Avalon Manual

August 2015

1. Introduction

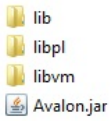
Avalon is a 3rd generation of MDELite and a 2nd generation bootstrap. It relies on a generative approach to produce and execute MDELite applications, rather than writing all MDELite application code by hand or by interpreting MDELite specifications.

Use Avalon to build an MDELite application. Avalon is a meta metamodel of MDELite: all Avalon applications are MDELite applications. Avalon generates tedious code and files that would otherwise have to be implemented manually. It doesn't do everything for you, but gets you started and tells you what you need to do. And it is a good way to appreciate the details of what goes on in an MDE application. Here is a table of contents of this manual:

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

2. Installation

Avalon requires the MDELite framework. The Avalon directory and executable contains:



- **lib** – a library of jar files needed by Avalon,
- **libpl** – a library of Prolog database definitions, conformance files, and M2M transformations, and
- **libvm** – a library of VM2T velocity templates, and
- **Avalon.jar** – the Avalon executable.

To install Avalon, you must:

1. Install the MDELite framework,
2. Place the Avalon directory in a global directory where you have other MDELite tools, such as MDELite,
3. Add Avalon.jar to your CLASSPATH.

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

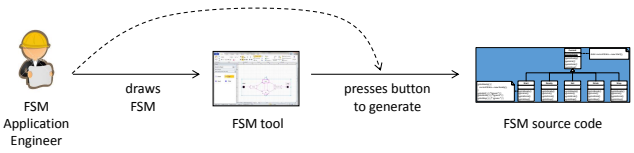
```
C>java avalon.Main

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

Option: conform <classname> <filename>
        <classname> in ( AvalonSpec AvalonSpecCD ClassViolet Meta
           Metamodel Schema Start VioletPl )
        buildClass      <filename>    // of type class.violet
        buildState      <filename>    // of type state.violet
        help            <filename>    // of type start.tmp
        coordinates
```

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

3. An Overview of Avalon

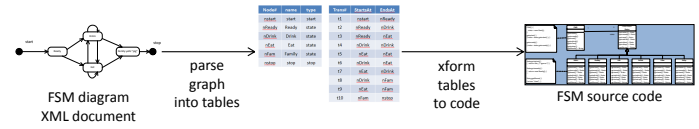
Here's a simple MDE application: you want engineers to draw a *finite state machine* (FSM) using Violet, a free Java UML diagram drawing tool. You want it to perform the following tasks:

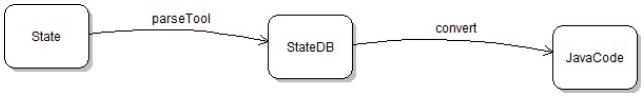
- Apply constraints to ensure that every drawn FSM is reasonable, *eg,* no unreachable states, no start state, etc.
- Transform the FSM into Java code by a push of a button. See figure below:



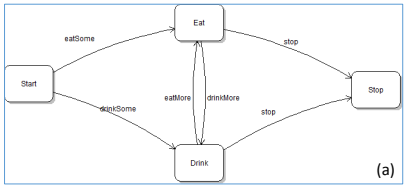
Internally, here's what is going on: Violet stores an FSM diagram as an ugly XML document. This document must be parsed to harvest the essential data that describes the FSM. MDELite stores this data as a relational database (a set of relational tables) in Prolog. Constraints that determine the sanity of a diagram (or rather, its relational database representation) must be written and then applied to this database. If no constraint is violated, the FSM is declared "sane". This triggers the next task: translate the database into FSM source code. MDELite uses an adapted version of Apache Velocity to do this. Velocity is a free text-generation tool that we have adapted to read Prolog databases. See figure below:

The simplest way to describe any MDE application is to draw a *category diagram*: a bubble D denotes a domain D of objects called documents or models, and an arrow $A : D \rightarrow R$ is a transformation that maps an object in domain D to an object in co-domain R. Here's a category diagram for this application:





The Violet tool is constructor: it produces a XML document (an instance of domain `State`). You have to write an XML parser to translate this XML document into a Prolog file that encodes a beautiful database. Figure 1a shows my eating habits (I stop eating when my family yells at me) and Figure 1b is a prolog database that encodes this FSM.




```
:-style_check(-discontiguous).

dbase(fsmdb,[state,transition]).

table(state,[id,"name"]).
state(s0,'Start').
state(s1,'Eat').
state(s2,'Drink').
state(s3,'Stop').

table(transition,[id,"name",startid,endid]).
transition(t1,'eatSome',s0,s1).
transition(t2,'drinkSome',s0,s2).
transition(t3,'eatMore',s1,s2).
transition(t4,'drinkMore',s2,s1).
transition(t5,'stop',s1,s3).
transition(t6,'stop,s2,s3).
```
(b)

Figure 1: FSM Diagram and its MDELite Prolog Database.

The Prolog facts in Figure 1b define a database schema and more:

- Prolog likes all facts with the same name to be on consecutive lines. `table` facts fail this requirement. The first line, `style_check`, tells Prolog not to expect facts to be consecutive.
- `dbase(dbname,list-of-table-schema-names)` is how MDELite declares a database schema. A database schema has a name `dbname` and a list of names of table schemas.
- `table(tname,list-of-column-names)` is how MDELite declares a relation schema. A table schema has a name `tname` and a list of column names. A column name can be unquoted (`id`) or double-quoted (`"name"`). A quoted column name means that values in that column must be single-quoted. Column values for tables are never lists and always singleton values.
- `tname(comma-separated-column-values)` is how MDELite declares a tuple in table `tname`. A tuple is a comma-separated-list of column values, the order in which values are listed is the order in which their columns are defined in the table schema.

Another task of our MDE application is to translate the database in Figure 1b into source code. Figure 2 shows a velocity script that performs this translation. Please review the VM2T (Velocity Model 2 Text) manual in the Documents directory of the `MDELite` distribution. A velocity template is fairly easy to read – the only obscure commands are the first two lines in Figure 2: line 1 defines text that is a output file `MARKER`; line 2 says direct file output to the file listed on the `VM2T` command line; default is `vm2toutput.txt`. The text of the Java class that is produced by the above database and template is shown below:

```
   private class Start extends FSM {
      Start() {}

      FSM transition(String transitionName ) {
         switch(transitionName) {
         case "eatSome":
            return new Eat();
         case "drinkSome":
            return new Drink();
         default:
            return this;  // nothing happens
         }
      }

      String currentStateName() {
         return "Start";
      }
   }
```

```
-----------------------------------------------------------------
#set($MARKER="//----")
${MARKER}${OutputFileName}
class FSM {
   FSM state = new Start();

   void nextState( String transitionName ) {
      state = state.transition(transitionName);
   }

   String currentStateName() {
     return state.currentStateName();
   }

   FSM transition( String x ) {
      throw new RuntimeException("should not be called");
   }

   #foreach( $s in $stateS)
   private class ${s.name} extends FSM {
     ${s.name}() {}

     FSM transition(String transitionName ) {
        switch(transitionName) {
     #foreach( $t in $transitionS )
        #if ($t.startid == $s.id)
           #foreach( $e in $stateS )
             #if($e.id == $t.endid)
                #set( $newState = "$e.name")
                #break
             #end
           #end
        case "${t.name}":
          return new ${newState}();
        #end
     #end
        default:
          return this; // nothing happens
        }
     }

     String currentStateName() {
        return "${s.name}";
     }

   }
   #end
}
```

Figure 2: A Velocity Template

4. How to Write Arrows or Document Xforms

In MDELite, you (the programmer) have to write a program for each arrow in a category diagram. This could be a Java program, Prolog program, or Velocity script. MDELite insists on a few conventions that you must follow; they are described below.

