MDELiteApp Manual

August 11, 2015

# Introduction

MDELiteApp is a classical example of an MDE application, as well as an MDELite application. The idea is to convert UML diagrams, in this case UML class diagrams, created in one tool to a corresponding (class) diagram in another tool. A category diagram of this MDELite application is shown below, and that is detailed in [**Teaching Model Driven Engineering from a Relational Database Perspective**](http://www.cs.utexas.edu/ftp/predator/15SoSym.pdf)**.** The contents of this document are:

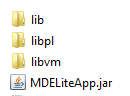
* [Installation Instructions](#_Installation_Instructions)
* [How to use MDELiteApp](#_How_to_Use)
  + [Converting Violet to Yuml](#_Converting_Violet_to)
  + [Converting Yuml to Violet](#_Converting_Yuml_to)
  + [Conformance Tests](#_Conformance_Tests)
* [Under the Covers](#_Under_the_Covers)
  + [Bootstrapping using Avalon](#_Bootstrapping_using_Avalon)
  + [Getting Started](#_Getting_Started)

# Installation Instructions

You can download MDELiteApp from this link:

<http://www.cs.utexas.edu/users/schwartz/MDELite/index.html>

The MDELiteApp directory contains:



* **lib** – is a library of jar files needed by CoreMDELite.
* **libpl** – is a library of Prolog database definitions, conformance files, and M2M transformations used by CoreMDELite,
* **libvm** – is a library of VM2T velocity templates used by CoreMDELite,
* **MDELiteApp.jar** – this is the MDELiteApp executable.

To install MDELiteApp, you must:

1. Place the MDELiteApp directory in a global directory in which you can place other MDELite applications, and
2. Add MDELiteApp.jar to your CLASSPATH.

To see if you accomplished the above steps correctly, run the Main file of the MDELiteApp application:

C>java mdeliteapp.Main

Usage: mdeliteapp.Main <option> <files>

Format: <file> = <filename>.<domain>.<domainType>

Option: conform <classname> <filename>

<classname> in { VClass Violet Sdb Nopos

Dot Kielerdot Pos Yuml YClass Start }

help <filename> // of type start.tmp

violet2yuml <filename> // of type class.violet

yuml2violet <filename> // of type yclass.yuml

coordinates

If you get the above response, congratulations! You installed MDELiteApp. If you didn’t get the above, it is a CLASSPATH problem. Check you CLASSPATH again.

# How to Use MDELiteApp

Here is its basic idea: you have 2 public UML drawing tools, [Yuml](http://yuml.me/) and [Violet](http://alexdp.free.fr/violetumleditor/page.php). You want to draw a UML diagram in Yuml and then convert it to a Violet diagram, and vice versa. Both tools use very different representations of their diagrams. MDELiteApp is a tool that performs this translation.

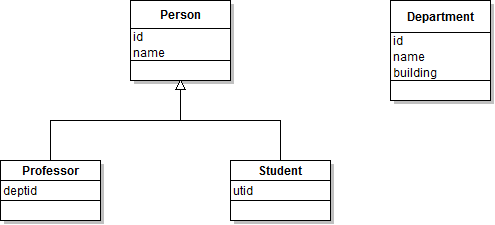
From the last part of the last section, the main application is mdeliteapp.Main. It basically offers three applications:

* [Converting Violet UML class diagrams into Yuml class diagram specifications](#_Converting_Violet_to),
* [Converting a Yuml class diagram spec into a Violet class diagram document](#_Converting_Yuml_to), and
* [To run conformance tests](#_Conformance_Tests) on a variety of documents, but most specifically Yuml UML class diagram documents (documents of type YClass) and Violet UML class documents (documents of type VClass).

The following subsections detail each application.

## Converting Violet to Yuml

Fire up Violet and draw the following class diagram, which you save as document **school.class.violet**.



**Note**: start violet by typing

> java Utils.Violet

You can convert this diagram/document into a yuml specification by running:

> java mdeliteapp.Main violet2yuml school

file school\_000.yclass.yuml was produced.

