# MDELite6 Manual

Don Batory
**batory@cs.utexas.edu**
November 2016

## 1 Introduction

MDELite6 is an alternative approach to Eclipse tools to teach and explore concepts in *Model Driven Engineering* (MDE). Rather than:

- Storing models and metamodels as obscure XML documents, MDELite6 encodes them as readable relational databases expressed as elementary facts.
- Using *Object Constraint Language* (OCL) to express constraints, MDELite6 uses Java Streams, a new extension to a basic language in Computer Science.
- Writing *model-to-model* (M2M) transformations in the *Atlas Transformation Language* (ATL), an outgrowth of OCL, MDELite6 again relies on Java.

For its *model-to-text* (M2T) tool, MDELite6 uses Apache Velocity, an off-the-shelf-tool used in industry. The benefits and overview of MDELite6 are explained in a 2013 MODELS paper.

Here is a table of contents for this manual:



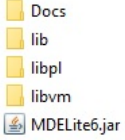
**Note:** As hard as I have tried, I know there are bugs in MDELite6. Please let me know when you find them. I will do my best to fix them – dsb

## 2 Installation

You can download MDELite6 from this link:

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

The MDELite6 directory and executable contains:




- Docs – documentation, including this manual,
- lib – a library of jar files needed by MDELite6,
- libpl – a library of predefined schemas, and
- libvm – a library of Velocity templates, and
- MDELite6.jar – the MDELite6 jar.

Installation of MDELite6 is simple: just place MDELite6.jar on your CLASSPATH. In Windows, the incantation to do so is:

```
> set CLASSPATH=%CLASSPATH%;C:\xfer\dist\MDELite6.jar
```

where C:\xfer\dist is the absolute path to the directory containing MDELite6.jar. To check to see if you did the above tasks correctly, run the program MDL.VerifyInstall:

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

## 3 MDELite6-Relational Schemas

In MDE-speak, a model conforms to a metamodel. In MDELite6-speak, a metamodel is a relational schema; a relational database is a model that conforms to its schema. (There are also constraints that are associated with a database schema, which we cover later in Section **??**.)

MDELite allows you to outline a relational schema in a fact-based way that is 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".

The following lines define attributes that are specific to the professor, department, and student tables. There are three important conventions used in MDELite6 tables:

1) The first attribute of a MDELite6 table that has no 'parent' or 'super' table is a manufactured id or identifier field. The name need not be 'id'.
2) There are two kinds of fields in MDELite table schemas: those with unquoted attribute names and those with single-quoted names.
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.

**Note.** In this example, id values are unquoted as the id attribute name is unquoted in the table declaration. name values are single-quoted as the name attribute name is double-quoted in the table declaration.

4) Tables can be arranged in an inheritance hierarchy, which are specified by subtable declarations like:

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

This declaration says that the subtables of person are professor and student. Equivalently, every professor and student is a person.

MDELite6 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 the .ooschema version and the .schema version 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 MDELite6-Relational Databases

A MDELite6 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 'y' is the name of the instance, 'school' is the schema, and 'pl' denotes an MDELite6 file. Here is the my.school.pl file:

```
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:** MDELite6 does not automatically ensure that all tables (even empty ones) are represented in a MDELite6 database file, or that the schema declarations of the database match that of the corresponding .schema file. So be careful. I don't every recall a problem, but it can happen. MDELite6 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 the school.schema.pl schema definition, 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 MDELite6 Tools

MDELite6 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
- Vm2T (Velocity)
- Yuml Class Parser
- Yuml Class UnParser

**All** – I need a reminder, occasionally, of the list of MDELite6 tools, like the above.

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

**InstanceOf** – A useful check is to verify that a database is an instance of a database schema. We saw a use for this in

an earlier section. Do 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** – In MDELite6, model conformance is checking whether a database conforms to a set of constraints. There is nothing in MDELite6 that evaluates constraints. Rather, using MDELite6 packages, you can write a Java program using Java Streams to stream tuples of one or more tables, apply filters and report errors as they are found, as discussed in class lectures. There is a MDELite6DemoPrograms.html in the MDELite6 Docs Directory that shows examples of such programs, how you should write them (e.g., conforming to MDELite6 tool standards), and how to invoke them. It is easy.

A conformance file for schema <S> is a set of constraints, c1...cn. These constraints are written as Java Stream expressions in a Java file, typically named <S>Conform.java (although this naming convention is not required as MDELite6 cannot enforce it). We will use the school database of a previous section as a running example. Here are two constraints on this database are:

- **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
```

