MDELite Manual

Don Batory
batory@cs.utexas.edu
May 2017

1 Introduction

MDELite is an alternative to Eclipse for teaching and exploring concepts in *Model Driven Engineering* (MDE). Rather than:

- Storing models and metamodels as obscure XML documents, **MDELite** encodes them as readable relational databases expressed as elementary facts;
- Using *Object Constraint Language* (OCL) to express constraints, **MDELite** uses Java Streams;
- Writing *model-to-model* (M2M) transformations in the *Atlas Transformation Language* (ATL), an outgrowth of OCL, **MDELite** relies on Java; and
- Using yet another language/tool for *model-2-text* (M2T) translations, **MDELite** again relies on Java.

Here is an overview of this manual:

- Installation of **MDELite**
- **MDELite**-Relational Schemas
- **MDELite**-Relational Databases
- **MDELite** Tools

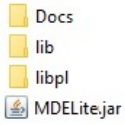
Note: Even with my best effort, I know there are bugs in **MDELite**. Please let me know when you find them `dsb`.

2 Installation

You can download **MDELite** from:

`www.cs.utexas.edu/users/schwartz/MDELite/index.html`

The **MDELite** directory and executable contains:



- `Docs` – documentation, including this manual,
- `lib` – a library of jar files needed by **MDELite**,
- `libpl` – a library of predefined schemas, and
- `MDELite.jar` – the **MDELite** jar.

To install **MDELite**, just place `MDELite.jar` on your `CLASSPATH`. In Windows, the incantation to do so is:

```
> set CLASSPATH=%CLASSPATH%;C:\xfer\dist\MDELite.jar
```

where `C:\xfer\dist` is the absolute path to the directory containing `MDELite.jar`. To check to see if you did the above tasks correctly, run the program `MDL.VerifyInstall`:

```
C>java MDL.VerifyInstall
Violet should be running now.
If not, something is wrong.
Otherwise, please close Violet,
and MDELite Ready to Use!
```

3 MDELite-Relational Schemas

In MDE-speak, a model conforms to a metamodel. In **MDELite**-speak, a metamodel is a relational schema; a relational database is a model that conforms to its schema. (There are constraints that are associated with a database schema, which we cover later in Section 5.)

MDELite allows you to outline a relational schema in a way inspired by Prolog facts. Here is a typical 'short' declaration in `school.ooschema.pl`:

```
dbase(school,[person,professor,department,student]).

table(person,[id,"name"]).
table(professor,[deptid]).
table(department,[id,"name","building"]).
table(student,[utid]).

subtable(person,[professor,student]).
```

The above means:

- The name of this schema is `school`. It contains four tables: `person`, `professor`, `department`, `student`;
- Every table has a name and a list of columns (attributes). The `person` table has two attributes: `id` and "`name`"; and
- Tables can be arranged in an inheritance hierarchy. The `subtable` declaration says classes `professor` and `student` are subtables of `person`.

There are three important conventions used in **MDELite** tables:

1) The first attribute of an **MDELite** table is an identifier field whose Tname need not be '`id`';
2) There are two kinds of fields in MDELite table schemas: those with unquoted attribute names and

those with quoted names. A quoted-name field means that all of its values must be quoted (single-or-double); an unquoted-name field means that all of its values must be unquoted;

3) An n-tuple of a table t is written as a prolog fact: $t(v_1 \ldots v_n)$. Some person tuples might be:

```
person(p1,'Don').
person(p2,'Barack Obama').
```

Values of a tuple are listed in the order that their column/attributes are listed in their table definition.

MDELite uses a more elaborate definition of a schema. You can produced this schema by running:

```
> java MDL.OO2schema school.ooschema.pl
       // school.schema.pl produced

> type school.schema.pl
dbase(school,[person,professor,department,student]).

table(person,[id,"name"]).
table(professor,[id,"name",deptid]).
table(department,[id,"name","building"]).
table(student,[id,"name",utid]).

subtable(person,[professor,student])
```

The only difference between a .ooschema file and its .schema counterpart is that attributes of super-tables are propagated to its sub-tables, recursively. Above, every professor tuple and every student tuple will have person attributes.

