start state').
c3b :- node(A,_,start),node(B,_,start),not(A=B),error('multiple start states').
allConstraints :- not(c1),not(c2),not(c3a),not(c3b).
```

4 Model-to-Model Transformations

Fundamental activities in MDE are *model-to-model (M2M)* transformations. Instead of using languages that were specifically invented for MDE, Prolog can be used to write database-to-database (or M2M) transformations declaratively.

Suppose we want to translate the database of Figure 1d to a database that conforms to the metamodel of Figure 5b. (We shade abstract classes to make them easier to recognize.) The Prolog rules to express this transformation are:

```
start(I,A) :- node(I,A,start).
stop(I,A) :- node(I,A,stop).
normalState(I,A) :- node(I,A,state).
transition(A,B,C) :- edge(A,B,C).
```

Another example: The tuples of the `staff` and `faculty` tables of Figure 4 do not appear in the `member` table. To propagate tuples from subclass tables into superclass tables, the following transformations can be used:

```
newMember(I,F,L) :- member(I,F,L).
newMember(I,F,L) :- staff(I,F,L,_).
newMember(I,F,L) :- faculty(I,F,L,_).
newStaff(I,F,L,R) :- staff(I,F,L,R).
newFaculty(I,F,L,P) :- faculty(I,F,L,P).
```

As Prolog is Turing-complete, database transformations can be arbitrarily complex.

Observations. There is an intimate connection between database design and metamodel design. Presenting MDE in the above manner reinforces this connection. Further, 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 obvious 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 code. This is accomplished by *Model-to-Text (M2T)* transformations. There are many text template engines used in industry. Apache Velocity is a particularly easy-to-learn and powerful example [4]. We made two small modifications to Velocity to cleanly integrate it with Prolog databases. Our tool is called *Velocity Model-2-Text (VM2T)*.

First, we defined Velocity variables for tables. If the name of a table is "`table`" then the table variable is "`tableS`" (appending an "`S`" to "`table`"). This enables a Velocity `foreach` statement to iterate over all tuples of a table:

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

Figure 6a shows a metamodel for classes. Two instances of this metamodel, `city` and `account`, are shown in Figure 6b. The database containing both instances is Figure 6c.

Figure 7a is a VM2T template. When the non-`MARKER` statements are executed, Figure 7b is the output. Perferably, the definition of each class should be in its own file. When all statements are executed, the desired two files are produced (Figure 7c).

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 8.

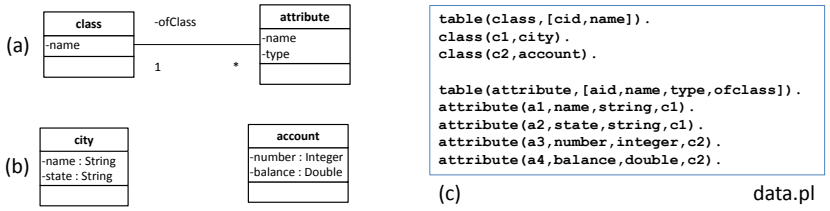


Fig. 6. A Class Metamodel, a Model, and its Prolog Database.

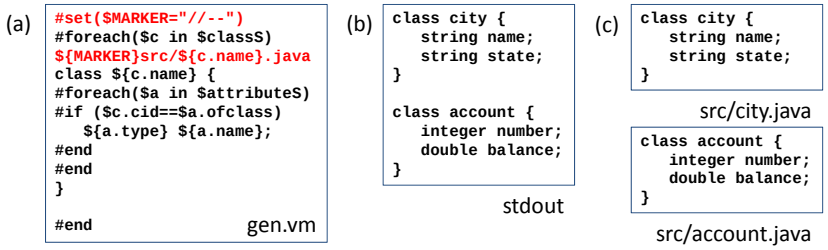


Fig. 7. A VM2T Template and Two Outputs.

Observations. 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. In our opinion, it is easy to learn and relatively painless to use.

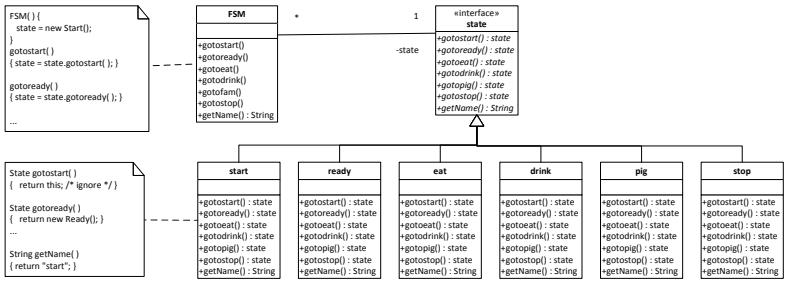


Fig. 8. Class Diagram of FSM Code Output.

6 Text-to-Model Transformations

Given the above, it is not difficult for students to understand Figure 9: an application engineer specifies a FSM using a graphical tool, the tool produces a set of tables, the tables are transformed, and VM2T produces the source code for the FSM.

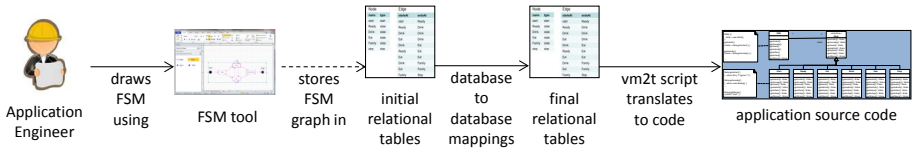


Fig. 9. FSM Application Engineering in MDE.

