The translator consists of two components:

The development environment is a set of plugins for the MagicDraw UML modeling tool. These plugins let you describe translation and extraction graphically in UML class diagrams: you can "compile" these diagrams into translation rules used by the runtime. It should be stated here that UML class diagrams were created to describe structures: they were not really meant to describe transformations, so we will be taking liberties and bending the notation. Depending on how you structure them, these rules can be either unidirectional or bidirectional. Once the rules have been produced, the development environment is no longer necessary. If you wish to develop your own rules, you need to have MagicDraw (the relatively inexpensive "Personal Edition" is satisfactory).

The runtime component uses the rules produced by the development environment to actually convert or extract data. Currently the input data and output data are in XML format, and XML schemas are required for these files. STEP files described by EXPRESS schemas should be available soon. The runtime requires the JESS rule engine. You will need to obtain a JESS license from Sandia National Labs: JESS is not open source, and does not come with the translator.

Both components are expected to run under Windows and Linux, but they have been tested only Macintosh OS X 10.4.

To install the development environment, you need to:

- Create a "Prometheus" directory in your home directory
- Unzip the development environment and, put the contents in the MagicDraw Plugins directory.
- Restart MagicDraw.
- You should now see some extra choices in the context menus for the MagicDraw Containment hierarchy browser. For example, Packages will have the menu item "Generate mappings (both, file)". This means "Generate two rules files (one for left-to-right translation, and one for right-to-left translation) containing one rule for each diagram in the package." The rules files are placed in the ~/Prometheus directory. Additional choices are also added to the context menu for diagrams in the containment browser.

To install the runtime, you need to:

- Install Ruby (used for Schema processing: this requirement will be removed in future versions).
- Install jRuby (used for the runtime proper).
- Put the JESS jars somewhere convenient.
- Create an environmental variable that creates a classpath fragment containing the paths to all the JESS jars. I did it thusly under OS X (DSL identifies a directory of development libraries):

```
export JESS_HOME=${DSL}/Jess70p1
export JESS_CLASSPATH=`echo ${JESS_HOME}/lib/*.jar | tr ' ' :`
```

- You can then cd into the translator directory, and run the translator regression tests the command line (this should all be on a single line:

```
CLASSPATH=$JESS_CLASSPATH  jruby -J-Xmx1024m -J-Xms256m  /usr/local/bin/jspec
./specs --color
```

- You can look at some of the high level regression tests in specs/spec_translator.rb to see how to call the translator (it's easy).

There are several steps to producing rules. The first is to produce the necessary UML model and rule templates. The rule templates are rather like schemas/models/ontologies for rules. Currently part of this functionality is in the runtime, not the development environment, and part is still under development. The first step is to modify Rakefile.rb in the runtime directory. Modify one of the tasks to contain the file path to the schema you wish to use. The cd into the runtime directory and run

```
rake task_name  # Use the name of whichever task you modified.
```

If you started out with a file named someSchema.xsd, this will create in the runtime temp directory:

- A someSchema.clp file containing the templates.

- A someSchema_twiggy.rb file containing a "Twiggy" model of the data.

- A someSchema.rb file containing a Ruby of the data (not used by the translator).

The next step is to import the "Twiggy" model into MagicDraw. This is the part of the tool chain is not part of this delivery and is not yet available. If your model is small, you will be able to inspect the Twiggy file and manually reproduce it as a UML model. It may help to compare the example Twiggy files (for GenCam and Offspring) with their corresponding example UML models.

The above two steps need to be done once for the "input" schema and once for the "output" schema: at that point you should have two UML class models. To describe the translation mappings, you put both models in the same class diagram, and draw dependencies that show how data moves during the translation. I like to draw all of these dependencies in a single diagram, because it shows me the entire forest, not just a few trees. Then copy fragments of this diagram into smaller diagrams. How small do these diagrams have to be? That will be answered in a moment. Each of these smaller diagrams will correspond to one rule. A rule is similar to a production in BNF. It tells you how to transform one part of the class model graph. Each rule has a "pattern" (which specifies what will be acted upon) and a "consequence", which specifies what will be done to the data that corresponded to the pattern. So to fully specify a rule, you need to indicate in the diagram which classes are part of the pattern, and which are part of the consequence. We can not use UML stereotypes for this purpose because a given class may be part of the consequence for one rule (diagram), and part of the pattern for a different rule (diagram). So instead we specify this by color coding the classes. The color codes (and the menu picks to generate them) are listed in the Vertex Color Codes table. Diagrams are presumed to describe both left-to-right and right-to-left transformations unless you apply to the diagram the <<LTR_Only>> or <<RTL_Only>> stereotype. These stereotypes affect only bulk rule generation (for all the diagrams in a

