AGE Tutorial
Version 0.3.1

Jesús Sánchez Cuadrado

May 26, 2007

Contents

1 Introduction **3**
1.1 Installation 3

2 Class-to-Table. Step by step. **3**
2.1 Creating a project 3
2.2 Creating metamodels 4
2.3 Creating source models 6
2.3.1 Creating models with Ruby 6
2.3.2 Creating XMI models 7
2.4 Model-to-model transformations 7
2.5 Model-to-code transformations 8
2.6 Launching transformation tasks 10

1 Introduction

This document is UNDER CONSTRUCTION.

1.1 Installation

Download the tool from **http://gts.inf.um.es/age** and unzip the downloaded file in any directory. Click on the age(.exe) to run it. There are only two requisites, the Java and Ruby runtimes must be installed.

To install the Ruby interpreter download it from:

- http://www.ruby-lang.org/
- http://rubyforge.org/frs/?group_id=167 (for windows)

Once installed the interpreter, AGE should be configured with the path where the Ruby interpreter has been installed. Click on Windows → Preferences and select Ruby → Installed Interpreters preferences. In this dialog, you can add the installed interpreter, usually the path `c:/ruby/bin/ruby.exe` or `/usr/bin/ruby` is used.

2 Class-to-Table. Step by step.

In this section we will detail the basic steps needed to perform a complete model transformation, from a class model to a set of files (java files and sql files), going through an intermediate relational model.

The steps we will follow are:

1. A new project to hold the transformation is created.
2. Class and relational metamodels are created using the metamodel editor provided with the tool.
3. A sample source model is created using a Ruby textual syntax.
4. A model-to-model transformation is written to transform the class source model in a relational target model.
5. Model-to-code transformations are written relying on Erb (a Ruby templating system) to define how to generate files.

The example can be download from **http://gts.inf.um.es/age**. Import it with File → Existing Projects into Workspace.

2.1 Creating a project

The first time the tool is run the *Resource perspective* is shown. To change to the *AGE perspective* click on Window → Open Perspective → Other... and select AGE in the dialog.

The *AGE perspective* provides additional menus and shortcuts to the Eclipse default ones. Figure 1 shows a snapshot of the perspective and how to access easily to AGE shortcuts in order to create projects and files. This shortcuts will be used in the rest of the tutorial.