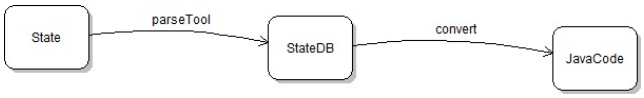
- MDELite File Naming Conventions
- Java Programs
- Prolog Programs
- Model Constraint Files
- Model-to-Model Transformation Files
- VM2T (Velocity) Programs

4.1 MDELite File Naming Conventions

All files in MDELite have triplet names "N.D.E":

- `N` is the name of your application,
- `D` is the name of an MDELite domain. This is the name of the bubble that we have used earlier in category diagrams, and
- `E` is an extension – like `.java`, `.txt`, `.class`, or `.pl`. These extensions used by operating systems to denote a file's type.

In the category diagram of our running example:



Violet produces an XML file whose name is `n.state.violet` where:

- `n` is the name of the FSM application that you have given,
- `state` is the name of the MDELite domain, and
- `violet` is the extension that tells you and the *operating system* (OS) that this is a Violet XML file.

By convention in MDELite, when a `state` document is translated into a `statedb` document, the application name is preserved. So arrow `parseTool` maps document `n.state.violet` to `n.statedb.pl` where:

- `n` is the name of the FSM application
- `statedb` is the name of the MDELite domain (in lowercase)
- `pl` tells you and the OS that this is a Prolog file.

Similarly the `convert` arrow translates `n.statedb.pl` to `n.javacode.java`, where:

- `n` is the name of the FSM application
- `javacode` is the name of the MDELite domain (in lowercase)
- `java` tells you and the OS that this is a Java file.

This file naming convention was put into place so that you can immediately see (upon looking at the classes in a directory) what files belong to the same application. The domain name of these files tells you the representation that is encoded in that file.

4.2 Java Programs

Most arrows in MDELite category diagrams are functions of the form $\texttt{A} : \texttt{D} \rightarrow \texttt{R}$; that is, a document from domain `D` is taken as input and a document from co-domain `R` is produced as output. The name of the parsing tool discussed in the last section is `parseTool`.

Create an Eclipse/Netbeans project called `parseTool`. It has a `Main` class, the façade to the tool. The standard structure for all MDELite Java tools is shown below. The `main()` method takes an open-ended array of arguments. In a specific tool, such as `parseTool`, the number of inputs and output is precisely known. In the example below, the parse tool takes a file `x.y.violet` (`x.y` is user supplied) as input and a file `z.w.pl` (`z.w` is user supplied) is produced as output.

```
package parseTool;

import CoreMDELite.Error;

public class Main {

    public static void main(String... args) {
        if (args.length != 2) {
            System.out.println("Usage: ParseTool.Main <in>.violet <out>.pl");
            throw new Error("incorrect invocation of program");
        }
```

This organization provides the following benefits:

- You can develop and debug arrows (Java programs) in isolation. An MDE application is essentially a collection of programs that read and write documents of fixed types.
- If you discover a problem with your MDELite application, you can track the bug in some arrow, and then fire up your favorite IDE to hunt for it and squash it.

Some further hints: There are MDE transformations that take $\texttt{n} > 1$ input documents and produce $\texttt{m} > 1$ output documents. You encode this transformation in MDELite by a pair of arrows. The first arrow $\texttt{F} : \texttt{D}_1 \ldots \texttt{D}_\texttt{n} \rightarrow \texttt{R}$ that takes the `n` input documents and produces a single document of type `R` as output. Then you have a set $\pi_\texttt{i} : \texttt{R} \rightarrow \texttt{O}_\texttt{i}$ projection arrows, $1 \leq \texttt{i} \leq \texttt{m}$, that allows you to produce one output at a time. There are two simple ways to implement this:

1) Arrow `F` is a Java program that has $\texttt{n}+1$ arguments: the first `n` arguments are documents from domains $\texttt{D}_1 \ldots \texttt{D}_\texttt{n}$ and the last argument is an `R` document. Program `F` really produces $\texttt{m}+1$ outputs (the first `m` are documents from the output domains $\texttt{O}_1 \ldots \texttt{O}_\texttt{m}$) and the last is an `R` document; but the command-line interface to `F` says it only produces an `R` document.[1] An `R` document could simply be a text file that lists the path names of the produced $\texttt{O}_1 \ldots \texttt{O}_\texttt{m}$ documents. A $\pi_\texttt{i}$ program is then trivial: it takes an `R` document as input returns document $\texttt{o}_\texttt{i}$ as output. In this way, MDELite allows you to encode any MDE transformation that you want.
2) Arrow `F` is a Java program that follows the structure in 1) above. But this time, `R` is not a document but instead is the name of a directory. For example, you can generate any number of Java files, but they all belong to a directory `R`. In our running example, the `JavaCode` domain could be a directory of Java files.

