Catalina Manual

September 14, 2014

MDELite is a proposal to make tools for developing MDE applications – at least in a classroom setting – easier to use and less heavy-weight than Eclipse. Catalina is its 2nd-generation. MDELite was a code generator; Catalina is an interpreter. The difference in infrastructure is a growth from MDELite's miniscule core ~500 LOC to Catalina's ~5600 lines. Of course, there is quite a difference in appearance, but whether these extra capabilities were worth the effort remain to be seen.

This document presents a quick overview of Catalina, its fundamentals, and how to use it. Installation is described in a separate document.

1. Fundamentals: Part 1

MDE tools are normally based on class diagram metamodels. That is, you create a class diagram to define your metamodel, and voila! A customized tool is produced to allow you to draw instances of your metamodel, complete with automatic constraint checking. Of course, you need to provide the conformance rules for the class diagram, but this is expected.

Catalina is based on metamodels of categories. An MDE application is a set of domains and arrows (a.k.a. transformations). For each domain, you may need to define its class diagram and constraints (much like MDE tools above). Cosmically, the overall process is similar, except that the definition of an MDE application has been elevated from the structure of an individual domain to, basically, a program, rather than just a module in a program.

There are three sets of tools in a Catalina distribution:

- **Core** – the framework (NOT an MDE application)
- **Catalina** – a tool that allows engineers to define a category (a Catalina MDE application)
- **MDE** – a tool that encodes the original MDELite application, in terms of Catalina tools. Another Catalina MDE application.

The metamodel for Catalina is shown below:[1] It is drawn in the Violet UML diagram editor as a "state diagram" that has nodes, arcs, and notes.

[1] This is actually a simplification of the real Lite category: there are many more domains (intermediate results) and arrows, but the essential concepts are the same.

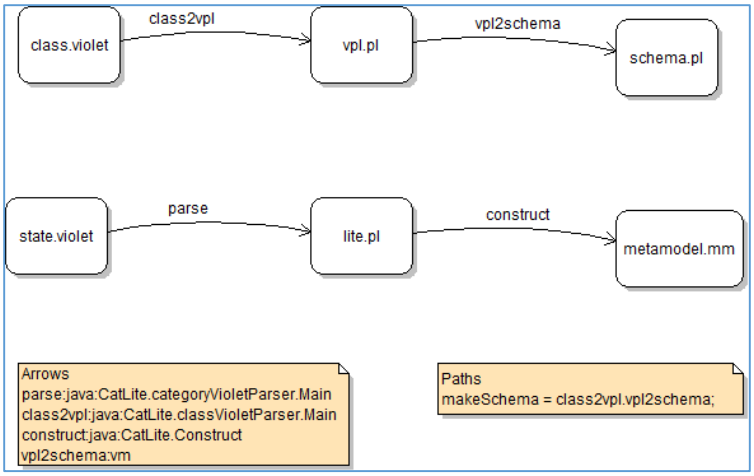
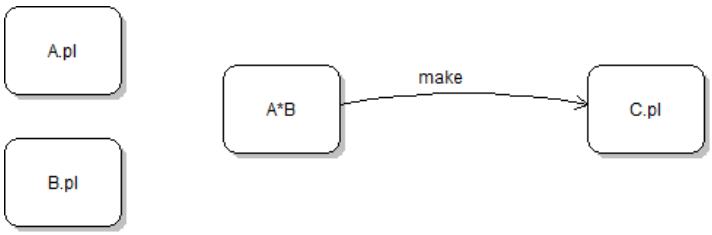


Figure 1 Catalina MetaModel

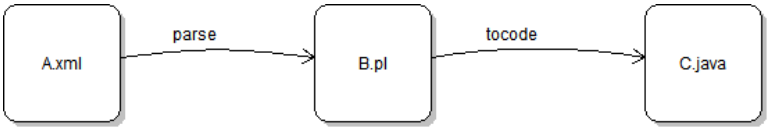
Each rounded-edge box is a domain; each directed edge is an arrow -- a function from a domain to codomain. Arrows with multiple input domains are represented as cross-products of domains, as in (`A*B`) in the figure below, which is the cross product of the domain of `A` and `B`. Arrow $make$ maps $(a, b) \in A \times B$ to $c \in C$:



Here's a tour of the Catalina metamodel. When you see a category drawn in this fashion (which, by the way, is called an **external diagram** of a category), just think of it as a flowchart. Nodes are placeholders for values and arrows are programs that take node inputs and produce node outputs.[2]

Everything starts by defining a category: the domains and arrows that define an MDE application. As a general rule, every arrow is a stand-alone program, typically a Java program, but not always. An instance of every domain is a file (or in special cases, a file directory). Every domain has a name and a file extension, which is used to distinguish files from different domains. In fact, all file names in Catalina are triples, $N.D.E$, where N is the name of an application; D is a domain name, and E is an extension.

> **Example**: the file `app.lite.pl` belongs to application '`app`'; in particular it is the domain '`lite`' representation, which is of type '`pl`' (Prolog). To see why this naming scheme is needed, consider the category below:



[2] At present, Catalina does not work for multiple outputs. In the interim, an arrow of the form $\rightarrow \alpha: X \rightarrow A * B$ is written as a pair of arrows $\alpha 1: X \rightarrow A$ and $\alpha 2: X \rightarrow B$.

It has 3 domains (A, B, C) and two arrows ($parse, tocode$). Arrow $parse$ transforms an A instance into a B instance. Arrow $tocode$ transforms a B instance into a C instance.

Suppose we have two instances of domain A that we want to transform into code, namely `app1` and `app2`. Their files are `app1.A.xml` and `app2.A.xml`. Program/arrow $parse$ converts `app1.A.xml` into `app1.B.pl`, and program/arrow $tocode$ converts `app1.B.pl` into `app1.C.java`.

Similarly, $parse$ and $tocode$ map `app2.A.xml` to `app2.B.pl` and then to `app2.C.pl`. Each application (e.g. `app1`) has 3 different representations: a domain A representation (`app1.A.xml`), a domain B representation (`app1.B.pl`) and a domain C representation (`app1.C.java`). Every application has multiple representations. This file naming scheme allows you to recognize files that belong to an application and to distinguish their different representations.

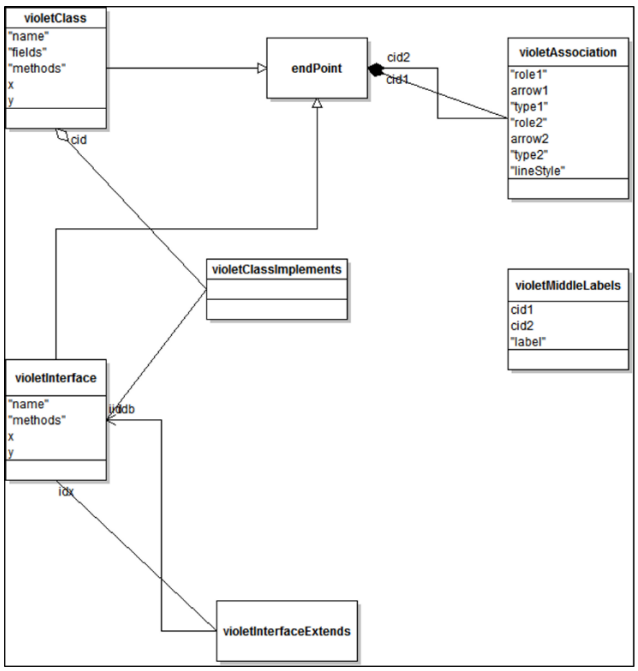
Finally in a Catalina category specification, every domain has a name and an extension. Instances of domain A, above, are `xml` files. Instances of domain B are Prolog files (with `pl` extensions). Instances of domain C are `java` files. So as a computation proceeds in Catalina, a series of files with the same prefix are produced (ex. "`app1`"), each having a distinct 2-part suffix (e.g. "`B.pl`").

