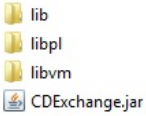
CDExchange Manual

August 2015

1. Introduction

CDExchange is a classical MDE application. It converts UML diagrams, in this case UML class diagrams, created by one tool to a corresponding diagram in another tool. A category diagram of MDE application is shown below and is detailed in a 2013 MODELS paper. In this diagram, a bubble denotes a domain of documents and an arrow $A : B \rightarrow C$ is a function that transforms a document from domain B to a document in domain C:



The table of contents of this document are:

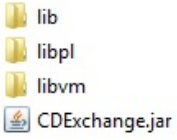
Warning: As hard as I have tried, I know there are bugs in CDExchange. Please let me know when you find them. I will do my best to fix them.

2. Installation

You can download CDExchange from:

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

The CDExchange directory contains:



- **lib** – a library of jars needed by CDExchange,
- **libpl** – a library of prolog database schema definitions, conformance files, and M2M transformations used by CDExchange,
- **libvm** – a library of Velocity templates used by CDExchange,
- CDExchange.jar – this is the CDExchange executable.

To install CDExchange:

- place the CDExchange directory within a global directory in which other MDELite applications reside, and
- add CDExchange.jar to your CLASSPATH.

To see if you have accomplished the above steps correctly, run:

```
> java cdexchange.Main

Usage: cdexchange.Main <option> <files>
Format: <file> = <filename>.<domain>.<domainType>

Option: conform <classname> <filename>
      <classname> in ( Dot Kielerdot Nopos Pos Sdb Start
          Violet VioletDB Yuml YumlDB )
      help            <filename>    // of type start.tmp
      violet2yuml     <filename>    // of type class.violet
      yuml2violet     <filename>    // of type yuml.yuml
      coordinates
```

If you get the above response, congratulations! You installed CDExchange. If you didn't, check your CLASSPATH.

3. How To Use CDExchange

Here is the basic idea of CDExchange: you have two public UML drawing tools, Yuml and Violet. You want to draw a UML diagram in Yuml and convert it to a Violet diagram and vice versa. Both tools use very different diagram representations. CDExchange is a tool that performs this translation.

From the end of the last section, the main application is CDExchange.Main. It bundles three smaller applications:

- Convert Violet class diagrams into Yuml class diagrams,
- Convert a Yuml class diagram into a Violet class diagram, and
- Run conformance tests on a variety of documents, but most specifically Yuml class diagrams and Violet class diagrams.

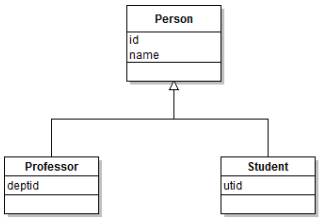
The following subsections detail each application.

3.1 Converting Violet to Yuml

Fire up Violet by typing:

```
> java Utils.Violet
```

and draw the following class diagram, which you save as document school.class.violet.



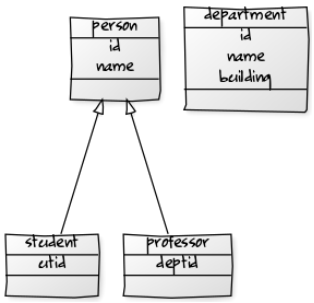
You can convert this diagram/document into a Yuml document by running:

```
> java cdexchange.Main violet2yuml school
```

File `school_000.yuml.yuml` is produced as output. Look at its content:

```
[person|id;name|]
[professor|deptid|]
[student|utid|]
[department|id;name;building|]
[person]^-[professor]
[person]^-[student]
```

When this content is input to Yuml, you get this beautiful diagram:



3.2 Converting Yuml to Violet

You can convert the above Yuml document back into a Violet document:

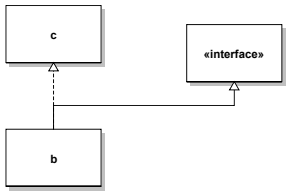
```
> java cdexchange.Main yuml2violet school_000
```

File `school_001.class.violet` is produced as output. By opening this file in Violet, you'll see this beautiful diagram, which is equivalent, but identical, to the original starting diagram.

In general, it is difficult, if not impossible, to have an identity map in such translations. *Lesson: you can't always make tools perfectly interoperable if they were never designed with interoperability in mind.* Too much information is lost (or was never specified) to create an exact match. You can come close, and that is as much as you can expect.

3.3 Conformance Tests

`CDExchange` performs conformance tests. The diagram below is of a Violet file `stupid.class.violet` that makes no sense because: (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.
Finding these errors is the purpose of conformance tests. To find them, type:

```
>java cdexchange.Main conform Violet stupid
Conformance Error!

class or interface has null name interfacenode0
interface X cannot implement class where X=b
```

4. Under the Covers

Violet to creates an ugly XML document that is an instance of the `Violet` domain. You want to convert this document into a Yuml specification, an instance of the `Yuml` 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.