Later, I'll show how to define such arrows in a category diagram.

Here's another topic: error reporting. MDELite understands two kinds of exceptions:

- **Conformance** – errors that are classified as model constraint failures, and
- **Error** – all other errors.

Conformance errors are written to the MDELite file whose name is `conform.txt`; all other errors are posted in file whose name is `error.txt`. Usually I try to post as many errors as possible before

[1] So the technique is to lie `:-(`.

throwing a conformance exceptions. The convention is: when a conformance error is thrown:

```
throw new ConformanceError("some reason");
```

then programmers should look at the `conform.txt`[2] for details. When an execution error is thrown:

```
throw new Error("some reason");
```

then programmers should look at the `error.txt`[3] for details. In general, MDELite tools will announce, as their output, which of these two files should be examined, if they don't print these files for you.

4.3 Prolog Files

There are two different Prolog executables that you can create:

- One that defines metamodel constraints to validate a model. In MDELite-speak, this is a file that defines Prolog constraints to check the sanity of a Prolog database.
- Another is to define model-to-model transformations. In MDELite-speak, this is a set of Prolog rules that translates one Prolog database into another.

I consider each in turn.

4.3.1 Constraint Files

Standard fare in MDE tooling is a language to express metamodel constraints. The standard language today is OCL. For many reasons, I'm unhappy with OCL and feel that Prolog is a better and more fundamental language for this task. I presume that you're familiar at least Prolog basics.

Here are two constraints that I want to impose on any FSM database. First, all states have unique names. Second, all transitions reference an existing starting state and ending state. There are indeed other constraints I could write, but these are sufficient to illustrate the general process.

Here is a Prolog file, named `cons.pl`, that defines these constraints:

```
/* duplicate names constraint */
duplicateNames(X):-state(I1,X),state(I2,X),@I1 @< @I2.

uniqueNames:-forall(duplicateNames(X),isError('multiple states have duplicate names ',X)).

/* bad reference constraint */
aBadref(X):-not(state(X,_)).

badRef(X):-transition(_,_,X,_),aBadref(X).
badRef(X):-transition(_,_,_,X),aBadref(X).

badRefs:-forall(badRef(X),isError('reference to a non-existent state ', X)).

/* run executes all constraints */
run:-uniqueNames,badRefs.

/* for interactive development */
reload:-[cons].
```

The first rule finds two states that have the same name but different `id`s. The `uniqueNames` constraint finds all duplicate names and reports them via the `isError(N,V)` predicate. `N` is a string that is printed with the value `V`. The `forall` predicate says: for every binding that `duplicateNames(X)` is true, execute `isError('...',X)`. The bad reference constraint is a bit more complicated. `aBadRef(X)` is true if there exists no state with `X` as a state `id`. A `badRef(X)` is true if either the `startid` or `endid` referenced in a transition tuple is bad. The `badRefs` constraint reports all bad references.

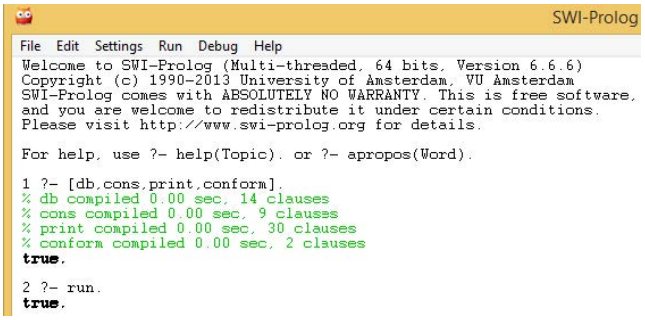
MDELite uses predicate `run` as the Prolog rule that evaluates all constraints. Note these constraints NEVER fail. MDELite sees if evaluating a constraint file produces any error output. MDELite directs all constraint violations to a text file named `conform.txt`. After constraint evaluation, if `conform.txt` is empty, the database satisfies the constraints, and MDELite is happy. If non-empty, then MDELite reports an error.

