Velocity M2T Tool Tutorial

Model to Text Transformation

Robert Berg (robert.berg@utexas.edu), Eric Huneke (QualityCan@gmail.com),
Amin Shali (amshali@cs.utexas.edu), Joyce Ho (joyceho@utexas.edu)

Date: **May 11, 2012**

Updated August, 2014

Problem Introduction

Model Driven Engineering **(MDE)** is a methodology used in software design to simplify the goals, ideas, and design of domain-specific applications. Models are developed to give all parties (including engineers and scientists) working in a domain easier conceptual access to applications. For instance, the ***Unified Modeling Language*** **(UML)** represents the set of classes and relationships in an object oriented-system. While the source code underlying a UML model can be complex, the visual representation of the model provides a simpler view to anyone working in the domain who isn't familiar with coding practices. Thus, the benefit of MDE is clear. MDE becomes an even more powerful tool if the model in question can be translated into different representations. While this idea is simple to understand conceptually, it is not trivial to implement. In transforming models, three problems arise:

1. The source model must be transformed into the target model. Such an example might be the transformation between a truth table (the source model) and a binary decision tree (the target model).
2. The transformation must be validated to ensure the target model is correct within the new application domain while still encompassing the rules and restrictions that applied in the source domain.
3. For a complete transformation, we want to translate the target model into source code automatically, such as a Java classes containing all the information encoded in the target model.

The focus of this paper is on the third task of ***Model to Text*** **(M2T)** generation. In the sections that follow, a description of M2T transformations is given, including its uses and importance. Finally, a M2T tool, called ***Velocity Model-to-Text (VM2T),*** based on Apache Velocity is presented and used in two M2T transformation examples. These examples transformation are presented in a tutorial manner, detailing how to use the tool while also illuminating the concepts and process of model-to-text transformations.

The body of this manual is from the original text by Berg, et al. New additions to VM2T are discussed in the text (having **red titles**). Make sure that you read and understand these additions before proceeding to use VM2G.

Model-to-Text Transformations

Overview

M2T transformations are concerned with converting a given model structure (UML, XML, XMI, etc.) into some text-based representation. The output text can be anything from Java source code files (.java), to MySQL scripts (.sql), to XML structures (.xml); the options are limitless. A tractable example of M2T transformation is that of a UML class diagram transformed to Java source code (note that performing this transformation is detailed in the *Tutorial* section below). To illustrate, assume we want to transform the UML class diagram of Figure M1 to its associated java classes.

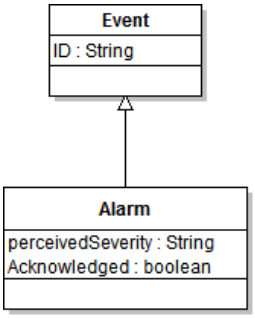


Figure M1. Example UML class diagram of which we would like to transform to java source code

An M2T transformation on the UML class diagram in Figure M1 should produce two Java class: **`Event.java`** and **`Alarm.java`**. In each case, the output source code for each class should include all information provided in the UML class diagram. For instance, `Alarm.java` should contain the data fields `perceivedSeverity` and `acknowledged` in addition to specifying that `Alarm.java` extends `Event.java`. As UML diagrams don't contain implementation details of class or instance methods, those details are left to the engineer utilizing the source code. The desired output for `Event.java` is shown below in Figure M2.

```java
public class Event {
  // Define the class data fields
  public String ID;

  // Class constructor
  public Event (String ID) {
    this.ID = ID;
  }
}
```

```java
public class Alarm extends Event {
  // Define the class data fields
  public boolean acknowledged;
  public String perceivedSeverity;

  // Class constructor
  public Alarm (String ID, boolean acknowledged, String
perceivedSeverity) {
    super(ID);
    this.acknowledged = acknowledged;
    this.perceivedSeverity = perceivedSeverity;
  }
}
```

Figure M2. The output java code of Alarm .java and Event.java after performing the M2T transformation of the UML class diagram of Figure M1.