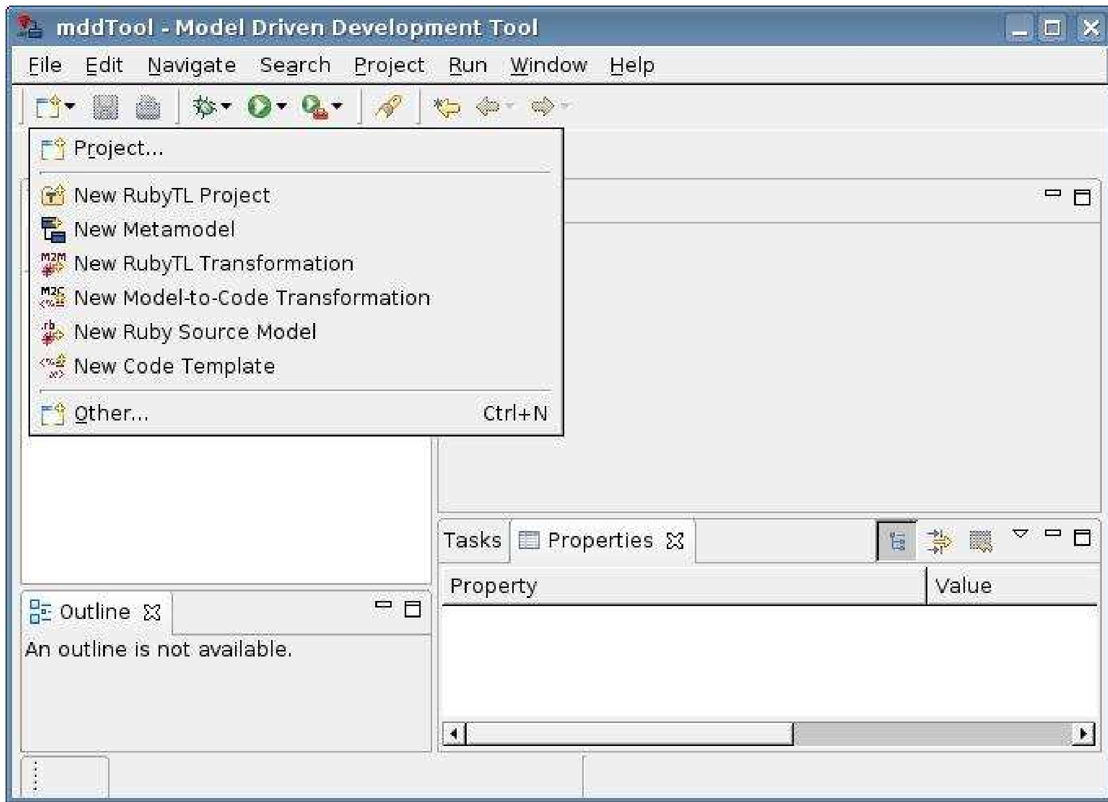

Figure 1: AGE perspective.

To create a new project, click on File → New ... → New RubyTL Project or use a shortcut. In the dialog, name the project as `class2table-tutorial`. A new project will be created, and it will have four folders[1] initally:

- metamodels. Metamodel files (.ecore files usually created with Eclipse metamodel editor).
- models. Model files, either Ruby model files (see subsection **??**) or xmi files.
- templates. Code templates used in model-to-code transformations.
- transformations. Either model-to-model or model-to-code transformations.

2.2 Creating metamodels

The first step before writing a transformation is creating the source and target metamodels. The metamodel editor provided in previous versions of the tool has been removed. You can use the Eclipse tree editor to create metamodels.

[1]In a future version we aim to rely on a model repository rather than on folders and files

Right-click on the metamodel folder and select New metamodel. In the dialog write the file name of the metamodel, in this case `ClassM.ecore` (be sure it ends with `.ecore`)[2]. Once the metamodel file has been created, double-click on it to open the editor.

Figure 2 shows the metamodel editor (including the properties view) and the final Class metamodel. The detailed steps to create this metamodel are the following:

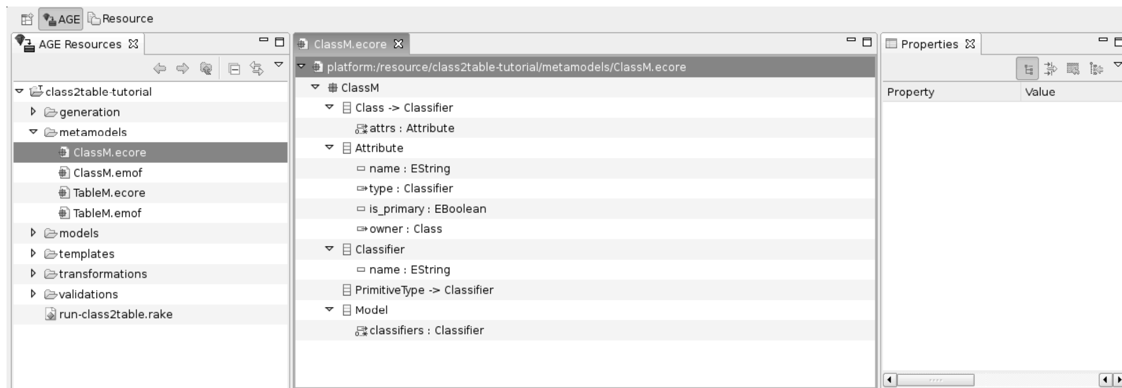

Figure 2: Metamodel editor editing the Class metamodel

1. For the main package, write the package name (`ClassM`) using the properties view. It is a recommended practice to write an URI and a prefix for the XMI file.