Let's now return to Figure 1 to interpret it. Basically it says: if you want to create a MDE application, you have to define its category, which is instance of domain "`state.violet`". I use Violet, a free UML diagram drawing tool, to draw Figure 1. It produces a file N`.state.violet`, where N is the name that I gave to my application. Violet always produces xml files with suffix "`.violet`".

The first transformation ($parse$) maps a violet state file (which is an XML document) to an instance of a Prolog database of type `lite.pl` (the name of the codomain of $parse$).[3] The schema for `lite` databases has yet to be defined – we'll deal with that shortly. So when arrow $parse$ is executed, it maps file `N.state.violet` to `N.lite.pl` (a `lite` Prolog database). Similarly, transformation $tocode$ maps `N.lite.pl` to a Java file `N.C.java`. That's how to think about arrows and their inputs and outputs.

Every category has a set of database domains – these are recognized by domains with "`pl`" extensions. Three database domains appear in Figure 1. Namely, `lite`, `vpl`, and `schema`. Every database in Catalina needs a schema. A schema is specified Figure 1 as Violet class diagram, from which a Prolog schema is computed. A violet class diagram is an XML file that is an instance of domain `class.violet`. Its Prolog schema is an instance of domain `schema.pl`. As an example, Figure 2 Schema Specificationsbelow shows the Violet class diagram for `vpl` and its derived database schema (`vpl.schema.pl`). Again, I draw the pretty Violet picture on the left, and its schema is automatically computed on the right using program $parse$ (which I have to write). You'll see how this computation is invoked soon.

[3] Violet does not distinguish standard "state chart" drawings from "Catalina category" drawings. Both such drawings are stored in `.state.violet` files. You, as a designer, should remember which files are which.




```
dbase(vpl,[endPoint,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","lineStyle",cid1,cid2]).
table(violetMiddleLabels,[id,cid1,cid2,"label"]).
table(endPoint,[id]).

tuple(violetClass,L):-violetClass(A,B,C,D,E,F),L=[A,B,C,D,E,F].
tuple(violetInterface,L):-violetInterface(A,B,C,D,E),L=[A,B,C,D,E].
tuple(violetAssociation,L):-violetAssociation(A,B,C,D,E,F,G,H,I,J),
                            L=[A,B,C,D,E,F,G,H,I,J].
tuple(violetMiddleLabels,L):-violetMiddleLabels(A,B,C,D),
                             L=[A,B,C,D].
tuple(endPoint,L):-endPoint(A),L=[A].


% inheritance relationships

endPointI(A1):-violetClass(A1,_,_,_,_,_).
endPointI(A1):-violetInterface(A1,_,_,_,_).
```

Figure 2 Schema Specifications

Given schemas for all Prolog databases, the last thing to specify in a category is how to implement each arrow. That's the purpose of the notes in Figure 1, which I reproduce below:

```
Arrows
parse:java:CatLite.categoryVioletParser.Main
class2vpl:java:CatLite.classVioletParser.Main
construct:java:CatLite.Construct
vpl2schema:vm
```

```
Paths
makeSchema = class2vpl.vpl2schema;
```

Each arrow has an entry in the **Arrows** note: its name, type, and executable. Arrow *parse* is the first listed. It is a **java** executable (that's the purpose of the "`:java:`" syntax) and the executable is the Java program "`Catalina.categoryVioletParser.Main`" that someone has to write. According to the category diagram, it takes a violet `.state.violet` file as input and produces a `vpl` database as output (whose Prolog schema was just discussed – see Figure 2).

Now skip on down to the `vpl2schema` arrow. It is a **Velocity** template (designated by "`:vm`" and by default whose file is "`vpl2schema.vm`") that defines a model-to-text mapping. According to the category diagram, it takes a Prolog `vpl` database and this template (`vpl2schema.vm`) as input, and produces an instance of the `schema` database as output.

The **Paths** note defines arrow compositions that are convenient. Only one is defined, called `makeSchema`. It is the composition of arrows `class2vpl` and `vpl2schema`. Invoking the `makeSchema` arrow is equivalent to invoking `class2schema` and `vpl2schema` in that order.

At present, Catalina arrows are of type `java` (java executables), `pl` (Prolog database-to-database mappings), `vm` (Velocity templates), or `exe` (any Windows executables).

To recap, a Catalina application is defined by a Violet category diagram, which contains domains and arrows. Each domain is typed – is it a violet xml document? is it a Prolog database? Is it a text file? If it is a Prolog database, you will have to draw a Violet class diagram for Catalina tools to compute its Prolog schema. For each arrow, you must specify the executable (program) that maps arrow inputs to the arrow output. Unless you are lucky, you will have to write a program for each arrow. As a general rule, an arrow program has the command line arguments:

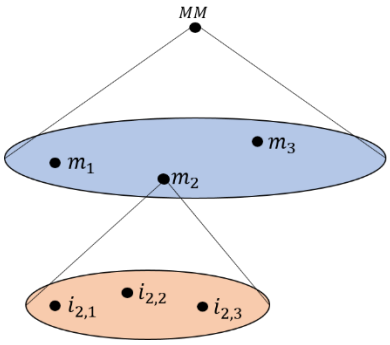
```
> arrow_program.exe one-or-more-file-inputs lone-file-output
```

The computation of a Catalina category is to transcribe all of this into an internal (Prolog database) form that is suitable for execution by the Catalina interpreter.

The next section presents other fundamental ideas behind Catalina. After that, we present the Catalina GUI to specify these computations, and the last sections will step through an example Catalina application.

2. Fundamentals: Part 2

A fundamental idea in MDE is that of a **metamodel (MM)** and its model instances. A MM is a template for stamping out instances, much like a Java class definition can be used to stamp out its objects (i.e., class instances). Diagrammatically, we show this as a cone of instances: the MM is the root and its domain of instances, labeled $dom(MM)$, is shown by the population of its models in the large blue oval. In the figure below, m_1, m_2, m_3 are instances of $dom(MM)$. The m_i can themselves be metamodels, which have their own cone (domain) of instances, such as $i_{2,1}$, $i_{2,2}$, $i_{2,3} \in dom(m_2)$ in the figure below.



In *any* MDE implementation, you need a tool $\mathcal{T}(x)$ to create instances of a metamodel x. Two tools are commonly used today. One allows users to define their own class diagrams – we call this tool $\mathcal{C}(x)$ given metamodel x. Another allows users to create object diagrams – we call this tool $\mathcal{O}(x)$, for metamodel x Thus, in the above figure, users create any of the m_i using tool $\mathcal{C}(MM)$. They then create instances of these models, such as m_2, using $\mathcal{O}(m_2)$.

Ecore is the metamodel of the **Eclipse Modeling Framework (EMF)** that allows MDE designers to graphically represent their models. Below is the **meta-object facility (MOF)** hierarchy:

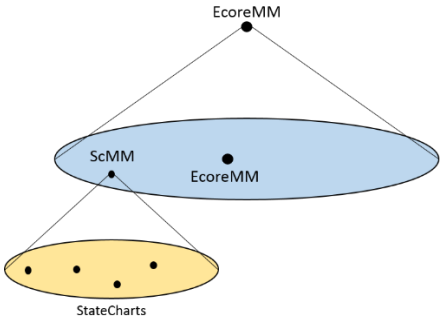


Figure 3 MOF Hierarchy

The root of this diagram, $EcoreMM$, is the Ecore metamodel – the metamodel whose instances are class diagrams. In true MDE style, $EcoreMM \in dom(EcoreMM)$ – that is, $EcoreMM$ is an instance of itself. The Diagram Editor is the main model drawing tool in Ecore. This Editor seems to be the union of at least 3 distinct tools: one that allows users to draw class diagrams (our $\mathcal{C}$ above), another to manually create (not draw) instances one instance at a time ($\mathcal{O}_1$ above), and a third that allows users to draw object diagrams of a given MM ($\mathcal{O}_2$ above). For teeny models, $\mathcal{O}_1$ is sufficient. Otherwise $\mathcal{O}_2$ is preferred.