Here's how you can interactively develop and debug a constraint file. Look at the last statement in the Prolog constraint file above:

```
/* for interactive development */
reload:-[cons].
```

The name of this file is `cons.pl`. The `reload` rule executes the instruction `[cons]`, which means throw away all prior definitions of file `cons.pl` and reload/recompile `cons.pl`.

In a separate directory, I place a file `db.pl` (an example database), `cons.pl` (the above constraint file), `conform.pl` (which contains the MDELite definition of the `isError` predicate) and `print.pl` (an MDELite file that prints a database). The `conform.pl` and `print.pl` files are copied from the `MDELite/libpl` directory, which you installed prior to `Avalon`. I open 3 windows – one for `db.pl`, another for `cons.pl`, and a third running SWI-Prolog. In SWI-prolog window, I compile all four files:



I never change `print.pl` or `conform.pl`. But when I'm interactively developing model constraints, I'm always modifying `cons.pl` (and then I `reload` and `run` the constraints), and occasionally I modify `db.pl` to introduce erroneous facts that I want to catch by my error rules.

> **Note:** This triad of windows is what would be really useful in an IDE plugin for MDELite, which I leave to more industrious folk to create.

4.3.2 Model-to-Model Transformation Files

The other kind of Prolog file performs a M2M (model-to-model) transformation, or in MDELite-speak, a database-to-database translation.

If you studied the Velocity template presented earlier, you would have noticed that there is an extra loop to translate a `startid` attribute of a transition tuple into a `state` name. The template would have been simplified if transition tuples used `state` names instead of state `id`s. We can use this as an example of a M2M (database-to-database) transformation. Here is a partial schema definition of our target Prolog database, in file `fsmdb1.pl`:

```
dbase(fsmdb1,[stat,trans]).
table(stat,[id,"name"]).
table(trans,[id,"name",start,end]).
```

A complete schema that MDELite can use is produced by invoking the `Util.ElaborateSchema` utility:

```
> java Util.ElaborateSchema fsmdb1.pl
```

Which produces the `fsmd1.schema.pl` file below:

[2] This name may change. Look at `CoreMDELite.Globals.conformFile`.

[3] This name may change. Look at `CoreMDELite.Globals.errorFile`

```
dbase(fsmdb1,[stat,trans]).

table(stat,[id,"name"]).
table(trans,[id,"name",start,end]).

tuple(stat,L):-stat(A1,A2),L=[A1,A2].
tuple(trans,L):-trans(A1,A2,A3,A4),L=[A1,A2,A3,A4].

statALL(A1,A2):-stat(A1,A2).
transALL(A1,A2,A3,A4):-trans(A1,A2,A3,A4).
```

Note: The `tuple` facts are used to print prolog databases. The `ALL` facts are present if you want to retrieve tuples from tables that are related by an inheritance hierarchy. There are no such inheritance relationships in this schema.

Here is a Prolog file, `m2m.pl`, that translates an `fsmdb` database into an `fsmdb1` database:

```
% m2m.pl

stat(I,N):-state(I,N).
trans(I,N,S,E):-transition(I,N,Sid,Eid),state(Sid,S),state(Eid,E).

reload:-[m2m].
```