Look at the produced file, whose content is:

[person|id;name|]

[professor|deptid|]

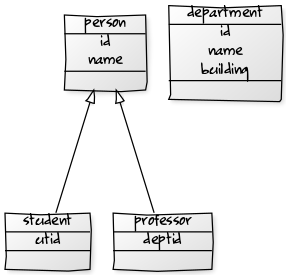
[student|utid|]

[department|id;name;building|]

[person]^-[professor]

[person]^-[student]

When this content is input input to [Yuml](http://yuml.me/diagram/scruffy/class/draw), you get this beautiful diagram;



## Converting Yuml to Violet

You can convert the above produced yuml file back into a Violet file:

> java mdeliteappl.Main yuml2violet school\_000

file school\_000\_000.class.violet was produced.

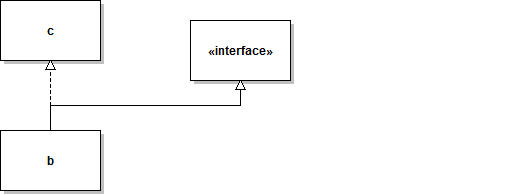
If you open this file in Violet, you get this beautiful diagram, which is equivalent to, but identical to, the original starting diagram:



In general, it is difficult, if not impossible, to have an identity map in such translations. Read: you can’t always make tools perfectly interoperable if they were never designed with this in mind. Too much information is lost (or was never specified) to create an exact match. You can come very close, and that’s good enough for practice.

## Conformance Tests

Here is the diagram of a Violet file **stupid.class.violet** that makes no sense:



Why? For two reasons: (1) the interface icon has no name and (2) both c and b are classes; the dashed arrow means b implements interface c – and c is not an interface. We want to know these errors. That’s the purpose of conformance tests. To find these errors, type:

c>java mdeliteapp.Main conform VClass stupid

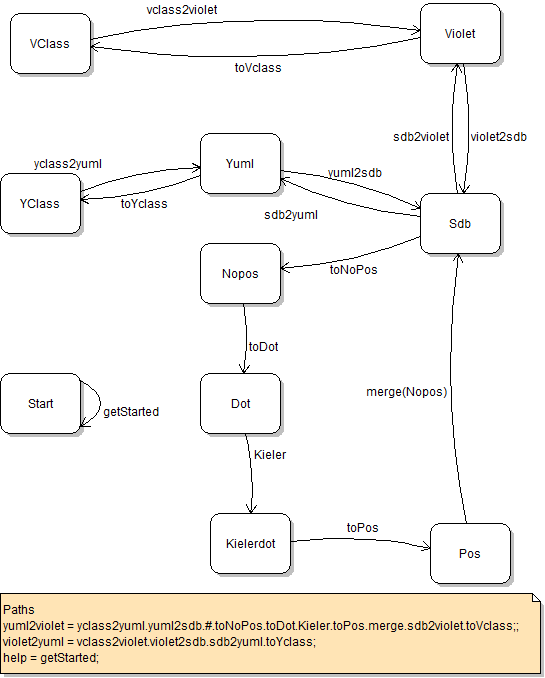
Conformance Error!

class or interface has null name interfacenode0

interface X cannot implement class where X=b

# Under the Covers

Here’s what is going on in MDELiteApp. You use Violet to create an ugly XML document that is an instance of the VClass domain. You want to convert this document into a Yuml specification, an instance of the YClass domain. In MDELite, computations such as these are traversals in a category diagram. The traversal that we are interested in is the path **violet2yuml**, defined in in the Paths note in the figure below. It defines the computation to convert a VClass document into a YClass document:

1. **vclass2violet** maps a Violet XML file into a Violet prolog database.
2. **violet2sdb** maps a Violet prolog database into a “standard” database that is an instance of the Sdb domain.
3. **sdb2yuml** maps a SDB prolog databast to a Yuml prolog database.
4. **toyclass** maps a Yuml Prolog database to a Yuml UML class document.

Now, converting a Yuml document into a Violet UML class diagram is the path **yuml2violet**. This path is considerably longer than **violet2yuml** for the following reason: yuml documents encode no positioning information. The Yuml tool, when it draws a UML diagram, supplies this information (but never needs it in a spec). The Violet tool, on the other hand, needs this (x,y) positioning information, otherwise all classes would be drawn on top of each other.