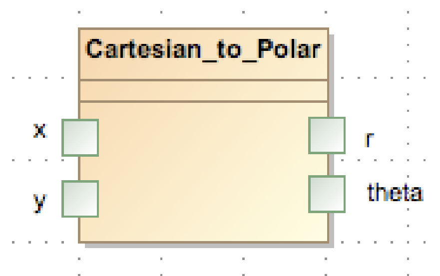
package): if you want to try to generate a right-to-left rule for a specific diagram that does not support it, this tool will not prevent you.

Sometimes just moving data is not enough, you need to manipulate the data somehow. An example is the conversion between Cartesian coordinates and polar coordinates:

Cartesian to Polar	Polar to Cartesian
r = sqrt(x^2 + y^2)	x = r * cos(theta)
theta = arctan(y/x) if x>0 and y>=0 theta = 2*PI + arctan(y/x) if x>0 and y<0 theta = PI/2 if x = 0 and y>0 theta = 0 if x=0 and y=0 theta = PI+PI/2 if x=0 and y<0 theta = PI + arctan(y/x) if x<0	y = r * sin(theta)

Somehow we need to specify in the diagram that these functions need to be called during the execution of the corresponding rule. To do this, create a Class in the diagram, and give it the <<Xform>> stereotype. For every input to the function, create a port on the class (with the correct datadype). Likewise create a port for every output. Then use dependencies to hook up those ports to the left and right sides. This functionality is provided for functions that take multiple arguments and/or produce multiple results. You do not need to use this mechanism to tell the rule compiler that it needs to convert one primitive type to another: this is deduced automatically. Later on we will explain how to provide the code that implements these functions (both the primitive translations and the Xform).

The appearance in the diagram of a Cartesian_to_Polar transformation. In this case, x and y are inputs for left-to-right transformations, and outputs for right-to-left transformations. In addition to placing this in the class diagram, you will need to write two functions, Cartesain_to_Polar_ltr and Cartesain_to_Polar_rtl.



It's also useful to be able to indicate globals in the diagram. To do this, create a Class and give it the Global stereotype. This class actually acts a s holder or namespace for multiple globals. For each global, add a port to the class, and connect it with dependencies to all the places the globals are used.

The last type of vertex you can create in a rule-defining UML class diagram represents collections. Normally collections in UML diagrams are implied by multiplicity on associations. The rule builder knows this, and handles those collections the way you would expect, but sometimes you need to explic-

itly show collections (for example, when you need to append a newly created object to a collection). You can do this by creating an instance of MappingTools::Collection in the diagram. But how do you show that some class instance is holding that collection, without converting your entire class diagram to an instance diagram? Just use a dependency from the class of the holder to the Collection instance, set it's stereotype to <<Link>>, and give the Collection instance a name that matches the association. We don't use associations here, because an association with that role (pointing to the contents of the collection) is already in use. These Collection instances participate in patterns and may be deleted or destroyed in consequences very much like classes do, so you need to be color coded just like the classes. You do not need to color code the Xforms or the Global though: their participation in the rules can be deduced from the incoming and outgoing dependencies.

Lastly, there are several types of dependencies. The mapping dependencies that have already been discussed come in two flavors: unidirectional and bidirectional. Bidirectional dependencies function as shown when you translate from left to right. When you translate from right to left, however, the direction is reversed, so that the head of the arrow is treated as though it were really a tail, and the tail is treatede as though it were really a tail. Unidirectional mappings on the other hand do not reverse their direction. Another important type of dependency (not previously discussed) can be used to indicate associations implied by matching attribute values (such as XML Schema key/keyRef constraints). When one of these attributes is inherited, you can create a port to represent the inherited attribute on the class of interest. Lastly, you can use dependencies to indicate operations. Currently the only supported operation is <<Append>>. The dependency tail must be a Collection instance, and the head must be a class: when the rule fires, the appropriate instance of that class (either newly created, or identified in the pattern) will be added to the Collection. To distinguish between these different cases, it's best to select the dependency and then choose the semantics from the context menu from the containment browser's icon representing the containing diagram.