Motivation and Value of M2T Transformations

Automation is generally superior to manual construction; M2T is no exception. In M2T, we want to produce source code from a model input with minimal effort (i.e. maximum automation). The key idea is that a relatively small piece of transformation code will generate much larger pieces of output code that is syntactically correct. This process provides several benefits. First, it allows for the rapid creation of redundant or boiler-plate code (e.g. hundreds of class templates from a large UML model). Second, given that the transformation code is correct, the automation process in M2T minimizes bug propagation because output source code is generated automatically rather than by hand. Finally, the M2T process is versatile in its output; if something needs to be changed in the generated output, such as a data structure or design pattern, the engineer need only manipulate the transformation code. In summation, it is believed that M2T decreases development costs while also providing the engineer with the means to easily maintain his project's source code base.

M2T Challenges

For robust transformations, the M2T tool must be flexible and able to handle several types of model inputs. Similarly, the tool must be able to output in an arbitrary target language. Finally, the text generation should be simpler than writing the code by hand, especially when scaled to large projects. The following sections investigate our M2T tool that we believe addresses these limitations. We describe the structure, input and output domains, and usage of the tool with emphasis on usability. For simplicity, all input models to the M2T tool developed are encoded as Prolog tables.

VM2T Tool

The M2T tool we developed contains many components, each playing an important role in the transformation process. These components are described in the subsections below followed by a brief overview of the tool's architecture.

Velocity (Apache)

At the heart of our tool is the Apache Velocity engine, a Java-based template engine designed to access and print data stored in Java classes. While Velocity was developed primarily for web applications, its Java foundation allows for versatility lending itself to M2T and source code generation. The engine has the flexibility to handle any input or output format and is moderately straightforward to use. In addition, Velocity documentation is plentiful, with developer guides, wikis, articles, and books.

Velocity is built around three structures: a context, a template, and an engine. The context contains objects of various types and is the framework for holding well-formatted input data which Velocity can later access via the Velocity template. In addition to accessing data from the context, the template, written in ***Velocity Template Language*** **(VTL)**, also describes the rules for output code generation. The Velocity engine takes the information provided by a context (for us, a prolog database) and template and generates the output text. Figure V1 illustrates a category that summarizes the basics of M2T generation using Velocity.

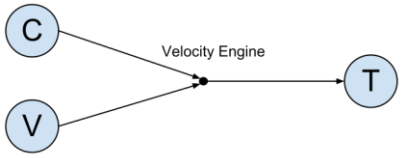


Figure V1: Category of M2T generation using Velocity. C is the set of velocity contexts, V is the set of velocity templates and T is the set of output text from the M2T transformations.

As stated in the caption of Figure V1, C is the set of Velocity contexts, V is the set of Velocity templates, and T is the set of output texts generated from the Velocity engine. More specifically, an element of C and an element of V are fed to the Velocity engine where they are composed and transformed to the desired output text. This process can be represented as the function:

$$VelocityEngine: C \times V \rightarrow T \quad (1)$$

Strictly speaking, this functional representation is not how Velocity's software is structured. For more information on the inner workings of Velocity, we refer the reader to the Velocity Developer Guide.[1]

With a high level understanding of Velocity fresh in our minds, we turn our attention to VTL. VTL is a simple and powerful scripting language. A VTL statement begins with a **#** character and contains a directive. Commonly used directives are **`set`**, **`foreach`**, **`if`**, and **`else`**. References to the current context are denoted by the **$** character. Variables, properties, and methods are the different types of VTL references. Any text that does not have the # or **$** character is automatically added/printed to the output (file or console). The text shown below is a simple Velocity template (`hello.html.vm`) that generates a customized HTML page for the value of *person* in the context.

```
<html>
<body>
     Hello $person
</body>
</html>
```

To access data in a template, a context must be created. Velocity includes a basic implementation class called VelocityContext for general purpose needs. Objects are added to the context through the Java Hashtable class. Figure V2 shows how to add the variable `person`---associated with the value `Jack`---to the context. Pairing the template created above with the code below will result in an HTML page that reads "Hello Jack". The output of running the code with the context in Figure V2 is shown in Figure V3.