> **Question**: has $\mathcal{C}(x)$ ever been invoked with $x \neq EcoreMM$? My guess is "yes" or ***it could be*.** Imagine the domain class diagrams that have no associations, $dom(Cat)$.[4] I can use $\mathcal{C}(EcoreMM)$ to create its metamodel Cat. In principle, I could then use $\mathcal{C}(Cat)$ to create any instance in the domain of $dom(Cat)$.

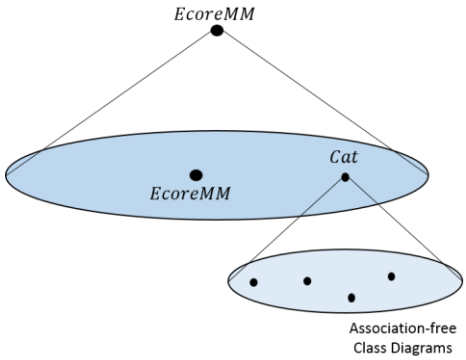


Figure 4 Another MOF hierarchy

Categories have the following interesting property. The **external diagram** of a category (where nodes are domains and edges are arrows) is a multi-graph: a graph where multiple edges can connect the same nodes – see Figure 5a. An **internal diagram** of a category shows the cone of instances of selected domains and the arrows among their instances – see Figure 5b. Both the external diagram (Figure 5a) and the diagram that is formed by instances and their arrows (green in Figure 5b) are categories, i.e. multi-graphs.

[4] Such things are indeed useful – they are called **categories** – a concept that we have already encountered in this document.

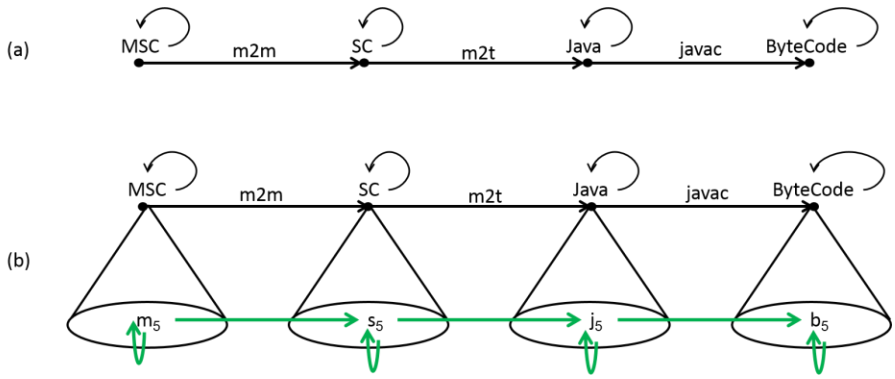


Figure 5 Categories

This means that a tool, whose metamodel is based on categories, rather than class diagrams, has the following hierarchy:

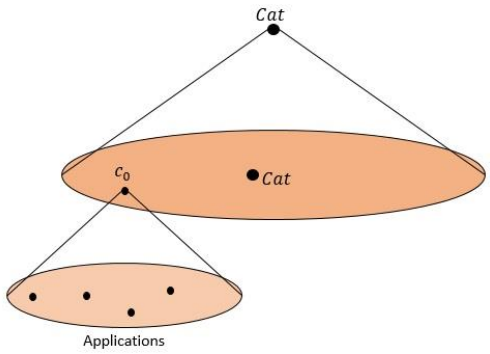


Figure 6 Category Hierarchy

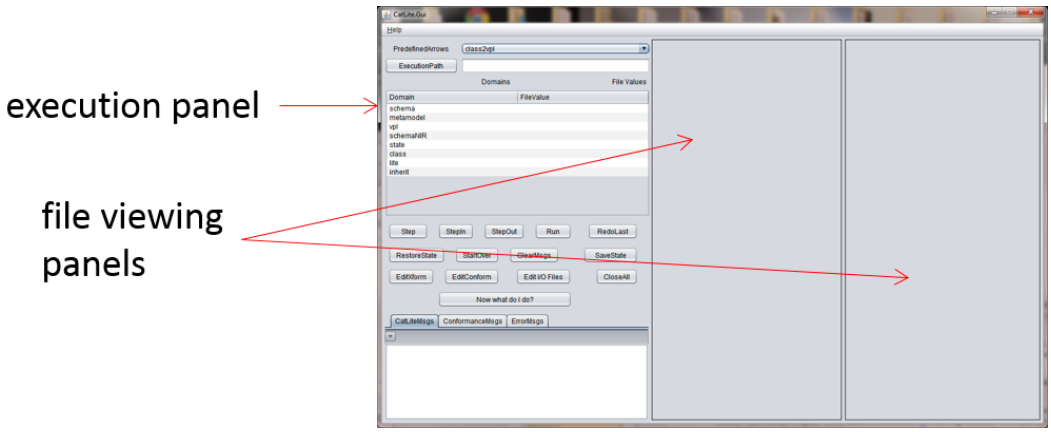
A single tool, $\mathcal{K}(x)$, can create instances of domain $x \in dom(Cat)$. A user would specify his/her MDE application c_2 as a category, using $\mathcal{K}(Cat)$, and then would instantiate that application using $\mathcal{K}(c_2)$. This is fundamental to Catalina tools.

3. The Catalina GUI Tool

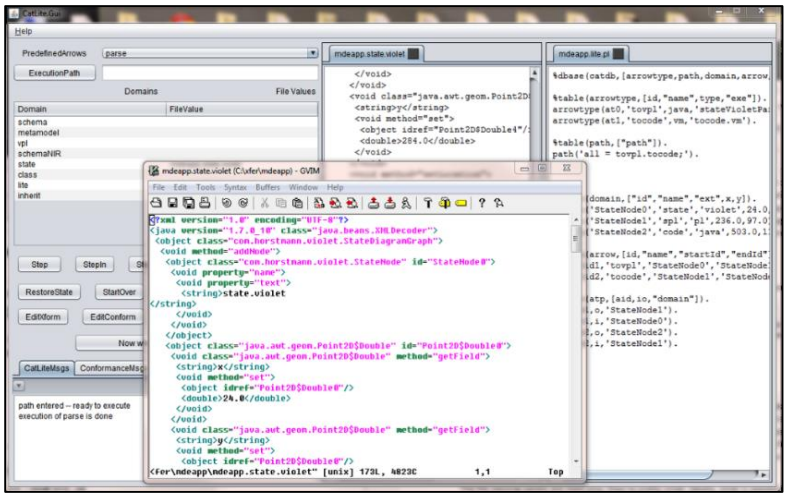
Now let's take a quick tour of the Catalina Gui tool, a prototype for an eventual IDE plug-in. The GUI can be invoked from a command line by:

```
> java CatCore.Gui
```

whose graphical front-end is shown below:

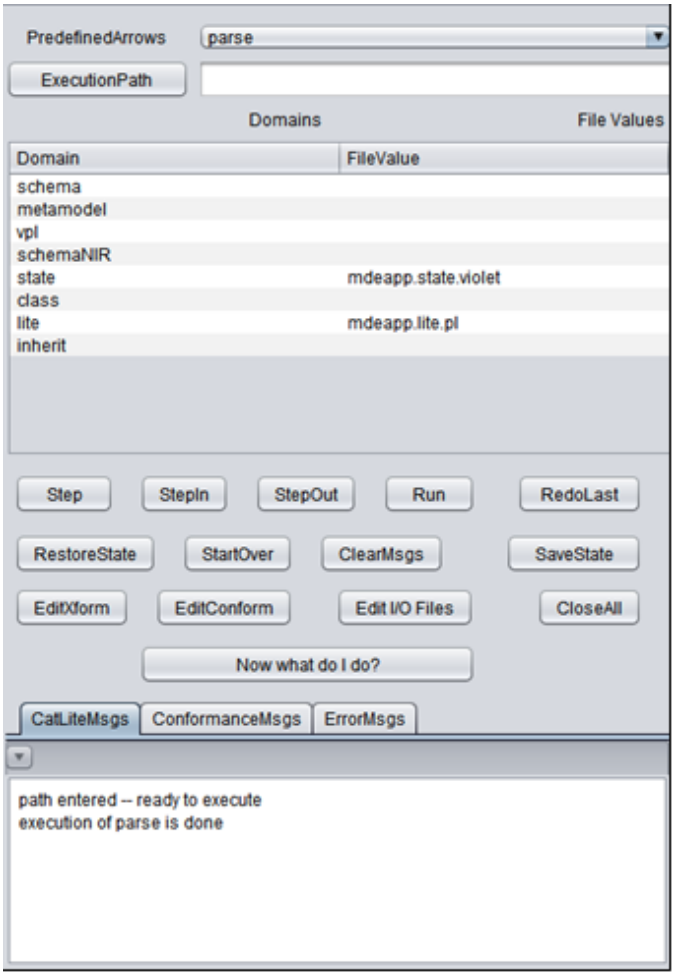


There are 3 vertical panels: the left displays a set of buttons and tables to execute the arrows of a category. The other panels are for file viewing. The idea is this: when an arrow is executed, you (as a MDE designer) want to see the files that were input and the output that is produced. The middle panel shows (in separate tabs) each file that is input. The right-most panel displays in a single tab the file that was output. So during a multi-arrow execution, you can step through the execution one arrow at a time and view individual arrow inputs and outputs. I have found this useful in debugging categories.



The file viewing panels are read-only: they're pretty small. Ideally, what is needed is a file editor, which can give a designer a general-purpose way to explore, and possibly edit, a file. Invoking such an editor is accomplished by *right clicking the tab of the view file*. Catalina currently uses GVIM as its editor. The figure below shows a GVIM application for editing/exploring a violet category file.

The fun part of CatGui is the Execution panel, whose close-up is shown in the figure below.



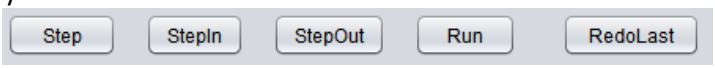
The top part allows a designer to select an arrow to execute. A pull-down menu lists defined primitive arrows or paths in a Catalina category specification. Alternatively, an ad hoc path can be typed into a text field to be executed.

Arguments to an arrow are specified by entering them in the **domain table**, which in the above figure has a row for each non-cross-product domain of Figure 1. You can click on a domain to choose the file instance for that domain. In the figure, the `state` domain has instance `mdeapp.state.violet` and the `lite` domain has instance `mdeapp.lite.pl`. When an arrow is executed, its arguments are taken from this table. If an argument is missing – the tool tells you the arrow can't be executed in the **CatalinaMsgs** tab at the bottom of the panel. Conformance errors are displayed in the **ConformangeMsgs** tab; execution errors (something that is really wrong) appears in the **ErrorMsgs** tab.[5]

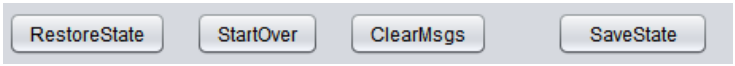
Now, let's take a look at the button panel. The first row deals with arrow execution. Once you have selected an arrow and entered its inputs, you can execute it by clicking **Step**. Often one enters a path (a multi-arrow execution). In such cases, you'd like to step through its execution one primitive arrow at a time. That's the purpose of **StepIn** – it expands a path into its constituent arrows. The **StepOut** button executes the remaining arrows in that path. **Run** executes the entire path. **StepIn** is the most useful of

[5] What could be an "error"? A file could be corrupted. All arrows (executables) assume basic conditions are satisfied – in effect, these are the conformance tests that are applied to a model. But during development, it is easy to forget an essential condition (or to test for it), so errors do arise, should be reported, and that need to be fixed.

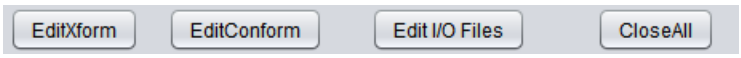
these buttons; the **RedoLast** button is the next most useful. It says re-execute the last arrow. Before you re-execute an arrow, you can modify its input files. This is helpful for interactive development. I'll explain how this is done shortly.



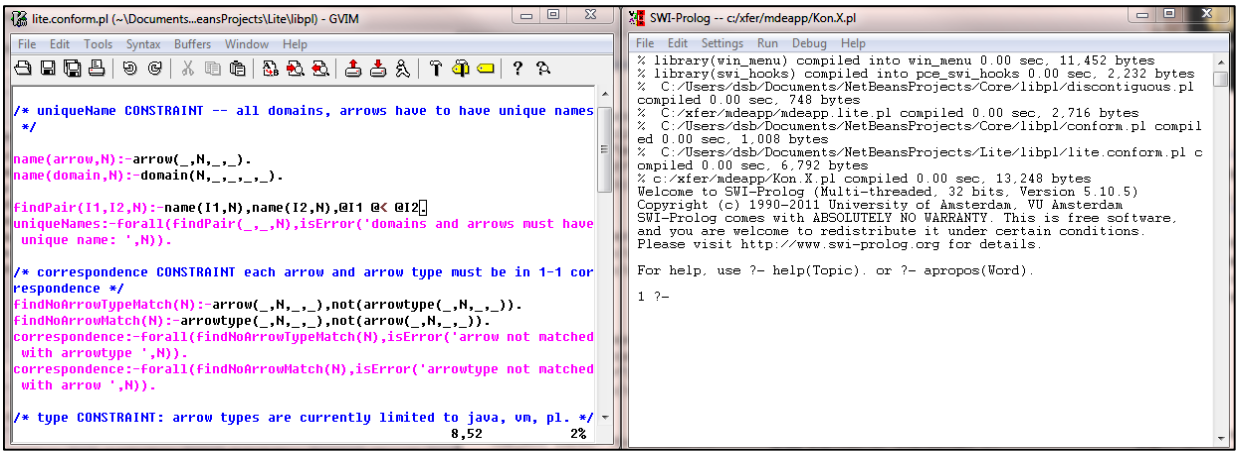
The second row of buttons save you a lot of clicking. Initializing the domain table for a test and selecting an arrow to execute takes time. **SaveState** saves the content of the domain table. **RestoreState** restores the last saved content. **StartOver** clears the table and selected arrow. **ClearMsgs** clears the message tabs at the bottom of the panel.



The third row of buttons is useful for editing files. Clicking the **EditXform** button opens a GVIM window to allow you to edit the last Prolog or Velocity transformation that was executed.



In the case of a Prolog transformation, a pair of windows is opened: a GVIM window containing the Prolog text and another window running SWI-Prolog (see figure below). This is a useful configuration where one can edit a file, reload it in SWI-Prolog and re-execute it, enabling the interactive development of a Prolog database transformation or Prolog database conformance tests. Essentially, it is a pair of windows that a designer needs to interactively develop and debug a Prolog application is presented. See the refresh note discussed later.