The extra steps in yuml2violet are to compute this positioning information. Here they are:

1. **toNoPos** transforms an SDB prolog database into a Nopos prolog database. “NoPos” literally means no positioning information.
2. **toDot** transforms a NoPos Prolog database into a DOT document (which is a graph specification language). This DOT document defines nodes and edges of a graph, but not their position.
3. **Kieler** transforms a DOT document into another DOT document, but this document has (x,y) positions for each class.
4. **toPos** transforms the (x,y)-enhanced DOT document into a Pos prolog database (which contains a single relation that defines x-y positioning information).
5. **merge**(NoPos) takes a Pos prolog databse and a NoPos Prolog database and merges them into an **SDB** database.

So look at the big picture of what the above steps accomplish: a Yuml class diagram specification was transformed into an SDB prolog database. This database has a position relation, but it is “empty” in that its data is meaningless. The above 5 steps transform this SDB database without positioning information to one that has positioning information, and this can be mapped to a Violet class diagram document.

Now, there is one tricky step that we haven’t covered, and it has to do with MDELite file naming conventions. The path yclass2yuml.yuml2sdb maps a YClass document X.yclass.violet to X.sdb.pl. When the above path **toNoPos.toDot.Kieler.toPos.merge** is executed, it maps an X.spb.pl document to X.spb.pl – that is it overrides X.sdb.pl. You don’t want to do this. The “#” arrow in the yuml2violet path says rename (or version) X.sdb.pl to X\_000.sdb.pl. So when merge is executed, it takes the original X.sdb.pl file and the newly created X\_000.pos.db to produce a new X\_000.sdb.pl document, which can then be mapped to an **X\_000.yclass.yuml** file/document[[1]](#footnote-1).

Why is this “versioning” operation needed? One of the goals of MDELite is to show that MDE is really OO metaprogramming: MDE objects are documents and MDE transformations/arrows are programs that convert documents of one type into documents of other types. In OO programming, you create a LOT of temporary objects, and when your program is completed they are thrown away. Well, MDELite creates a lot of temporary objects (a.k.a. files) and throws them away, but it still has to deal with them and their naming conventions. Here is literally the Java code that executes the yuml2violet path:

String filename = args[1];

YClass v0 = new YClass(filename);

Yuml v1 = v0.yclass2yuml();

Sdb v2 = v1.yuml2sdb();

v2.newVersionNumber();

Nopos v3 = v2.toNoPos();

Dot v4 = v3.toDot();

Kielerdot v5 = v4.Kieler();

Pos v6 = v5.toPos();

Sdb v7 = v6.merge(v3);

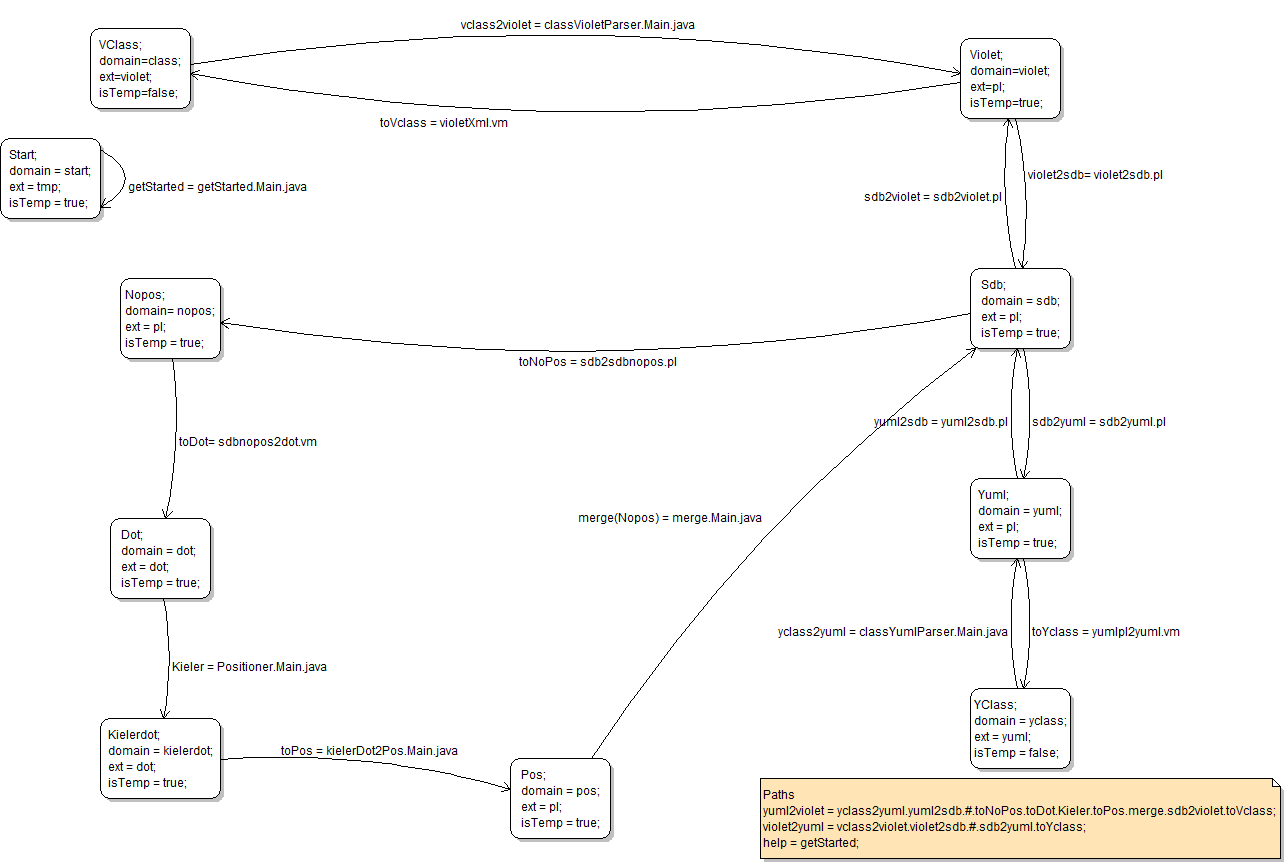
Violet v8 = v7.sdb2violet();