What is missing is a *Text-to-Model (T2M)* transformation (the dashed arrow in Figure 9) that converts grossly-verbose XML output of a graphics tool into a clean set of Prolog tables. It is easy to write a simple Java program that reads XML, parses it, and outputs a single text file containing a Prolog database. Using a more general tool that parses XML into Prolog may be preferable, but loses the advantage a hands-on understanding of the inner workings of T2M transformations.

Finding suitable *graphical editor* $\mathcal{GE}$ is a three-fold challenge:

(a) its XML must simple to understand,
(b) its XML is stable, meaning its XML format is unlikely to change anytime soon, and
(c) its palette[4] is customizable.

MS Visio is easy to use and its palette is easily customizable, but its XML files are incomprehensible and MS periodically modifies the format of these files. Simpler $\mathcal{GE}$s, such as Violet [21], yUML [22], UMLFactory [20], 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 per UML diagram. One cannot define a set of icons (with graphic properties) to draw customized graphs. All one can do is to translate XML documents that were specifically designed for a given UML diagram to Prolog tables. This isn't bad; it just isn't ideal. Until a flexible $\mathcal{GE}$ is found, bootstrapping MDELite (to build customized $\mathcal{GE}$s for target domains, a key idea in MDE) is difficult to demonstrate. More on this in Section 9.

Observations. 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 can be 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 document, translating the document into Prolog tables, evaluating Prolog constraints, and displaying the errors encountered. Modifying existing tools to present this feedback could be done, but this is not high-priority.

[4] The icons/classes that one can drag and drop onto a canvas to create instances.

7 MDELite and its Applications

MDELite is a small set of tools (SWI Prolog, VM2T) that are loosely connected by a tiny Java framework that implements the ideas of the prior section. An *MDELite application* uses this framework and is expressed as a category [5,16]. A *category* is simply a directed multigraph; nodes are *domains* and *arrows* are 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 (e.g., [9]), and they define the structure of an MDE application.[5]

As an example of an MDELite application, consider the tool chain that allows users to draw FSMs and generate their corresponding Java source (Figure 9). This tool chain is a category with four domains (Figure 10): the domain of XML documents that are output by the FSM tool, 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 Source Code whose elements are FSM programs.

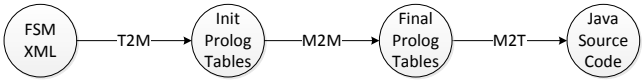
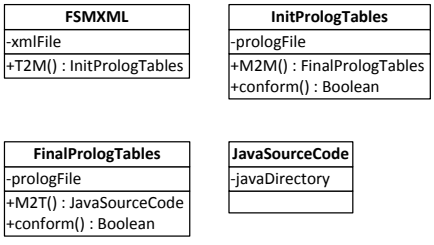


Fig. 10. Category of a FSM Tool.

When this category is written in Java, each domain is a class and each arrow is a method (see Figure 11a). Unlike most UML class diagrams, MDELite designs typically have no associations, but can have inheritance relationships.

To perform an action of the FSM tool (*i.e.* a method in Figure 11a), one writes a straight-line script to invoke the appropriate transformations and checks. Figure 11b shows the sequence of method calls in an MDELite program to translate an FSMXML file—an XML file produced by the FSM drawing tool—into a Java program. Any error encountered during translation or conformance test simply halts the MDELite application with an explanative message.

(a)



(b)