`violet2yuml` is the path that converts a `Violet` document into a `Yuml` document. The individual steps of this path are:

1. `VClass2violet` maps a Violet XML file into a Violet prolog database;
2. `violet2sdb` maps a Violet prolog database into a prolog database that is an instance of the `Sdb` domain;
3. `sdb2yuml` maps a `Sdb` prolog database to a Yuml prolog database; and
4. `toYuml` maps a Yuml prolog database to a Yuml class document.

Converting a Yuml document into a Violet document 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 $(\texttt{x}, \texttt{y})$ positioning information, otherwise all classes are drawn on top of each other, yielding an unreadable mess.

The extra steps in `yuml2violet` are to compute positioning information for class icons. Here are they are:

1. `toNoPos` transforms an `Sdb` prolog database into a `Nopos` prolog database. "`Nopos`" means no positioning information;
2. `toDot` transforms a `Nopos` database into a `Dot` document, which is a graph specification. A `Dot` document defines nodes and edges of a graph, but not their position;
3. `Kieler` transforms a `Dot` document into another `Dot` document that has $(\texttt{x}, \texttt{y})$ positions for each class;
4. `toPos` transforms the $(\texttt{x}, \texttt{y})$-enhanced `Dot` document into a `Pos` prolog database which contains a single relation that defines $(\texttt{x}, \texttt{y})$ positioning information; and

5. `merge(Nopos)` takes a `Pos` database and a `Nopos` database and merges them into an `Sdb` database.

Look at what the above steps accomplish: a Yuml class diagram is transformed into an SDB database. This database has a position relation, but it is empty/useless. The above five steps transform this `Sdb` database without positioning information to one that has positioning information. This composed database can be mapped to a readable Violet class diagram.

There is one tricky step that we haven't covered, and it has to do with MDELite file naming conventions. These naming conventions are documented in the `MDELite` manual.

Briefly, every MDELite file has a triple `A.D.E` for a name. `A` is the application name, `D` is the name of the domain, and `E` is the file extension. An MDELite transformation always retains the application name `A`, but may change the `D.E` suffix. So the `toDot` arrow transforms document `A.nopos.pl` to `A.dot.dot`.

Now, here's the problem. The subpath `toNoPos.toDot.Kieler.toPos.merge(Nopos)` into a new definition of `A.sdb.pl` that overrides the original `A.sdb.pl`. Overriding existing files is something that I've learned to avoid in MDELite. The fix is this: The `#` step in the `yuml2violet` path says copy (or version) the file that was just produced – in this case `A.sdb.pl` – and name it to `A_000.sdb.pl`. When `merge` is executed, it takes the original `A.sdb.pl` file and the newly created `A_000.sdb.pl` to produce a new `A_000.sdb.pl` document, which is then translated to an `A_000.Yuml.yuml` file/document.[1]

Here is the Java code that executes the `yuml2violet` path:

```
String filename = args[1];
Yuml v0 = new Yuml(filename);
Yuml v1 = v0.YClass2yuml();
Sdb v2 = v1.yuml2sdb();
v2.newVersionNumber();
Nopos v3 = v2.toNoPos();
Dot v4 = v3.toDot();
System.out.format("file %s is produced\n",v4.fullName);
```

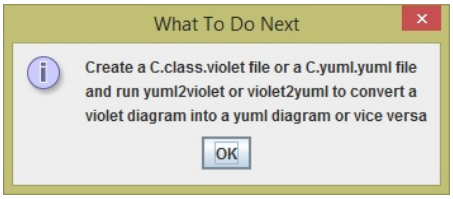
This code is sufficiently structured to be automatable. That is what the Avalon MDELite application does.

4.1 Getting Started

If you're new to `CDExchange`, type:

```
> java cdexchange.Main help C
```

where `C` is some name. `CDExchange` responds by posting: and



basically tells you to create a Violet class diagram or Yuml class diagram, and convert it. The conversion calls are:

```
> rem if C is a Violet diagram
> java cdexchange.Main violet2yuml C
```

or

```
> rem if C is a yuml diagram
> java cdexchange.Main yuml2violet C
```

You can also use `CDExchange` to validate the sanity of your class diagram. The conformation calls are:

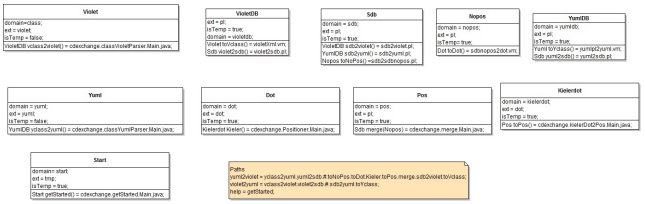
```
> rem if C is a yuml diagram
> java cdexchange.Main conform  Violet C
```

or

```
> rem if C is a yuml diagram
> java cdexchange.Main conform Yuml C
```

5. How `CDExchange` was Bootstrapped

`CDExchange` is itself an `CDExchange` application. It has a class diagram, which is shown below. I know this is really small, but it is a digitally enlargeable figure.



[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 finishes executing they are thrown away. Well, MDELite creates a lot of temporary objects (*aka,* files) and throws them away, but it still has to deal with them and their naming conventions.