VClass v9 = v8.toVclass();

The file that is produced, v9, is a Violet UML Class XML document. You have to write this code, manually. Or better yet, you can use the Avalon tool to generate this code automatically for you. The ONLY code that you have to write – which is not necessarily simple – are the programs that implement each arrow of your category diagram.

## Bootstrapping using Avalon

You can build the shell of a NetBeans (or Eclipse) projet for MDELiteApp by providing a specification of the category diagram (or equivalently a special class diagram). Here is the category diagram (really a specialized state diagram in Violet) for MDELiteApp. The figure is hyperlinked to a file which you can open with Violet.

[](mdeliteappFull.state.violet)

Every domain (bubble) has the following structure:

* the name of the domain – a Java class will be created in its name
* an alternate domain name – one that is used in actual file names
* the extension of such a file, ex: “.pl” for prolog, “.java” for Java files
* an indicator if this file is created as an intermediate result – if so, it is flagged as temporary.[[2]](#footnote-2)

Every arrow (transformation) has the following structure:

* the name of the arrow
* followed by “=” the program that implements it. Arrow vclass2violet is implemented by the Java program classVioletParser.Main. Arrow violet2sdb is implemented by the M2M prolog file violet2sdb.pl. And arrow toYclass is implemented by the VM2T/Velocity template yumlplyuml.vm.

The MDELiteApp offers 3 computations, which are defined as “paths” in the Paths note:

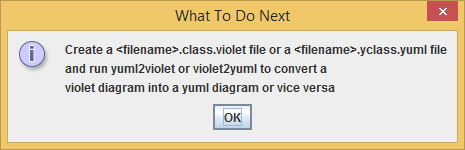
* yuml2violet – convert a Yuml specification into a Violet document
* violet2yuml – convert a violet document into a Yuml specificatoin
* help – getting started. See next section.

## Getting Started

I would hope that all MDELite applications have a “help” option. Choosing the help option (where you can supply any additional argument that you want) as in:

> java mdeliteapp.Main help x

Produces the following prompt:



When OK is clicked, Violet is started so that you can continue on you way. This “advice” or help program isn’t too much past typing:

> java mdeliteapp.Main

But sometimes, especially in the case of Avalon, it becomes really useful. Good luck!

1. Or A\_.D.E where n is a number to A\_ .D.E where [↑](#footnote-ref-1)
2. The reason is that MDELite is really dirty when it runs. As any Java program, lots of temporary variables are used. But in MDELite, these temporary variables have files as values. Flagging a domain as temporary means that its files will be deleted at the end of an MDELiteApp computation. [↑](#footnote-ref-2)