```
JavaSourceCode tool(FSMXML x)
  throws RunTimeException {
  ipt = x.T2M();
  ipt.conform();
  fpt = itp.M2M();
  fpt.conform();
  jsc = ftp.M2T();
  return jsc;
}
```

Fig. 11. MDELite Encoding of the Category of Figure 10.

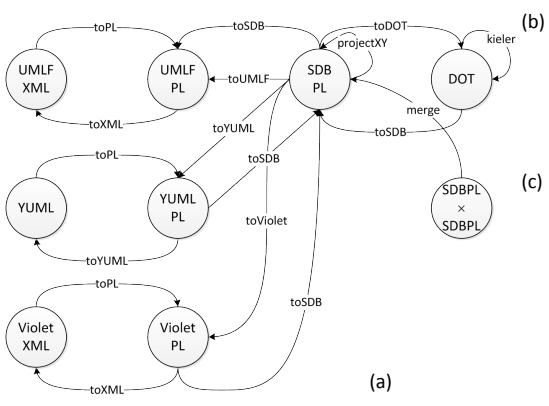
[5] Also known as *megamodels* [7] and *tool chain diagrams* [15].

Observations. *MDE lifts metamodel design to the level of* metaprogramming*—programs that build other programs.* 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 [6]. 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. Perhaps MDELite needs to be written in an OO shell scripting language, such as Python. We used Java to implement the MDELite framework (and may reconsider this decision—we figured Prolog is enough for undergraduates to absorb). MDELite is clearly a multi-lingual application.

8 Evaluation: A Case Study of MDELite

Our first application of MDELite was quite instructive. We found several free UML tools that we wanted to (i) draw UML class diagrams, (ii) apply the ideas of the previous sections, and (iii) integrate.

The integration of the Violet, UMLFactory, and yUML tools (as they existed in June 2012) is expressed by the category of Figure 12a.[6] We could draw UML class diagrams in each of these tools and have them displayed in any other tool. So a script that translated a Violet class diagram into a yUML class diagram is Figure 12b and vice versa is Figure 12c. Figure 13 shows the translation of a specific Violet class diagram (an XML file) into an SDBPL database and then into a yUML class diagram (a yUML file).



(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.projectXY();
  sdb3 = sdb2.merge(sdb1);
  upl = sdb3.toUMLF();
  upl.conform();
  return upl.toXML();
}
```

Fig. 12. A Category for an MDELite Application.

The category of Figure 12a is produced by a process that is similar to global schema design in databases that integrates database schemas of different tools [10]. Each tool exports and imports a distinct data format (read: database). A global schema (a Prolog

[6] The only oddity of Figure 12a 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).

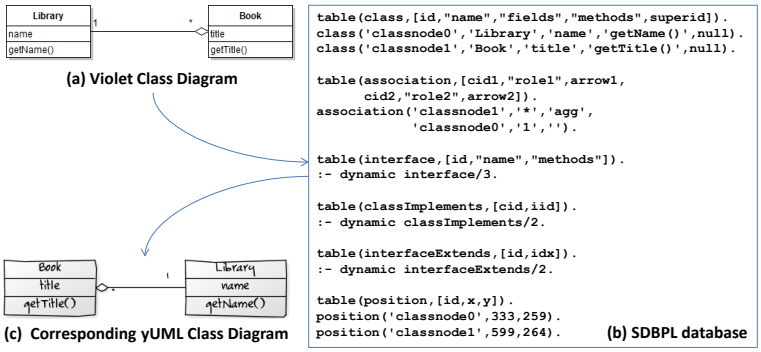


Fig. 13. A Violet Diagram mapped to an SDBPL database mapped to a yUML Diagram.

database, SDBPL, to which all tool-specific databases are translated) stores data that is shared by all tools. The hard part is manufacturing data that is not in the global database that is needed for tool-specific displays. An example is given shortly.