4 MDELite-Relational Databases

A **MDELite** database is an instance of a .schema.pl file. Recall the school.schema.pl of the previous section. An instance of this database is a separate file, named my.school.pl, where 'my' is the name of the instance, 'school' is the schema, and 'pl' denotes an **MDELite** file. Here is the my.school.pl database:

```
dbase(school,[person,professor,department,student]).

table(person,[id,"name"]).

table(professor,[id,"name",deptid]).
professor(p1,'don',d1).
professor(p2,'Robert',d1).
professor(p3,'Lorenzo',d2).
professor(p4,'kelly',d3).

table(department,[id,"name","building"]).
department(d1,'computer science','gates dell complex').
department(d2,'computer science','gates hall').
department(d3,'computer science','Bahen Centre').

table(student,[id,"name",utid])
student(s1,'zeke','zh333').
student(s2,'Brenda','UTgreat').
student(s3,'Thomas','astronaut201').
```

The above means:

- The student table has 3 tuples, department has 3 tuples, and professor has 4. Table person has 0 (no) tuples. This is like Java: objects/tuples are listed for the class/table in which they were created.
- The database schema definition is always included in a database file (that's the dbase() fact).
- The schema definition for each table is always included in a database file (that's the table() facts).
- The tuples of the table follow immediately after its table() fact. An absence of tuple declarations says the table is empty.

Note: **MDELite** does not automatically ensure that schema declarations of the database match that of the corresponding .schema file. So beware. **MDELite** has a tool that verifies (or reports differences) between a database schema and its database. To verify that the my.school.pl database conforms to school.schema.pl, run the MDL.InstanceOf tool below. In this case, conformance holds as there is silence for output.

```
>java MDL.InstanceOf my.school.pl school.schema.pl
```

5 MDELite Tools

MDELite offers the following tools:

- All
- InstanceOf
- Model Conformance
- Model-to-Model Transformation
- OOSchema to Schema Translation
- Reading Databases
- Reading Schemas
- Version
- Violet
- Violet Class Parser
- Violet Class UnParser
- Yuml Class Parser
- Yuml Class UnParser

All – List the **MDELite** tools, like the above.

```
C>java MDL.All
```

InstanceOf – This tool verifies that a database is an instance of its schema. We saw a use for this in an earlier section. To invoke this test, use the code below. Silence is returned if there are no errors.

```
C>java MDL.InstanceOf

Usage: MDL.InstanceOf <S>.schema.pl <Y>.<S>.pl
       confirms that database <Y> is an instance of <S>
```

Model Conformance – **MDELite** relies on you writing a Java program (typically using Java streams and **MDELite**-tool support) to write and evaluate constraints and to report errors. (The *Object Constraint Language* (OCL) is an awkward stream language; Java streams are more elegant).

Here are two constraints on the school database:

- **Person Name Constraint**: A Person's name must begin with a capital letter.
- **Name Uniqueness:** No two Persons have the same name.

A typical outline of schoolConform.java is sketched below.

```
import PrologDB.*;

public class schoolConform {

  public static void marquee() {
    System.err.format("Usage: %s <X>.school.pl\n",
                      schoolConform.class.getName());
    System.err.format("  <X> is name of database\n");
    System.exit(1);
  }

  static boolean checkCharacter(Tuple t) {
    String n = t.getName("name");
    if (n.length() == 0)
       return true;
    Character c = n.charAt(0);
    return Character.isLowerCase(c);
  }

  public static void main(String[] args) {
    if (args.length != 1 ||
       !args[0].endsWith(".school.pl")) {
      marquee();
    }

    DB db = DB.readDataBase(args[0]);
    Table person = db.findTableEH("person");
    ErrorReport er = new ErrorReport(System.out);

    // Person Name Constraint
    person.stream()
      .filter(t -> checkCharacter(t))
      .forEach(t->er.add("Person Name not " +
         "capitalized " + t.get("name")));

    // Name Uniqueness Contraint
    person.stream().filter(t->
      person.stream()
      .filter(g-> g.get("name").equals(t.get("name")))
      .count()>1)
    .forEach(t->er.add("Persons with duplicate" +
                    + "name : " + t.get("name")));

    try {
      er.printReport();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Perhaps the only thing strange is the use of class `ErrorReport`. An `ErrorReport` object maintains a list of errors that are posted to it by `Stream` expressions. When a report is printed and if at least one error was found, a `RuntimeException` is thrown. Incidentally, the output of this program is:

```
Person Name not capitalized don
Person Name not capitalized kelly
Person Name not capitalized zeke
Errors found
```

For further details see `MDELiteDemoPrograms.html`.

Model to Model (M2M) Transformation – A M2M transformation in **MDELite** is a Java program that implements a database-to-database transformation. It imports **MDELite** tools to read and write **MDELite** schemas and databases. Typically, although not required, it takes `2` arguments: the name of the input database file and the name of the output database file. Beyond that, how you write your database-to-database transformation is up to you. For further information see `MDELiteDemoPrograms.html`.

OOSchema Translation – `MDL.OO2schema` reads an input `x.ooschema.pl` file and converts it to a schema file `x.schema.pl`. Remember an `ooschema` file is a Java-like declaration of tables and their inheritance hierarchies. The attributes of a table are only those that are specific to that table. Flattening this schema propagates attributes of supertables to subtables. It is not difficut, but is error-prone. We saw an example use of `MDL.OO2schema` in the last section:

```
> java MDL.OO2schema

  Usage: MDL.OO2schema <X>.ooschema.pl
         outputs file <X>.schema.pl
```

Reading a Database – `MDL.ReadDB` reads a database and reports errors. If there are no errors, silence is returned:

```
> java MDL.ReadDB

  Usage: MDL.ReadDB <X>.<SCHEMA>.pl
         reads database <X> of type <SCHEMA> and
         reports errors
```

Reading a Schema – `MDL.ReadSC` reads a schema and reports errors. If there are no errors, silence is returned:

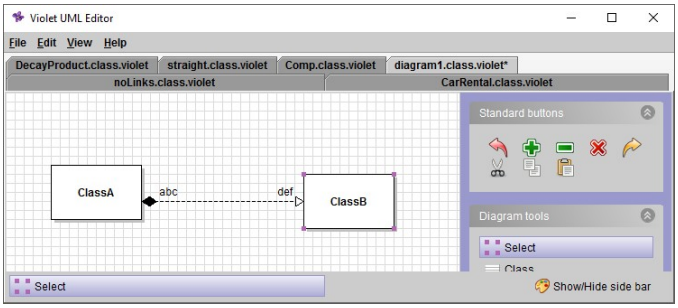
```
> java MDL.ReadSC

  Usage: MDL.ReadSC <X>.<SCH>.pl
         <SCH> is 'ooschema' or 'schema'
         reads schema x and reports errors
```

Version – returns the version number of **MDELite**:

```
> java MDL.Version
  MDELite version 6.0
```

Violet – You can invoke Violet directly through its jar file, but calling it from a command line is painful; `MDL.Violet` is easier:

```
> java MDL.Violet
// spawns Violet and waits for Violet to close
```



VioletClassParser – `MDL.ClassVioletParser` maps a Violet Class diagram file (`<X>.class.violet`) to a `vpl` database. The `vpl` schema is in `libpl/vpl.schema.pl` and is shown below:[1]

```
dbase(vpl,[violetMiddleLabels,violetAssociation,
           violetInterface,violetClass]).

table(violetClass,[id,"name","fields","methods",x,y]).
```

1. I have broken lines in code listings for presentation reasons. **MDELite** parsers expect one complete declaration per line.

```
table(violetInterface,[id,"name","methods",x,y]).
table(violetAssociation,[id,"role1","arrow1",type1,
         "role2","arrow2",type2,"bentStyle",
         "lineStyle",cid1,cid2]).

table(violetMiddleLabels,[id,cid1,cid2,"label"]).
```

To invoke the parser:

```
C>java MDL.ClassVioletParser

Usage: MDL.ClassVioletParser <in>.class.violet
       <out>.vpl.pl
```

VioletClassUnParser – `MDL.ClassVioletUnParser`maps a `vpl` database to a Violet Class diagram file (`<X>.class.violet`):

```
C>java MDL.ClassVioletUnParser

Usage: MDL.ClassVioletUnParser <X>.vpl.pl
        [<X>.class.violet]
       output file defaults to
       <X>.class.violet if unspecified
```

YumlClassParser – `MDL.ClassYumlParser`maps a Yuml specification file (`<X>.yuml.yuml`) to a `ypl` database. The `ypl` schema is in `libpl/ypl.schema.pl`:

```
dbase(vpl,[violetMiddleLabels,violetAssociation,
           violetInterface,violetClass]).

table(violetClass,[id,"name","fields","methods",x,y]).
table(violetInterface,[id,"name","methods",x,y]).
table(violetAssociation,[id,"role1","arrow1",type1,
         "role2","arrow2",type2,"bentStyle",
         "lineStyle",cid1,cid2]).

table(violetMiddleLabels,[id,cid1,cid2,"label"]).
```

To invoke the parser:

```
C>java MDL.ClassYumlParser

Usage: MDL.ClassYumlParser <IN>.yuml.yuml  <OUT>.vpl.pl
```

YumlClassUnParser – `MDL.ClassYumlUnParser`maps a `ypl` database to a Yuml specification file (`<X>.yuml.yuml`):

```
C>java MDL.ClassYumlUnParser

Usage: MDL.ClassYumlUnParser <X>.ypl.pl [<X>.yuml.yuml]
       output file defaults to <X>.yuml.yuml
       if unspecified
```

6 CLOSING

MDELite is a work in progress. It is possible that this documentation may get out-of-date with code releases. If so, please report them to me — `dsb`