2. Create a new metaclass clicking with the right button on the package element, and selecting New Child → EClass. Name the class `Classifier` and set the `abstract` property to true.

3. Add an attribute to `Classifier` class. In the same way as before, right-click on the `Classifier` metaclass and select New Child → EAttribute. Now, change the attribute properties in the properties view (use `name` as name, and `EString` as type).

4. Add three more classes like in the first step. Name this classes as: `Class`, `PrimitiveType`, `Attribute`.

5. Make `Class` and `PrimitiveType` a generalization of `Classifier`. To do this use the ESuperTypes properties (click on ...), and select the `Classifier` metaclass in the dialog.

6. Add attributes to the `Attribute` class, such as `name : EString` and `is_primary : EBoolean`.

7. Establish a reference between `Class` and `Attribute`. Right-click on the `Class` metaclass and select New Child → EReference. Fill the properties with `name = attrs`, `upperBound = -1` (because is a many-to-one relationship), and `containment = true`.

8. Establish the opposite reference between `Class` and `Attribute`. Right-click on the `Attribute` metaclass and select New Child → EReference. Fill the properties with `name = owner`, `upperBound = 1`. Set the eOpposite property with `attrs : EAttribute`, that is, the previously defined reference.

[2] You can create EMOF metamodels, but we recommend to use ECore metamodels. If you have EMOF metamodels, you can convert them to ECore, just right-clicking on the file and selecting `AGE → Convert From EMOF to Ecore`

9. Complete the rest of the metamodel in the same way.

The relational metamodel is shown in Figure 3. Create a new file named `TableM.ecore` and follow the same steps as before to create the metamodel.

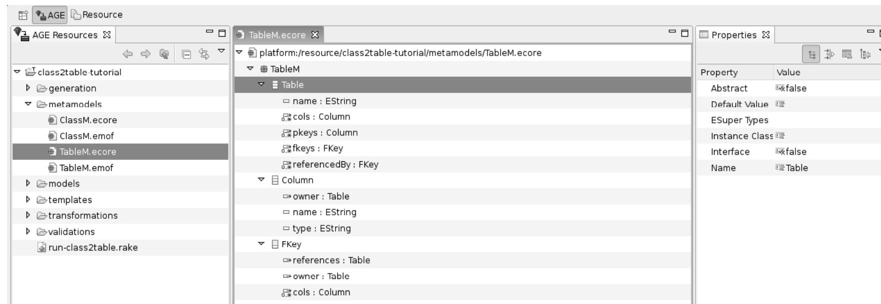

Figure 3: The relational metamodel

In future versions of the tool we plan to provide an embedded DSL to define metamodels.

2.3 Creating source models

Creating source models for any arbitrary underlying metamodel is a rather complicated issue. At present, there not exist any stable editor generator which is able to get a metamodel and produce the corresponding graphical editor. The GMF[3] aims to provide an infrastructure for developing, almost automatically, graphical editors based on EMF and GEF, but it is not stable yet.

2.3.1 Creating models with Ruby

We use a custom strategy based on Ruby to create source models (anyway it is possible to use XMI models, see next section). The key point is creating the objects of the models as if they were normal Ruby objects. You can create the source model below by right-clicking on the models folder and selecting New Ruby Source Model. Name this file as `class-source.rb`.

```
#! ClassM => metamodels/ClassM.ecore

dt_string = ClassM::PrimitiveType.new(:name => 'String')
dt_integer = ClassM::PrimitiveType.new(:name => 'Integer')
dt_boolean = ClassM::PrimitiveType.new(:name => 'Boolean')

job = ClassM::Class.new(:name => 'Job') do |klass|
  klass.attrs << ClassM::Attribute.new(:name => 'name', :type => dt_string, :is_primary => true)
  klass.attrs << ClassM::Attribute.new(:name => 'address', :type => dt_string, :is_primary => true)
end

pet = ClassM::Class.new(:name => 'Pet') do |klass|
  klass.attrs << ClassM::Attribute.new(:name => 'name', :type => dt_string, :is_primary => true)
  klass.attrs << ClassM::Attribute.new(:name => 'age', :type => dt_integer)
end
```