This application required all kinds of T2M, M2T, and M2M transformations. Figure 14 shows 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. 14. Size of MDELite Framework and Application: Lines of Prolog, Velocity, and Java Code

Observations. 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; most 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.

Interesting technical problems also arise. A yUML spec for Figure 13c is:

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

Translating a yUML spec to the XML document of another tool requires graphical $(\mathtt{x},\mathtt{y})$ positioning information about 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 Web Service [13]. We translated an SDBPL database into a DOT graph, transmited the DOT file to the Kieler server, and it returned a new DOT graph with the required positioning information. A simple T2M tool mapped the positioning information to a Prolog table, and this table was merged with a SDBPL database that lacked positioning information (as indicated in the Figure 12c script). Only then was a usable Violet file produced. Figure 15a shows the generated DOT file, Figure 15b the DOT file returned by the Kieler server, and Figure 15c the T2M extracted `position` table.

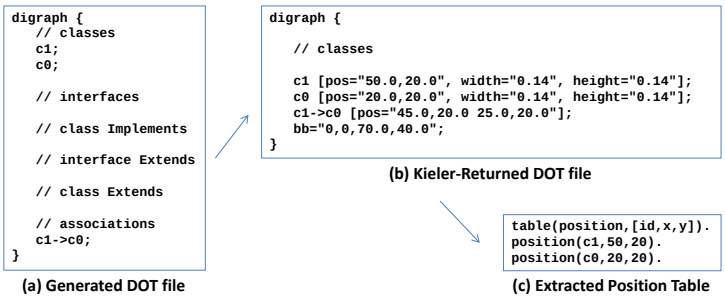


Fig. 15. DOT File Transformations.

9 Towards Bootstrapping

Although we have not fully bootstrapped MDELite for reasons discussed earlier, there are two basic steps to produce the FSM tool or any other domain-specific MDE tool.

First, we need to specify how metaclass instances are to be drawn by the $\mathcal{GE}$. The simplest way is to allow the $\mathcal{GE}$ to set properties of each metaclass to provide the necessary information. For example, Figure 16 uses stereotypes to declare that a `State` is to be drawn as an oval, except a `Start` state is a solid-circle and a `Stop` state is a double-circle (c.f. Figure 1). Other ways to encode this information are also possible.

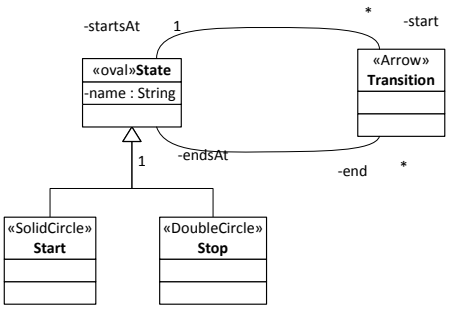


Fig. 16. FSM Metamodel with Graphical Stereotypes

Second, look at Figure 17. A FSM domain architect would (1) draw the FSM metamodel using a *Metamodel Drawing Tool (`MDT`)*, which mechanizes the rules of Section 2 to produce Prolog table definitions for the input metamodel and a palette of icon-metaclass pairings to customize the $\mathcal{GE}$, (2) write the Prolog metamodel constraints, the Prolog M2M transformations, and a Velocity M2T file, and (3) run a build script that integrates these inputs with a `MDE Tool Shell` to generate the FSM tool.

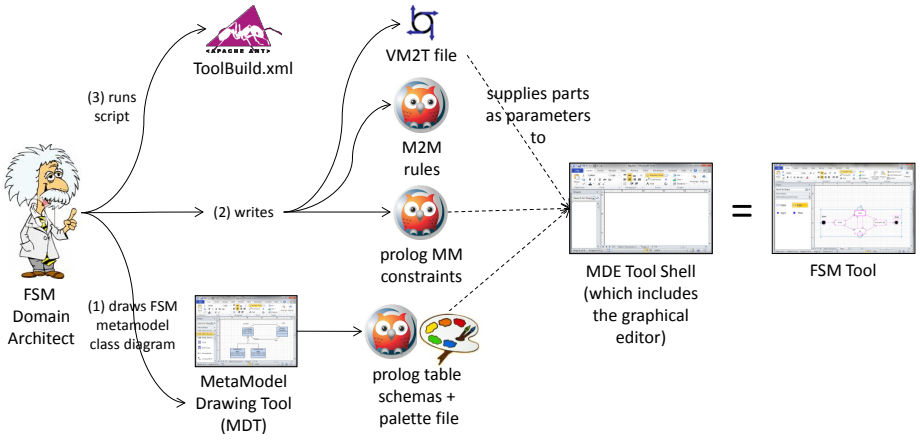


Fig. 17. Generating a MDE FSM Tool.