You can run this by (a) loading the `print.pl` file (first), then in any order, the database, M2M file, and the schema file of the database to produce, (b) execute `printDbase(fsmdb1)` – print the database whose schema is `fsmdb1`, as shown below. Type '`halt.`' to exit SWI-Prolog.

```
For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- [print,db,m2m,'fsmdb1.schema'].
% print compiled 0.00 sec, 30 clauses
% db compiled 0.00 sec, 14 clauses
% m2m compiled 0.00 sec, 3 clauses
% fsmdb1.schema compiled 0.00 sec, 8 clauses
true.

2 ?- printDbase(fsmdb1).
:-style_check(-discontiguous).

dbase(fsmdb1,[stat,trans]).

table(stat,[id,"name"]).
stat(s0,'Start').
stat(s1,'Eat').
stat(s2,'Drink').
stat(s3,'Stop').

table(trans,[id,"name",start,end]).
trans(t1,'eatSome',Start,Eat).
trans(t2,'drinkSome',Start,Drink).
trans(t3,'eatMore',Eat,Drink).
trans(t4,'drinkMore',Drink,Eat).
trans(t5,'stop',Eat,Stop).
trans(t6,'stop',Drink,Stop).

true.

3 ?- halt.
```

As before, you can develop M2M files interactively using a 3-window arrangement: one window for the database, another for the M2M file, and a third running SWI-Prolog.

4.3.3 Velocity (VM2T) Files

Weve already seen a velocity file. To invoke velocity from a command line, type:

```
>java vm2t.Main
Usage: vm2t.Main prolog-file template-file
       [output-file] [-cg ContextGeneratorClass]
```

if `output-file` is unspecified, output is directed to file `vm2t-output.txt` And to create the output file `x.java`, type:

```
> Java vm2t.Main db.pl xlate.vm x.java
```

Like Prolog files, you can develop velocity files interactively using a 3-window arrangement: one window for the database, another for the velocity file, and a third to execute Velocity and to view its output.

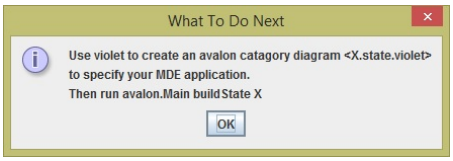
5. Creating an MDELite App Using `Avalon`

To create an MDELite application, perform the following steps:

1. Let `X` be the name of your MDE application. Run:

   ```
   >java avalon.Main help X
   ```

   The output is this message: Followed by a start up of Violet,

   for which you are to draw a category diagram of `X`. By default, Violet will store your design in file `X.state.violet`.
2. Next, run:

   ```
   >java avalon.Main buildState X
   ```

   And follow its instructions. Basically for every Prolog database (*ie,* a domain whose extension is '`pl`'), you are to draw its UML class diagram in Violet. `Avalon` computes its prolog schema file among other files, and if no errors are encountered, generates in directory `boot/` the files that you can drag into a NetBeans or Eclipse project to complete the development of application `X`.

Doing all of this, a directory called **boot/** is created in the same directory as `X.state.violet`. This will contain the shell of a Netbeans (or Eclipse) project from which you can:

- supply real Prolog conform.pl files in directory boot/libpl
- supply real .vm velocity scripts in directory boot/libvm
- supply .java files for unwritten Java arrows

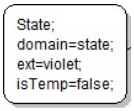
More details are in the following sections.

5.1 Creating a Category Diagram of Your Application

Start Violet[4] and create a state chart diagram. A category diagram is an annotated state diagram. Each bubble/state is a domain; each arrow `A: B` $\rightarrow$ `C` is a transformation that maps documents from domain `B` to co-domain `C`.

- For each domain, specify its corresponding Java class name, the name of the domain (which might be the same as the class name), the name of its file extensions, and whether or not a file/document of this domain should be retained.

  The figure below shows the bubble for the `State` domain. It's Java class is named "`State`", its domain name is "`state`", file extension is "`violet`", and it is not a temporary (deletable) file.

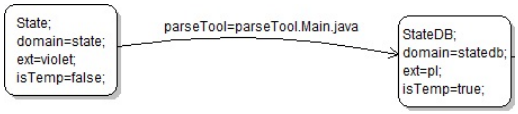
  

  Note: Prolog database domains requires the constraint `lowercase(Java class name) = domain name`. If this is not the case, `avalon` should flag this as an error for you to fix.