[3]http://www.eclipse.org/gmf/

```
person = ClassM::Class.new(:name => 'Person') do |klass|
  klass.attrs << ClassM::Attribute.new(:name => 'pet', :type => pet)
  klass.attrs << ClassM::Attribute.new(:name => 'job', :type => job)
  klass.attrs << ClassM::Attribute.new(:name => 'name', :type => dt_string, :is_primary => true)
  klass.attrs << ClassM::Attribute.new(:name => 'age', :type => dt_integer)
end

job.attrs << ClassM::Attribute.new(:name => 'best_employee', :type => person)
```

The Ruby idiom to have named parameters in the class constructor is used, that is, a hash with symbols (keywords starting with `:`) as keys, being these keys attribute names of the class. In Ruby, the key-value association in a hash is specified by the `=>` operator. Also, the `<<` operator is used to add elements to association ends with cardinality greater than one.

Finally, it is important to know that the model will be actually loaded before the transformation is executed.

2.3.2 Creating XMI models

There are two ways to create an XMI model. The first one is to define the model using Ruby code. Then, right-click on the file and select `AGE` → `Convert Ruby model to XMI` to generate the corresponding XMI file. It is important to write this instruction in the first line of the file (as shown above): `#! ClassM` => `metamodels/ClassM.ecore`, means that the metamodel for the model you are defining is `ClassM.ecore` and you are using `ClassM` as namespace for the metamodel classes.

The second way to define the model is to use the Eclipse tree-editor. Open the metamodel, select the root element, right-click and select `Create dynamic instance` on the pop-up menu.

2.4 Model-to-model transformations

Model-to-model transformations are performed by RubyTL. In this section we will not provide details about RubyTL, but in Section **??** a complete explanation of RubyTL is given.

To create a new transformation file, right-click on the transformations folder and select New RubyTL transformation. Name the file as `class2table.rb`, and copy the transformation below to the just created file.

```
transformation 'class2table'
input 'ClassM' => 'http://gts.inf.um.es/examples/class'
output 'TableM' => 'http://gts.inf.um.es/examples/relational'

top_rule 'klass2table' do
  from ClassM::Class
  to    TableM::Table
  mapping do |klass, table|
    table.name = klass.name
    table.cols = klass.attrs
  end
end

rule 'property2column' do
  from    ClassM::Attribute
  to      TableM::Column
  filter do |attr|
```

```
    attr.type.kind_of? ClassM::PrimitiveType
  end
  mapping do |attr, column|
    column.name = attr.name
    column.type = attr.type.name
    column.owner.pkeys << column if attr.is_primary
  end
end

rule 'reference2column' do
  from    ClassM::Attribute
  to      Set(TableM::Column)
  filter do |attr|
    attr.type.kind_of? ClassM::Class
  end
  mapping do |attr, set|
    set.values = attr.type.attrs.select { |a| a.is_primary }.map do |primary_attr|
      TableM::Column.new(:name => attr.type.name + "_" + primary_attr.name + '_id',
                         :type => primary_attr.type.name)
    end
  end
end

top_rule 'tofkey' do
  from  ClassM::Attribute
  to    TableM::FKey
  filter do |attr|
    attr.type.kind_of? ClassM::Class
  end
  mapping do |attr, fkey|
    fkey.cols = reference2column(attr)
    fkey.references = attr.type
    fkey.owner = attr.owner
  end
end
```

Please, refer to subsection 2.6 to know how to launch this transformation now. If you execute this transformation with the previous source model you should get a target model with 16 elements.