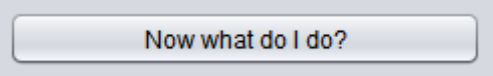


For arrows implemented by Velocity, a single GVIM window opens with the text of the Velocity template. You can edit, save your edits, and click **RedoLast** to re-execute the Velocity arrow. You can view the output of the arrow in one of the file view windows, or start a GVIM window to read it (discussed earlier).

As for Java and other executables, not much can be done. If you discover a problem, in that the output is incorrect, you have to use an IDE to debug your (Java) program. This is the least fun part of Catalina.

The **Edit I/O Files** button will open a GVIM editor for every input file and the output file of an arrow. **CloseAll** closes all editor and SWI windows that were created. (To avoid window pollution, Catalina tools close all opened windows before executing the next arrow).

Perhaps the most valuable button for a novice is the "**Now what do I do?**"



I find it bewildering to look at a category diagram (a.k.a. megamodel or tool chain diagram) to know where to begin its computations. As part of metamodel development, I ask that designers write a simple program to give advice on how to use the and how to walk novices through the process. The next section illustrates how to use Catalina and this magic button.[6]

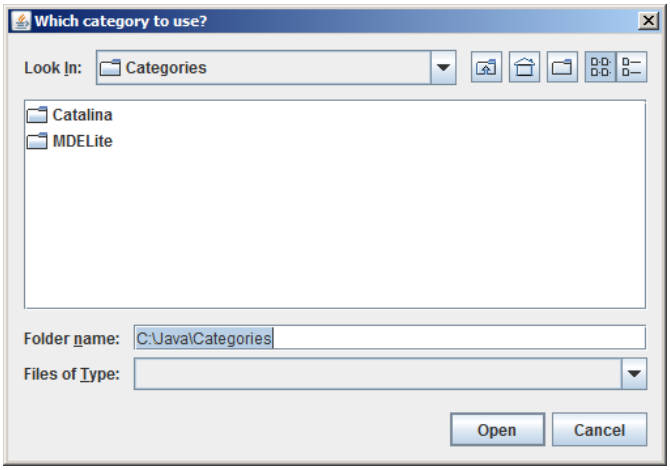
4. Development of a Catalina Application: Part 1

Here is your assignment. You want to create an MDE application using Catalina that will transform a beautiful state diagram into Java source code. Here's how to do it.

a. Create an empty working directory and run `CatCore.Gui` inside it.

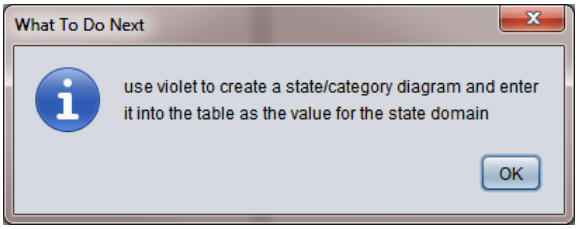
```
>> mkdir myapp
>> cd myapp
>> CatCore.Gui
```

The Gui asks you (via a dialog) which metamodel you want to use. Select "Catalina" as you are going to create a new MDE application.[7]



b. Now, you're ready to proceed and don't know what to do. Answer: click the "**Now what do I do?"** button. The response is:

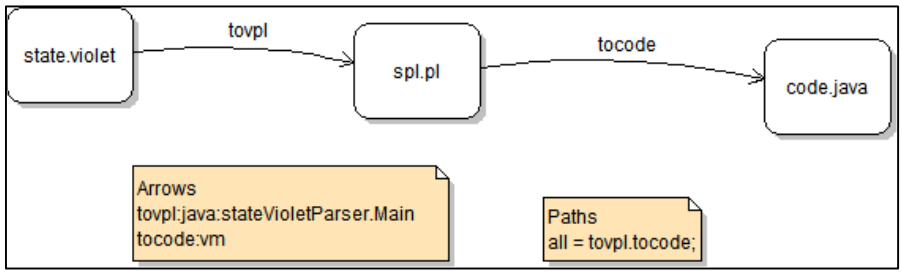
[6] Writing an Advice program is tedious. I think its development can be automated. I'm trying to figure out how now.
[7] Only Lite and MDE are Catalina applications. Normally, all you want to do is to execute an MDE application with new input files. You select Lite when you want to create a new MDE application, not run an existing one.



Thus, to create a Catalina MDE application, you have to draw its category diagram using Violet. Not only that, but the domain table is highlighted to tell you what row (`state`) to fill in:

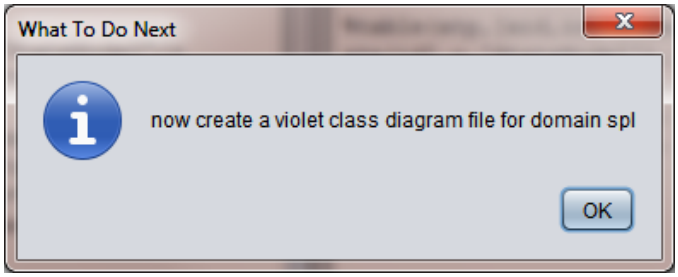
Domain	FileValue
schema	
metamodel	
vpl	
schemaNIR	
state	
class	
lite	
inherit	

c. Recall your task is to develop an MDE application to translate a state diagram to Java code. The violet category specification for this application is shown below:

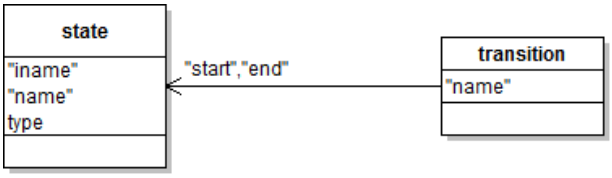


That is, you want people to draw a state diagram using violet, have arrow `tovpl` convert a diagram into a beautiful Prolog database and then have arrow `tocode` to produce Java code. Ok, but why this sequence? A Violet state diagram is an ugly xml file. (You should look at the files Violet produces). You want a representation of it that is easy to understand and for which you can write constraints in Prolog. That's the purpose of the `spl` database in the above category diagram. Further, the Velocity tool used in Catalina is tailored for Prolog database input. So a Velocity template is used to stamp out Java code given an `spl` database. Hence this category.

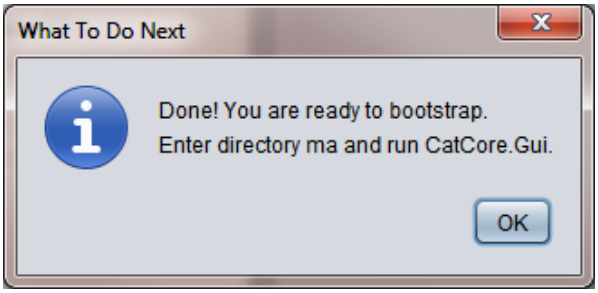
So now you draw the above category diagram, save it in file `myapp.state.violet`, and enter its name into the domain table. Not knowing what to do next, you click "**Now what do I do?**" The advice program responds:



d. To spell it out: if I want to produce a database of Prolog facts, you need to define its schema. As Catalina requires some consistency in writing and specifying schema, it asks you to draw the class diagram of its schema in Violet, and Cat tools will produce its schema. So you draw the diagram which encodes all of the facts I will need about state diagrams:



I enter this violet class diagram into the domain table, and click "**Now what do I do?**" The response:



e. Look inside the generated directory `myapp`. There are several files, one of which is `README.txt` with contents:

```
his directory contains the core definitions of the myapp metamodel.
It is still incomplete. Follow these instructions:

1) rewrite dummy libpl/spl.conform.pl
2) write java application stateVioletParser.Main and place it on your CLASSPATH
3) write the Velocity (M2T) file libvm/tocode.vm
4) post this completed directory in C:/Java/Categories for others to use
5) optionally, write an advice program to make your tool easier for others to use
```

To spell it out: you have to implement each arrow in my category and to write metamodel constraints for each Prolog database. (No surprise). There are three programs to write:

1. A Java program, which was defined in the category spec as "`stateVioletParser.Main`", to parse a violet state XMLfile into a Prolog database whose schema is in `libpl/vpl.schema.pl`, one of the generated files. Its partial contents are shown below:

```
dbase(spl,[transition,state]).
table(state,[id,"iname","name",type]).
table(transition,[id,"name","start","end"]).
```

The above says the `stateVioletParser.Main` will produce tuples for two tables: `state` and `transition`. The `state` table has 4-tuples: manufactured identifier (`id`), an internal name given by Violet (`iname`), a name that the user gave to the state (`name`), and its `type` (`start` – for start state, `end` – for an end state, `normal` – for all other states). The

latter enumeration is not obvious – but it was to me (as I knew exactly the meaning of each column, although Catalina schemas are a bit too primitive to capture such design information.

The `transition` table also has 4-tuples: a manufactured identifier (`id`), the name of the transition (`name`), and the user-given names of the starting (`start`), and ending (`end`) state. Again, the meaning of these attributes is what I intended when I defined `spl.state.violet`. The above schema was produced when this file was "compiled".[8]

When you write `stateVioletParser.Main`, it must take a `state.violet` (xml) file as input, and produce tuples according to the `spl` schema. This is not a difficult task, but it is tedious. This is a legacy decision of MDELite. Presumably, there are parsers (and predefined schemas) that could be reused. Creating a repository of such things will be a task for the future, so that little or no Java code needs to be written.

2. Replace the dummy `libpl/spl.conform.pl` file with Prolog constraints that you would expect all state machines to provide – like no two states have the same `name`. You can use CatGui to develop such constraints interactively, as mentioned previously.

3. Replace the dummy Velocity template `libvm/tocode.vm` with something real. Again, you can use Cat Tools to develop such files interactively.

f. Finally, if you want others to use your MDE application, you should write a Java Advice program to tell people how to use your tool. This is a bit more advanced than I want to consider at this point. All any user has to do is supply a violet `state.violet` file as input, and your tool will generate its equivalent Java code.

5. Development of a Catalina Application: Part 2

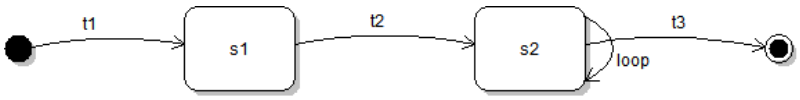
This is the fun part – writing conformance tests, Prolog M2M (database-to-database) translations, and Velocity (vm) files.

> **Note**: The ONLY painful task is writing Java code – if you can make due with using only Prolog and Velocity, you will be that much further ahead. Originally, MDELite stopped at this point, although there was indeed room for writing Java programs. Catalina makes the use of program executables much more explicit, because there are applications when Prolog and Velocity simply won't do. For this example, I've already written the `stateVioletParser`, and have included that in my CLASSPATH.

Descend into the generated `ma` directory, and run `CatCore.Gui`. Use violet to create a state diagram. This, after all, is the input to your MDE application. Create the following gorgeous file (called `bStateChart.state.violet`):

---

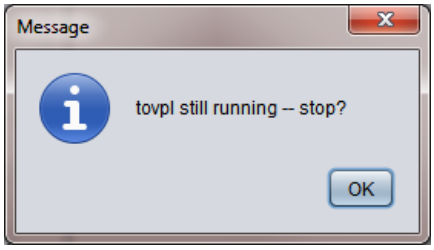
[8] Readers might wonder about the difference between "quoted-attributes" and non-quoted-attributes in schema declarations. Quoted-attributes have single-quoted values in Prolog, e.g., 'a', 'b'. Non-quoted attributes have unquoted values in Prolog, e.g. a, 1, z, 2. There are only 2 kinds of attributes in Catalina Prolog tables: quoted and non-quoted. The reasons for this distinction are that they are the two atom types in Prolog.



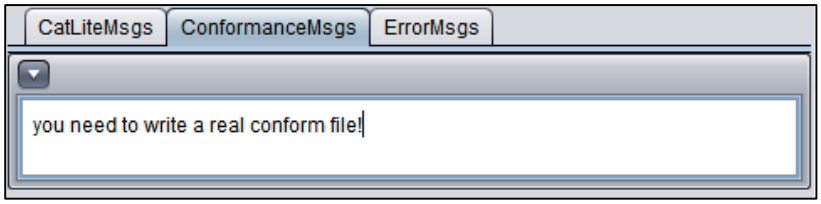
And enter it into the domain table and select the `tovpl` arrow. The domain table appears below:

Domain	FileValue
spl	
state	bStateChart.state.violet
code	

Note the coloring – rows that are input are in yellow; the row that is to be produced are in orange.[9] This is a nice, albeit flashy, feature of CatGui. Press any of the execution buttons (**Step**, **StepIn**, **Run**) and the arrow is executed. Doing so you'll get:



Something is wrong – either an error occurred in executing the `stateVioletParser` or a conformance error occurred, as something was wrong with the generated Prolog database. Clicking OK, you'll see it is a conformance error (in the **ConformanceMsgs** tab):