To bootstrap MDELite requires an MDE$\mathcal{G}$od to build the two tools (`MDT` and `MDE Tool Shell`) and script (`ToolBuild.xml`) that a Domain Architect (Einstein) invokes (see Figure 17). Specifically, MDE$\mathcal{G}$od writes the `ToolBuild.xml` script and purchases or outsources the writing of the `MDE Tool|Shell` (which includes the $\mathcal{GE}$). Initially the MDE$\mathcal{G}$od hacks a *MetaModel Drawing Tool (MDT)*. MDE$\mathcal{G}$od then relies on a fundamental MDE constraint that the MDELite meta-meta-model must be an instance of itself. So, the MDE$\mathcal{G}$od plays the role of a MetaModel Domain Architect, replacing Einstein in Figure 17 with him/herself. MDE$\mathcal{G}$od (1) draws the metamodel of all class diagrams, (2) writes its Prolog metamodel constraints, Prolog M2M transformations, and a VM2T file (which produces Prolog table schemas and a palette for drawing class diagrams from the Prolog database), (3) runs the build script to produce the `MDT` to complete the bootstrap, thereby building an `MDT` to replace the hacked `MDT`. Again, all of this hinges on finding a palette-customizable $\mathcal{GE}$.

10 Personal Experiences, Insights, and a Small Second Case Study

We created MDELite as an alternative to Eclipse MDE tools to understand and teach MDE concepts. Our work begs for an empirical study to evaluate the benefits of teaching MDELite; we intend to conduct such a study later this year. MDELite is an interesting technical contribution in its own regards, and that is what we focus on in this paper.

We used MDELite in a Fall 2012 undergraduate course on "Introduction to Software Design", giving an assignment more ambitious than what we tried in Fall 2011. Specifically, we asked students to:

1. Given a simple metamodel of class diagrams, manually produce the schemas of the metamodel's underlying Prolog tables;
2. Write a T2M transformation in Java using Java reflection to extract information about classes, methods, and fields from `.class` files and present this information as tuples in their tables;
3. Write Prolog constraints to evaluate the correctness of the tables they produced;

4. Write a Velocity M2T transformation that maps their tables into stubbed Java source;
5. Write another T2M transformation that converts Java reflection information to produce a yUML specification, which is then translated into a Violet diagram by MDELite; and
6. Extend the MDELite category (Figure 12) with the domains and arrows of Figure 18 by implementing the required classes and methods to script their transformations.

We can report many fewer difficulties with this assignment than the simpler assignment of the previous year that used Eclipse MDE tools. Still, there are some practical difficulties that we are obliged to alert readers.

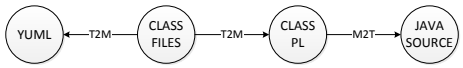


Fig. 18. Additional Domains and Arrows to Figure 12.

Multi-Paradigm Programming. We 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 we 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 to reset to the command prompt simply do not work. 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, recommend that MDELite be a pair-programming project: one person concentrating on Prolog, the other on Java, to minimize cross-world 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 not simple values and can have complex structures. *Writing transformations in any language is not simple*—it is easy to forget a case or miss-write 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.

Fig. 19. Debugging Transformation Scripts

A technique that we found useful—perhaps motivated by the "shape" of the category of Figure 12a—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.[7] This helped, but obviously did not eliminate all bugs.

Nonetheless, the fundamental challenge in debugging transformations becomes clearly evident: an error is detected in a database (far right of Figure 19). 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.