2.5 Model-to-code transformations

Model-to-code transformations are based on templates. In addition, special configuration files called `2code` files allows the mapping between templates and file names to be specified. This kind of files allows us to iterate over a model and select the model elements to be transformed to code. We will detail how to use templates and 2code files below.

2code files

Right-click on the transformations menu and select New Model-to-Code Transformation. Name the file as `table.2code`. Copy the following code into the file. Such code is intended to create an sql-file containing SQL "create table sentences".

```
main do
```

```
  compose_file 'tables.sql' do |file|
    TableM::Table.all_objects do |table|
      apply_template 'templates/create_table.rtemplate', :table => table
    end

    TableM::FKey.all_objects do |fkey|
      apply_template 'templates/create_fkeys.rtemplate', :fkey => fkey
    end
  end
end
```

As can be seen, Ruby code is used to specify the mapping between template, output files and model elements. There are four key points:

- The `main` keyword specify the entry point of the transformation, and must always exist.
- The `compose_file` method allows a file (`tables.sql` in this case) to be composed by the result of several template applications.
- `TableM::Table.all_objects` allows you to traverse all objects of the `Table` metaclass. At this moment this is the only structure provided by this DSL to traverse models, but we are looking into ways to improve that.
- The `apply_template` keyword is intended to specify mappings. Note that it has two parameters:
  - First, the name of the template to be applied. The result of the template application will be stored in file we have defined.
  - Second, variable mappings are specified using symbols (e.g. `:table`) and variables. This mappings gives a template the context to generate code, for instance, the `create_table` template will have access to a variable named `table`.

Another way is to create one sql file for each table. To do so, you can use the following code.

```
main do
  TableM::Table.all_objects do |table|
    template_to_file 'templates/create_table.rtemplate' => "#{table.name}.sql", :table => table
  end

  compose_file 'alter_tables.sql' do |file|
    TableM::FKey.all_objects do |fkey|
      apply_template 'templates/create_fkeys.rtemplate', :fkey => fkey
    end
  end
end
```

This language to generate code has other features. Download `class2java` example to see another style of generating code without using templates.

Template files

The next step is creating templates to generate the SQL code. To create templates right-click on the templates folder and select New Code Template. Create a file named `create_tables.rtemplate` and another file named `create_fkeys.rtemplate`.

`create_tables.rtemplate` takes a `table` object and outputs a "CREATE TABLE" statement. It should look like this

```
-- automatic generation of: <%= table.name %>
create table <%= table.name %> (
<% table.cols.each do |c| -%>
  <%= c.name %> <%= c.type.upcase %> NOT NULL,
<% end -%>

  constraint primary key (<%= table.pkeys.map { |c| c.name }.join(',') %>)
);
```

It is worth noting that a `table` can be accessed in the template. This is possible because we have bind such variable to a *table object* in the previous 2code file.

`create_fkeys.rtemplate` generates an ALTER TABLE statement to specify a foreign key constraint, and it should look like this.

```
ALTER TABLE ADD CONSTRAINT FOREIGN KEY (<%= fkey.cols.map { |c| c.name }.join(', ')%>)
```

The following issues are important when writing templates:

- `<%= %>` is used to execute Ruby code and output the result as a string.
- `<% %>` is used to write parts of Ruby code that must not be part of the output.
- Use `<% -%>` (with `-`) to omit newlines.

2.6 Launching transformation tasks

To launch transformation taks we rely on Rake[4]. Rake allows us to define task and dependencies between them. We have defined custom tasks to perform both model-to-model and model-to-code transformations.