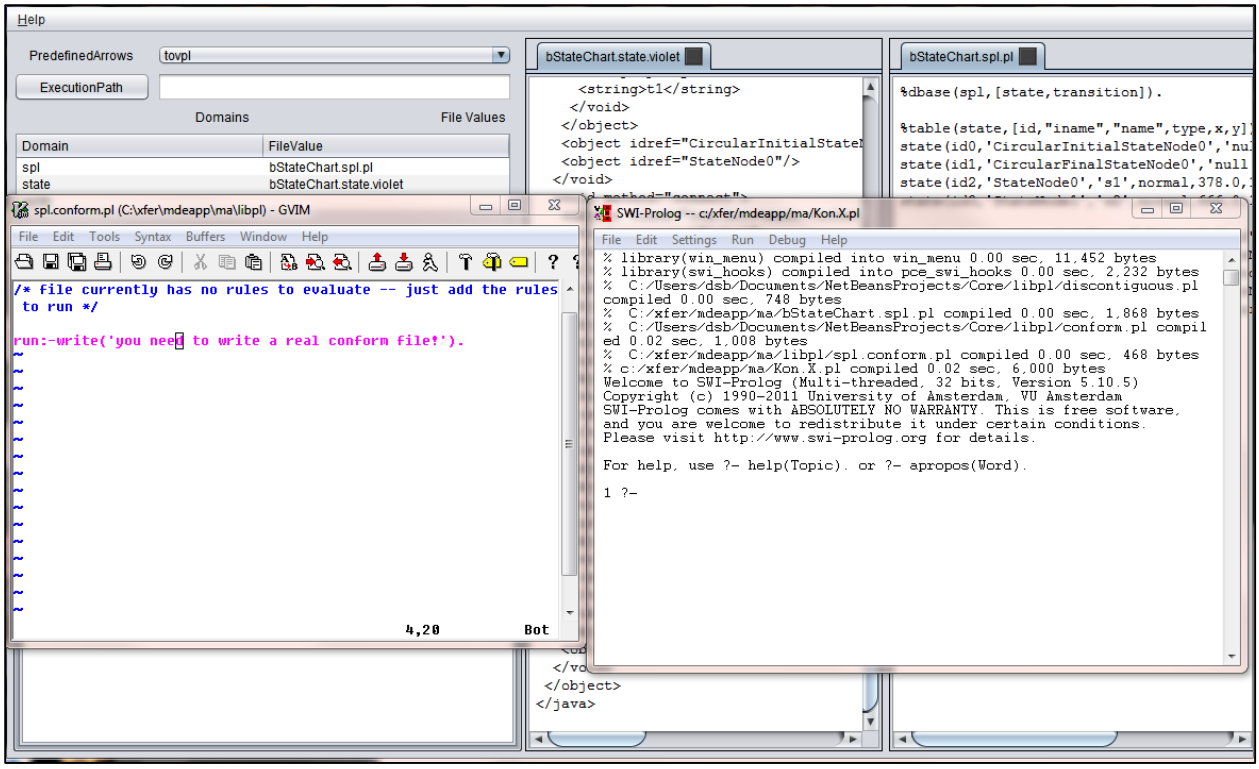


Click the **EditConform** button, which spawns two windows – a GVIM editor that allows you to write/update `spl` conformance rules and a swi-Prolog execution window (from which you can reload and run and debug your conformance rules).

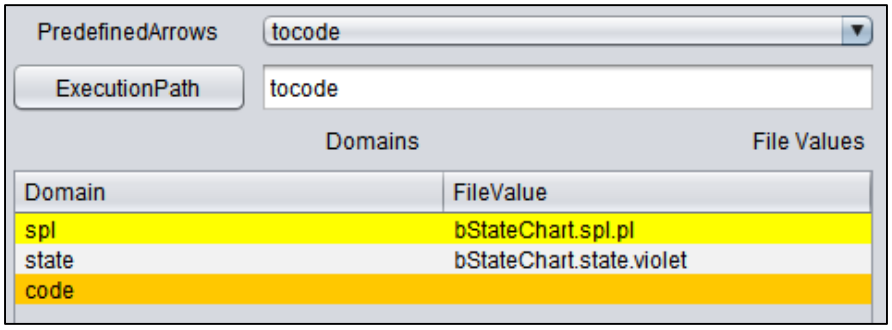
> **Hint**: edit the rules in your GVIM window, and save the result. Move over to the SWI-Prolog window and type the rule-name **`refresh`**. This rule reads in your file (along with others), compiles it, alerts you to errors, and otherwise allows you to run your ruleset interactively on the current database you are using to debug your rules.

When you're finished – which takes a wee bit of time, especially when you forget some basic commands (as I do) in Prolog – save the file and go onto the next step.

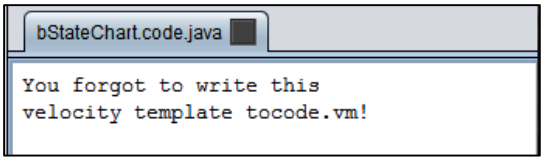
[9] I didn't mention this before, but this "feature" is present in Catalina.Gui too.



The next step is to convert the spl database into Java text. So dial in the `tocode` arrow:



press Run, and the output produced (shown in the output file tab) tells you everything:



So, again, you can interactively develop this file by clicking the **EditXform** button. This time, you are editing a Velocity file (via a GVIM window). As there is no interactive tool for Velocity, edit the file, save it, and press **RedoLast** (to rerun the Velocity execution) and see the results in the output file tab. This cycle continues until you are done.

At this point, your MDE application is complete. I can drag the `myapp` folder to the Category Repository to make it available for others to use. When they start `CatCore.Gui`, they simply select `myapp` as the

application to execute. Of course, giving myapp a better, more descriptive name would be the first order of business if I wanted to post this application to others.

6. Do I have to Use the Catalina Gui?

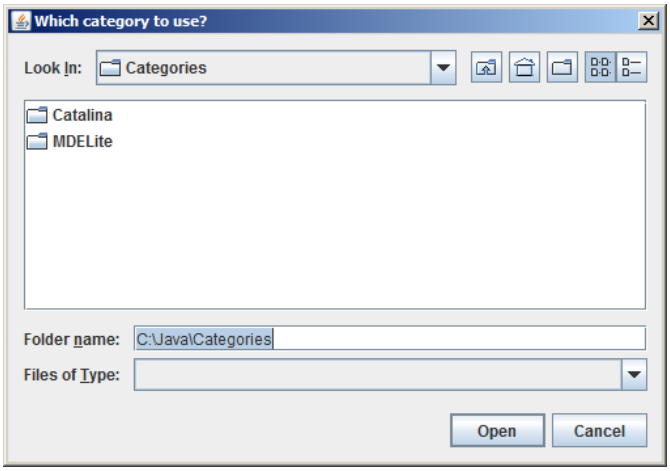
Not really. You just have to do more work manually. The Gui Advise hand-holds you through the creation of an MDE application. It makes sure, for example, that you have created schemas properly for every database. As the Catalina interpreter is prolog-table-driven, it ensures that you have created these tables correctly. You don't want to do this by hand. However, once a these tables are created, you can use the Gui or not to run your applications.

You can simply write MS Dos batch scripts or bash scripts, but it is a bit ugly. In fact, the ugliness and repetition was the motivation for the Gui development in the first place. Regardless always remember that every arrow is a stand-alone program in Catalina. So when you execute a Catalina MDE application, you are invoking a series of programs (arrows). You can execute an arrow by invoking `CatCore.Main`, once you have chosen its metamodel. (Think of `CatCore.Main` as constructor $\mathcal{K}(x)$ above. You have to specify argument x, which is its metamodel, in file config.properties, which is in the directory that you run `CatCore.Main`).

The simplest way to set the metamodel is via this invocation:

```
> cd test
> java CatCore.Main setMetaModel
```

By doing so, you will choose among the available (published) metamodels:



And in doing so, will create a `config.properties` file in the test directory above. Encoded in this properties file is your metamodel selection. You only have to do this once.

Now, suppose you want to invoke arrow $\alpha: A * B \rightarrow C$ using files $a \in A$ and $b \in B$ to produce file $c \in C$. Let $\eta(f)$ denote the filename of f. You can invoke α by the call:

```
> java CatCore.Main η(a) η(b) run η(c)
```

If all goes well, this execution is silent – no output will appear. If you do see output, then something is wrong. (The Catalina Gui will halt when such output occurs). There are three possible outcomes of an arrow execution – it worked, there were conformance errors, there were execution errors. Look in the following files:

- `conformance.txt` – contains the list of conformance errors. Your arrow produced a Prolog database, and associated with each Prolog database is a list of constraints (conformance rules) that were defined. Every rule that is violated produces an error describing the violation. All such violations appear in this file. If there are no errors, this file either does not exist or has length 0.
- `error.txt` – contains a list of internal errors that were produced during arrow execution. Typically such errors are reserved for something egregious – like parsing errors and the like. Something is really wrong that you need to fix.

So, always check these two files after every arrow execution.