Preparatory M2M Transformations. 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).
```

In a M2T transformation, the class table must be joined (twice) with the association table to convert class identifiers (`c1` and `c2`) into class names (`Customer` and `Address`). Similarly, other computations can arise to convert atoms (like '`arrow`' above) into rendering text (in this case, the character '>' to denote an arrow). Such translations significantly complicate Velocity templates—it would not be so bad if one could indent Velocity statements to pair up the start and end of loops and if-statements:

```
#forall($a in $associationS)
   #forall($c in $classS)
      #if ($a.id = $c.id)
         #set($classname=$c.name)
      #end
   #end
#end
```

Indenting, however, generates extra spaces, which is not always desirable. The alternative is to produce a table of association declarations that render Velocity printing trivial:

```
yumlAssociation('Customer','*','','Billing Address','1','>').
```

Using M2M transformations can reduce the size (read: complexity) of Velocity files substantially. Although this is not a hard-and-fast heuristic, our experience is that keeping Velocity templates as simple as possible is worth the extra stage in Prolog translation.

11 Related Work

A paper by Favre inspired our work [11]. He warned against adding complex technologies on top of already complex technologies, and advocated a back-to-basics approach,

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

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 [2,3]. 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 [1]. 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 [18]. 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 [14]. 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.[8] 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.

12 Conclusions

MDELite reinterprets MDE from the viewpoint of relational databases. A model is a database of tables; (meta-)model constraints and M2M transformations are expressed by Prolog. M2T and T2M transformations rely on simple Java programs. 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; a user study to evaluate the benefits of MDELite in teaching is a next step in our work.

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/schwartz/MDELite.html`

[8] 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.

Acknowledgements. We am indebted to Salva Trujillo (Ikerlan), Oscar Diaz (San Sebastian), and Perdita Stevens (Edinburgh) for their insightful comments on earlier drafts of this paper. We also thank Robert Berg, Eric Huneke, Amin Shali, and Joyce Ho for VM2T. We also appreciate the help given to me by Miro Spönemann on the Kieler graph layout tools and Ralf Lämmel his invaluable help answering questions about Prolog. We gratefully acknowledge support for this work by NSF grants CCF 0724979 and OCI-1148125.

References

1. Almendros-Jimenez, J., Iribarne, L.: A model transformation language based on logic programming. In: SOFSEM 2013: Theory and Practice of Computer Science (2013)
2. Almendros-Jimenez, J., Iribarne, L.: A framework for model transformation in logic programming (2008)
3. Almendros-Jimenez, J., Iribarne, L.: Odm-based uml model transformations using prolog (2011)
4. Apache Velocity Project. http://velocity.apache.org/
5. Batory, D., Azanza, M., Saraiva, J.: The Objects and Arrows of Computational Design. In: MODELS (Oct 2008)
6. Batory, D.: Multilevel models in model-driven engineering, product lines, and metaprogramming. IBM Syst. J. (Jul 2006)
7. Bezivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proc. of the OOPSLA/GPCE Workshop on Best Practices for Model-Driven Software Development (2004)
8. Dehayni, M., Feraud, L.: An approach of model transformation based on attribute grammars. In: Konstantas, D., Leonard, M., Pigneur, Y., Patel, S. (eds.) Object-Oriented Information Systems, Lecture Notes in Computer Science, vol. 2817. Springer Berlin Heidelberg (2003)
9. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: MoDELS (2008)
10. Elmasri, R., Navathe, S.: Fundamentals of Database Systems. Addison-Wesley (2010)
11. Favre, J.M.: Towards a basic theory to model model driven engineering. In: Workshop on Software Model Engineering, WISME 2004 (2004)
12. Hainaut, J.L.: The transformational approach to database engineering. In: GTTSE (2006)
13. Kieler Web Service Tool. http://trac.rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Releases/Tools
14. 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: LPNMR (2011)
15. Oldevik, J.: Umt: Uml model transformation tool overview and user guide documentation. http://umt-qvt.sourceforge.net/docs/ (2004)
16. Pierce, B.: Basic Category Theory for Computer Scientists. MIT Press (1991)
17. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling: state of the art and research challenges. In: Proc. of the 2007 Dagstuhl Conference on Model-Based engineering of embedded real-time systems (2010)
18. Strrle, H.: A prolog-based approach to representing and querying software engineering models
19. SWI-Prolog. http://www.swi-prolog.org/
20. UML Factory. http://www.umlfactory.com/
21. Violet UML Editor. http://alexdp.free.fr/violetumleditor/page.php
22. yUML Beta. http://yuml.me/