Click on File → New → Other ... and select File in the menu. Name the file as `run-class2table.rake`. Copy the following code into the file. It defines a transformation chain from class model to code, going through an intermediate relational model by means of the `class2table` model-to-model tranformation.

```
model_to_model :class2table do |t|
  t.sources :package => 'ClassM',
            :model     => 'models/class-source.rb' #, 'models/class-source.ecore.xmi',
            :metamodel => 'metamodels/ClassM.ecore'

  t.targets :package => 'TableM',
            :model     => 'models/relational.ecore.xmi',
            :metamodel => 'metamodels/TableM.ecore'

  t.transformation 'transformations/class2table.rb'

  t.plugins :default, :set_type, :top_rules, :explicit_calls, :ignore_conflicts
end

model_to_code :tosql => :class2table do |t|
  t.codebase = 'generation'
  t.generate 'transformations/tables.2code'
end
```

[4]Rake is a build tool, written in Ruby, using Ruby as a build language. Rake is similar to *GNU Make* in scope and purpose. More information in http://docs.rubyrake.org/

The `model_to_model` and `model_to_code` methods specify tasks and its dependencies. The `tosql` task depends on the `class2table` task since the `class2table` target model is used as the source model for the `tosql` code generation (notice that such relationship is automatically resolved). The `codebase` attribute in the `tosql` task allows us to give a root path where generated files will be stored (automatic creation of directories is performed if they do not exist).

Notice that the model-to-model task can use an XMI file as the input model. To do that, just remove the comment (which starts with #), and use `models/class-source.ecore.xmi` as the input model. At this moment, the only limitation to use Ecore models is that the file extension needs to be `.ecore` or `.ecore.xmi`.

There are two ways of launching a transformation task:

- An Eclipse-launcher can be configured manually doing the following: click on Run → Run.... In the dialog enter the data as in Figure 4. Click on Run to launch the task.
- The second way is selecting the transformation task name with the mouse, and doing right click to select the menu option Run as → Run rakefile. This creates a new launcher automatically. This is the easiest way to launch a transformation task.

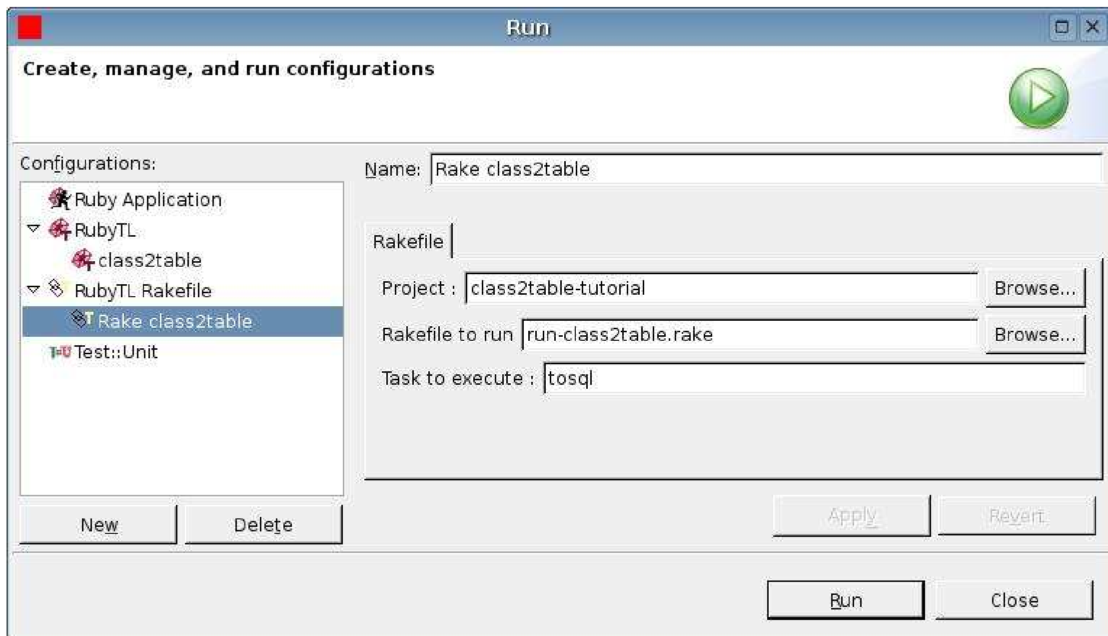

Figure 4: Run RubyTL transformation chain dialog