Catalina category path specifications are of the form $p = \alpha.\beta.\gamma$. Assume $\alpha: A \to B, \beta: B \to C, \gamma: C \to D$ and the input to α is $a \in A$. Effectively, `CatCore.Gui` translates this into the following calls:

```
>> java CatCore.Main η(a) run α
>> (check for errors)
>> java CatCore.Main η(b) run β
>> (check for errors)
>> java CatCore.Main η(c) run γ
>> (check for errors)
```

Or equivalently:

```
> java java CatCore.Main η(a) run α.β.γ
```

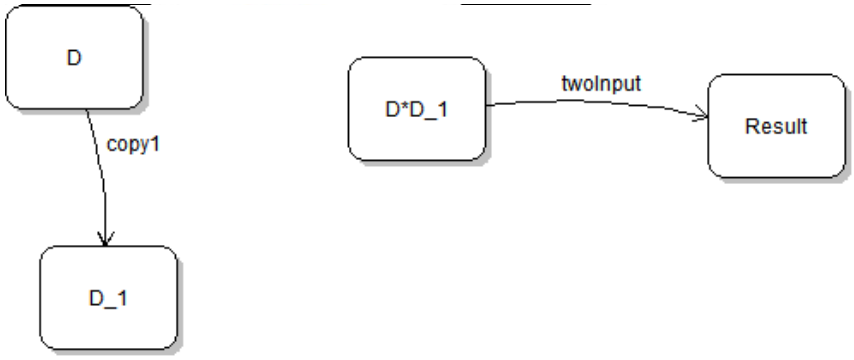
Where the (check for errors) is:

- halt if execution is not silent
- halt if `conformance.txt` has non-zero length
- halt if `error.txt` has non-zero length

7. Some Built-In Arrows I Haven't Told You About (Yet)

Catalina has several built-in arrows that you can use, or are automatically invoked.

- Whenever you produce a Prolog database, its Prolog rules are automatically checked. For all other domains, there is no implicit conformance checking. If you have a program that checks a non-Prolog domain, you have to encode it as an explicit arrow to execute. Typically I write this as an arrow $conform: D \to bool$.txt, where D is the domain whose file I want to check and `bool.txt` is a domain where a zero-length file means true and a non-zero-length file (containing the conformance errors) is false. The domain "`bool`" is built-into Catalina, and has this semantics.
- Rename – occasionally it is useful to rename files, just to distinguish them. Consider the category below:



A typical way to debug arrows is to evaluate the pseudo-identity transformation: $tovdb$ followed by $toViolet$ (i.e. the path $tovdb.toviolet$). I call them pseudo-identities in that the files that are input and produced are not syntactically identical, but are semantically equivalent. (Often, transformations lose irrelevant bits of information, and this is OK). So for all $a \in class.violet$, $a \equiv toviolet(tovdb(a))$. The problem with executing the path tovdb.toViolet is that Catalina overwrites the input file a. That is, arrow `tovdb` maps `a.class.violet` to `a.vdb.pl` and arrow `toViolet` maps `a.vdb.pl` to `a.class.violet`. You don't want to do that. So one way is to simply rename the file along the way. The "rename" arrow has the syntax:

```
domainName+Extra
```

where `domainName` is the name of a domain in the MDE category and Extra is a string (typically just a single character) that is appended to a file name. Here's the path that I would use in this situation:

```
tovdb.vdb+S.toViolet
```

where the file that is produced by the $toViolet$ arrow is $aS.class.violet$ given that the input file was named $a.class.violet$.

- Move – sometimes Rename is not sufficient. Here's an example: you have two databases $a.D.pl$ and $b.D.pl$ that you want 'merge'. The way the domain table works is that only one file is "current" in a domain at a time. Here you need 2 files in the same domain to be current. Here's a simple way to deal with this: create a "copy" domain.

D1 is a "copy" or "clone" of domain D. Simply invoke the "copy" arrow (which does nothing, really) except change file $a.D.pl$ to $a.D\_1.pl$, at which point, you can then evoke arrow $twoinput$. The path that you would use in this case is:

```
copy1.produceAnotherD.twoInput
```

copy1 takes the current file in domain D, $a.D.X$, and copies it to $a.D\_1.X$. The arrow $produceAnotherD$ produces another D input, say $b.D.X$, and arrow $twoInput$ would then take $b.D.X$ and $a.D\_1.X$ to produce $b.Result.Y$.

A slightly better way is to use the built-in move arrow which has syntax:

```
domainName>domainName_extra
```

where "_extra" is the name that distinguishes the domain "copy", as "_1" does in the above figure. The above path would be the same length, but one arrow is predefined:

```
D>D_1.produceAnotherD.twoInput
```

8. Other Items

1. Filth

The current version of Catalina is exceptionally dirty. It generates lots of files whose contents you will be clueless. (They are, in effect, present for debugging Catalina and not for your consumption). To get rid of them, run:

```
> java CatCore.Clean
```

The files that are removed include:

- **error.txt** – a Catalina arrow may have execution errors. If so, they are reported in this file. An empty-length **error.txt** file means (to the Catalina interpreter) that there were no execution errors
- **conformance.txt** – a Catalina arrow may invoke prolog constraints (typically only when a prolog database is created). An empty-length **conformance.txt** file means (to the Catalina interpreter) that there were no conformance errors. If there are conformance errors, they are reported in this file.
- **Exe.X.pl** – this is the prolog file that is executed in a M2M arrow execution.
- **Kon.X.pl** – this is the prolog file that is executed in a model conformance test
- **executedLine.bat** – this is a batch file that indicates the last program (arrow) that the Catalina interpreter executed.
- **script.txt** – this is an input script to SWIPL_EXE

The above files are produced during Catalina execution. There are additional files that are Catalina-MDE application specific. In a Catalina category specification, you'll be generating all sorts of intermediate results. Chances are, you don't want to see them once your application is trusted. Here's how to get rid of them. Edit the config.properties file of your MDE application. (This file was produced, along with libpl, libvm, etc). Add the line:

```
SAVE_LIST = pattern1   pattern2
```

What `CatCore.Clean` will do is NOT delete any file whose name includes the text of any pattern listed. If a file has a name that does NOT include any of the patterns listed, it will be deleted. (There are built-in patterns for .lite, libpl, libvm, etc, so that inherently "good" files are retained, so you don't have to specify them). An example is:

```
SAVE_LIST = .violet .yuml
```

All files whose names do not contain strings "`.violet`" and "`.yuml`" will be deleted. Use with caution.

2. Validation

Should you create your own MDE application, you'll want to add it to the Categories/ directory for others to use. To validate that all files referenced (e.g. vm2t template files, java executables, prolog files) can actually be referenced, you should cd into the metamodel directory where you will find a cat.lite.pl file. This is the prolog database that drives the Catalina interpreter. You really shouldn't change this file at all, but sometimes adding paths is OK without having to regenerate the entire MDE application. Anyways, to validate that all files referenced are present, you can run:

```
> cd mymetamodel
> java CatCore.Validate
```

This is purely optional – Validate was created by me just for a sanity check.

9. Final Thoughts

Catalina was initially developed a command-line tool to load the internal equivalent of the domain table and to execute arrows. It worked, but this was hardly better than MDELite.

I discovered that developing Catalina was an extraordinary challenge. Building an IDE prototype that presents an environment in which to develop, run, and debug MDE applications is *really* difficult. Having knowledge of programmatic access to system environments, forking processes, writing non-trivial GUIs, making sure all execution paths are set (and they can easily differ from one environment to another) requires monumental hacking. I see no way around it.

MDE additionally imposes its own special problems. Developing an MDE application spans from coding in the weeds (e.g., a state diagram parser that processes an xml file and produces a file of Prolog well-formed facts), to debugging arrows – arrows are functions that map complex files as input to complex file(s) as output – is itself more difficult than typical programming. And the leaps of abstraction, from low-level Java details to the high-level concepts of arrows, can be particularly intimidating. Sorting out all these ideas and keeping them straight (without confusing issues) is non-trivial. Using different environments for different kinds of programming significantly aid this process. One environment is needed for debugging Prolog, another for Velocity, yet another for Java: this supports separation of concerns. And understanding that categories are essentially high-level flow charts, and executing the arrows of a category (with guidance) is an advising program, ties a lot of ideas together.

Catalina is a minimalist attempt to make all of this happen. Comments are welcome.