Vertex Color Codes

Appearance	Semantics	Context Menu Item
Yellow fill	The pattern of left-to-right transformations will require an instance of the class	Vertex condition ltr
Pink fill	The pattern of right-to-left transformations will require an instance of the class	Vertex condition rtl
Blue fill	Both transformations will require an instance of this class in their patters.	Vertex condition both
Red boundary	Either deleted (if part of the pattern), or created (if not part of the pattern) in the consequence.	Vertex create/delete
Black boundary	Not deleted or created in the consequence	Vertex keep

Vertex Color Code Combinations

Fill Color	Border	Left-To-Right	Right-To-Left
Yellow	Black	Part of pattern, not deleted in the consequence	Ignored
Yellow	Red	Part of pattern, deleted in the consequence	Created in the consequence
Pink	Black	Ignored	Part of pattern, not deleted in the consequence
Pink	Red	Created in the consequence	Part of pattern, deleted in the consequence
Blue	Black	Part of pattern, not deleted in the consequence	
Blue	Red	Do not use this combination	

Edge types

Appearance	Element	Semantics	Menu
Green, wide arrow head	Dependency, <<Unidirectional>>	Copying of data	Edge unidirectional
Blue, wide arrow head	Dependency, <<Bidirectional>>	Copying of data	Edge bidirectional
Black, narrow arrow head	Dependency, <<Link>>	In Collection (implied by an association with multiplicity > 1)	Edge temporary
Black, narrow arrow head	Dependency, <<Append>>	Operation: add item at arrow head to the collection at the tail.	Edge append
Normal (black, wide arrow head)	Dependency, No stereotype	Implied association implemented by matching values.	
Normal	Association, max multiplicity = 1	Instance of class at tail holds instance of class at head	

Appearance	Element	Semantics	Menu
Normal	Association, max multiplicity > 1	Instance of class at tail holds collection containing instance of class at head.	

Issues:

- Enhancement - Additional constraints. It would be nice to be able to specify further constraints on instances that are operated upon by a given rule: for example, make the rule applicable only to instances that have a given set of attribute values. This is quite easy to express in the JESS language... but the representation in UML is not so straightforward. It actually ought to be easy in UML, using instances instead of classes... but there are subtle reasons why this does not work very well.

- Enhancement - Exception handling. Several kinds of exceptions can occur during translation. For example it could be that a Xform can not compute a result without human disambiguation. It could be that an Xform encounters input data that simply can not be represented in the output format, even with human intervention.

Issues present in initial delivery (now resolved):

- Bug (fixed) - JESS facts need to be unique. That means you can't create a fact representing an empty collection or object and expect to fill it in later. An ID can be added to make the instances unique. The existing code is adding those IDs in most places... but not all of them. Without the ID, if you tell JESS to create a new fact and it fails to do so because there already exists another fact with the same data, it will return false. This messes up subsequent operation of the translator.

- Bug (fixed) - Nil handling. Currently JESS fails if an Xform returns a nil value. JESS can (and does) handle nil in other contexts, so this should not be too hard to fix.

- Enhancement (done) - Ease of use. Currently a translation specification file is reversible (if the rules are reversible)... except for the input and output tweaking procedures. Currently you can specify this for one direction in the specification file, and override that when you invoke the translation, but it really it would be better to make this 100% reversible.

Lessons learned

- Both standards leave a lot to be desired. I encountered more ambiguities than I expected.

- Rule engines lack some features that might be desirable:

  Wildcard for slot name (find all the facts that hold a Color in *any* slot of template Drawing)

  Wildcard for template name (find all the facts (regardless of template) held in slot "background".

Change of template (without retracting and re-asserting) would avoid need to identify and modify all the holders of that fact.

- Even though graphical construction of rules simplifies things a lot, it’s still difficult.
- The MagicDraw plugin API is cumbersome. For example, it has 14 interfaces named “Classifier” (in different packages), and 7 classes named “ClassifierClass”. It also has a lot of “helper” classes that provide behaviors that really should exist on some other object.