[4] `>java Utils.Violet`

- For each arrow, specify its name, followed by =, followed by the executable that implements this arrow.
  - For each Java arrow, the application name (*eg,* `Prg.Main`) followed by `.java`, *ie,* '`Prg.Main.java`'.
  - For a prolog arrow `A : B → C`, the name of the prolog file is `libpl/b2c.pl`, where `b` is the domain name of `B` and `c` is the domain name of `C`. That is, `b2c.pl` has a predetermined name and it is stored in the `libpl/` directory of application `X`.[5]
  - For a velocity arrow, the name of the velocity script is `libvm/N.vm`, where `N` is any name, there are no restrictions.

  The figure below shows a declaration of arrow `parseTool`, which is implemented by the Java program `parseTool.Main`. We have already discussed in Section 4.2 the protocols that MDELite uses for calling such programs.



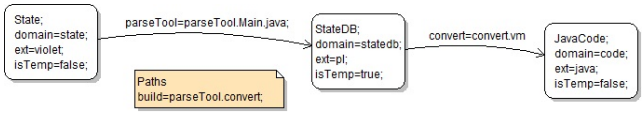
  Note: Violet doesn't allow you to draw multiple arrows between two nodes – if it does, the arrows simply overlap making an unreadable mess. To specify that there are two or more distinct arrows and their executables, use one arrow label for both declarations separated by '`;`' as below where two arrows, `parseTool` and `tool2`, are declared:

```
parseTool=parseTool.Main.java; tool2=xyz.Main.java
```

- A computation of application `X` is a path (sequence of arrows) in the application's category diagram. For each path, list its name, followed by '=', followed by a sequence of arrow names separated by dot ('.'). A path defines the order in which program/arrow executions are sequenced.

  There is only one path in our running example, `build`, which executes `parseTool` followed by `convert`. Any errors will halt a path execution.

The full Violet category specification is file `fsm.state.violet`:



The next step is for you to fill in the schema declarations for all prolog databases. This is done by executing:

```
>java avalon.Main buildState fsm
```

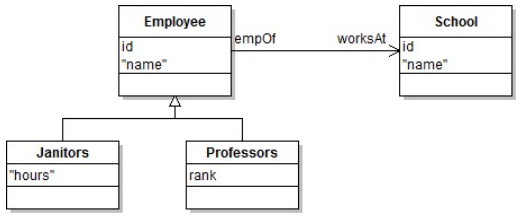
The generated files will be in directory `boot/`.

5.2 Creating a Schema Diagram for a Prolog Database

A Violet class diagram is used to define database schemas. The rules are simple:

- Each relation is drawn as a class. Attributes that are specific to that relation are listed; no methods should be declared.
- The first attribute of every relation should be an identifier (`id`) field. There is nothing at present that MDELite assumes about the first attribute, but all examples that we use have `id` as the first attribute of every relation.
- Attributes are one of two types: either their values are single-quoted or not. If they are to be quoted, the name of the attribute is in double-quotes.
- If you use associations, only labeled arrowheads are noticed. The label is the name of the attribute to be added (to the class from which the association points). All other labels and associations are ignored.
- If you use inheritance, only the top (root) relation has an identifier. All parent attributes will be propagated to child relations.

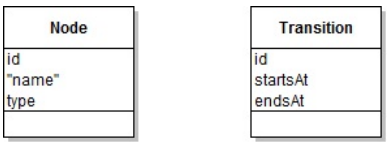
As an example, the figure below depicts a schema with inheritance and associations in file `theworks.class.violet`:



The schema that is generated is:

```
dbase(theworks,[janitors,professors,employee,school]).

table(janitors,[id,"name",worksat,"hours"]).
table(professors,[id,"name",worksat,rank]).
table(employee,[id,"name",worksat]).
table(school,[id,"name"]).

subtable(employee,[janitors,professors]).

tuple(janitors,L):-janitors(A1,A2,A3,A4),L=[A1,A2,A3,A4].
tuple(professors,L):-professors(A1,A2,A3,A4),L=[A1,A2,A3,A4].
tuple(employee,L):-employee(A1,A2,A3),L=[A1,A2,A3].
tuple(school,L):-school(A1,A2),L=[A1,A2].

janitorsALL(A1,A2,A3,A4):-janitors(A1,A2,A3,A4).
professorsALL(A1,A2,A3,A4):-professors(A1,A2,A3,A4).
employeeALL(A1,A2,A3):-employee(A1,A2,A3).
employeeALL(A1,A2,A3):-janitorsALL(A1,A2,A3,_).
employeeALL(A1,A2,A3):-professorsALL(A1,A2,A3,_).
schoolALL(A1,A2):-school(A1,A2).
```

For our running example, I tend to avoid associations and occasionally use inheritance. Here's my definition of the `StateDB` schema:



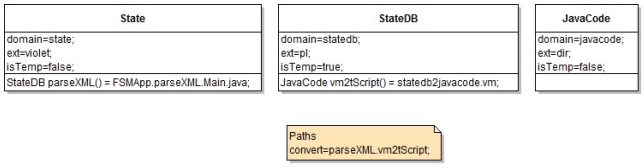
5.3 An Alternative to using Category Diagrams

Until now, we have used category diagrams to specify an `Avalon` application. As an alternative, which underscores the idea that MDE is OO metaprogramming, you can use a class diagram. Each domain corresponds to a class, and each arrow corresponds to a method.

Here is the Violet class diagram that could be used in place of a category diagram. It contains the same information, arranged

[5] Eventually I will remove this restriction.

and presented differently, but ultimately maps to the same internal Prolog derivation.



This is stored in file fsm.class.violet, and can be used to build the FSM application by typing:

```
>java avalon.Main buildClass fsm
```

5.4 Finally, Fielding Your Tool

When you are creating your tool in Netbeans or Eclipse, in your Netbeans/Eclipse project you will have `src/`, `libpl/`, and `libvm/` directories. And when your project executes, MDELite will look in these directories to find your `libpl/` and `libvm/` files, and all should be well.

But when you are about to field your tool for others to use, you will have to create a single directory with the following contents:

- `tool.jar` – the jar file of your tool
- `lib/` – a set of Jar files that your tool needs. Netbeans supplies this directory under directory `dist/` whenever you compile.
- `libpl/` – a copy of your `libpl/` directory
- `libvm/` – a copy of your `libvm/` directory
- `Docs` – your directory where you keep your tool documents

It is a real pain to create all of this manually. Netbeans has a postjar action which you can add the following xml code – which you need to customize to your specific tool:

```
<target name="-post-jar">
  <copy todir="dist/libpl">
    <fileset dir="libpl"/>
  </copy>
  <copy todir="dist/libvm">
    <fileset dir="libvm"/>
  </copy>
  <copy todir="dist/lib">
    <fileset dir="NeededJarFiles"/>
  </copy>
  <copy todir="dist/Docs">
    <fileset dir="Docs">
      <include name="*.pdf"/>
    </fileset>
  </copy>
  <delete file="dist/lib/MDELite3_05.jar"/>
  <delete file="dist/lib/commons-io-2.4.jar"/>
</target>
```

Here is what the above means:

- copy the `libpl/` directory into the DIST/ directory on every build.
- copy the `libvm/` directory into the `dist/` directory on every build.
- your project may need jar files to run. I create a `NeededJarFiles/` in my Netbeans project in which I place all of them. On every build, I copy these `NeededJarFiles/*` into `dist/lib/`.
- any `.pdf` files in my `Docs/` directory I copy into the `dist/Docs/` directory on every build.
- delete any jar files from `dist/lib/` that are already on my classpath (so that I don't have 2 or more inconsistent copies).

If I could figure out a way to hide all of this "stuff" from `avalon` users, I would be delighted. Any thoughts or comments are appreciated.

6. How `Avalon` was Bootstrapped

`Avalon` is itself an `Avalon` application. It has a category diagram, as other `Avalon` applications. Its diagram is shown below. I know this is really small, but it is a digitally enlargeable figure.