Further information on MDELite6 programming is in the MDELite6DemoPrograms.html manual.

**Model to Model (M2M) Transformation** – A M2M transformation in MDELite6 is a Java program that implements a database-to-database transformation. It imports MDELite6 tools to read and write MDELite6 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. Further information on MDELite6 programming is in MDELite6DemoPrograms.html.

**OOSchema Translation** – MDL.OO2schema reads an input x.ooschema.pl file and converts it to a schema file x.schema.pl that is usable by MDELite6. 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 much, but this is a task that is error-prone. We saw an example use of MDL.OO2schema in the last section. Here is how it is invoked:

```
> java MDL.OO2schema

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

**Reading Database** – MDL.ReadDB reads a database and reports errors. If there are no errors, silence is returned. Here is how this tool is invoked:

```
> java MDL.ReadDB

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

**Reading Schema** – MDL.ReadSC reads a schema and reports errors. If there are no errors, silence is returned. Here is how this tool is invoked:

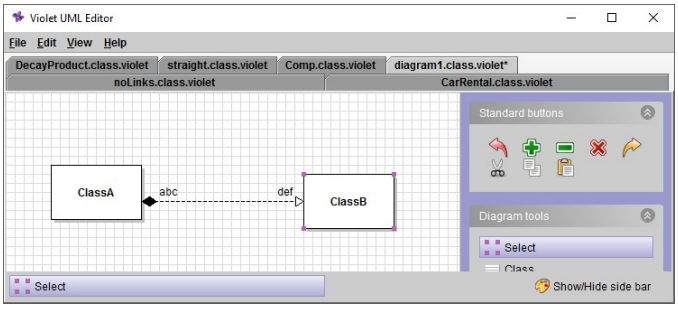
```
> java MDL.ReadSC

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

**Version** – returns the version number of `MDELite6`.

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

**Violet** – This program invokes the Violet tool. You can invoke Violet directly through its jar file, but calling it from a command line is painful; `MDL.Violet` makes it easy.

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




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

```
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.ClassVioletParser

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

**VioletClassUnParser** – `MDL.ClassVioletUnParser` is a tool that maps a `vpl` database to a Violet Class diagram file (`<X>.class.violet`). To invoke the parser:

```
C>java MDL.ClassVioletUnParser

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

**VM2T (Velocity)** – `MDL.Vm2t` is a model-to-text tool that understands `MDELite6` databases. It takes a database, such as file `db.S.pl`, and velocity template, such as file `template.vm`, as input. Where the output is sent is defined internally to `template.vm`, typically either to a special file `vm2toutput.txt`, or some other designated file that is based on name 'db'. Here is how `MDL.Vm2t` is invoked:

```
> java MDL.Vm2t

Usage: MDL.Vm2t database-file template-file
          [output-file]
       [-cg ContextGeneratorClass]
       if output-file is unspecified, output
       is directed to file vm2toutput.txt
```

See the VM2T manual, which is separate from this document for information on writing `MDELite6` Velocity templates.

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

```
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` is a tool that maps a `ypl` database to a Yuml specification file (`<X>.yuml.yuml`). To invoke the parser:

```
C>java MDL.ClassYumlUnParser

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

## 6 Closing

This tool is a work in progress. It is possible that this documentation may get out-of-date with code releases. If so, just report them to me and I will try to fix them a.s.a.p. — dsb

1. I have broken lines in code listings in this document for presentation reasons. Generally, the `MDELite6` parser expects one complete declaration per line.