[1] Velocity produces a single text file as output. VM2T extends velocity to output multiple files.

```
VelocityContext context = new VelocityContext();
context.put("person", new String("Jack"));
template = Velocity.getTemplate("hello.html.vm");
StringWriter sw = new StringWriter();
template.merge( context, sw );
```

Figure V2: Supplying data through the basic class to the Velocity template

```
<html>
<body>
    Hello Jack
</body>
</html>
```

Figure V3: The customized HTML Output for Jack based on `hello.html.vm`

To use Velocity within Eclipse, three Apache .jar files (commons-collections, commons-lang, and velocity) need to be included in the Java class path. There are several Eclipse plugins that can be used to format Velocity template code. After evaluating Velocity UI (http://veloedit.sourceforge.net/) and VeloEclipse (http://code.google.com/p/veloeclipse/), we did not use them.

Our Velocity Customization

Our group has created an Eclipse Java project that contains a Prolog parser, the tool packed in a .jar file, and example input and template files. ***The benefits of using our tool are that it does not require any Java coding, automatically accepts Prolog input, allows you to split the output text into multiple files, and negates the need to download the Velocity engine.***

Prolog Parser

As the model inputs to our tool are encoded as Prolog facts, we wrote a Prolog parser to access the model data and subsequently store it within a VelocityContext. This eliminates the need to write custom Java classes to encapsulate the context for each new Prolog input file. The parser takes a Prolog file containing Prolog facts as input and gives the template engine direct access to the table data. The parser only requires that the Prolog table schemas be defined, as seen in the Prolog comments of Figure V4.

```
%table(node, [id, name, type]).
node(1, Start, start).
node(2, Ready, state).
node(3, Drink, state).
node(4, Eat, state).
node(5, Family, state).
node(6, Stop, stop).

%table(edge, [id, startsAt, endsAt]).

edge(1, Start, Ready).
edge(2, Ready, Drink).
```

```
edge(3, Drink, Drink).
edge(4, Eat, Drink).
edge(5, Drink, Eat).
edge(6, Ready, Eat).
edge(7, Eat, Eat).
edge(8, Drink, Family).
edge(9, Eat, Family).
edge(10, Family, Stop).
```

Figure V4: Family finite-state machine model in Prolog

The first word after the colon is the name of the table, and the remaining words are the names of the columns. With the schema definitions, the tuples in each table are accessible as an iterable list of objects appended with the character "S". For example, the following template spits back the node tuples defined in Figure V4:

```
#foreach($node in $nodeS)
node(${node.id}, ${node.name}, ${node.type}).
#end
```

Our modified VTL parser has the capability to generate multiple output files. A marker is used to specify where the user can split a single template into multiple output files and needs to be set in the first line in the template:

```
#set($MARKER="//----")
```

The quoted string can be anything that will not be found at the beginning of any line in the template (i.e. `//----` must not appear at the beginning of any line in the template). Markers themselves are of the following form:

```
${MARKER}filename.extension
```

Each time a marker appears, all text that follows is placed in a new file with the specified name. For example, the following template file would produce two Java files as output: **A.java** and **B.java**. each containing an empty class.

```
#set($MARKER="//----")
 ${MARKER}A.java
class A {}
${MARKER}B.java
class B {}
```

If no marker is specified, the tool will output the result to the standard output.

2014 Additions (NEW)

We have discovered that Velocity has a number of drawbacks. We've made several changes to improve VM2T.

Command Line Invocation

VM2T now has the command line:

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

At minimum, you must provide a prolog database file and a VM2T template file. Output is directed to the specified output-file, else to standard out. Optionally you can provide a ContextGenerator Class. (I have yet to use this option).

Indenting

#foreach and **#if** statements in Velocity must be paired with **#end**. The problem here is that it is hard – read "impossible"—to clearly match **#ends** with their starting statements. We have modified our version of Velocity (i.e. VM2T) to have any number of blanks stripped from the beginning and ending of lines whose first non-blank character is #. Thus instead of the lump:

```
#foreach (...)
#if(...)
#end
#end
```

We now permit:

```
#foreach (...)
   #if(...)
   #end
#end
```

This should make programming with Velocity easier.

New Template Commands and Variables

VM2T now has one additional command and two additional variable that can be referenced. If a single file is generated, then re-running VM2T simply overrides this file. But when a set of files is produced by VM2T and possibly different files might be output, the standard action would be to delete the directory in which generated files are to be placed, so that "old" files do not corrupt generated output. This is the purpose of the **DELDIR** command (or variable). The following sequence deletes directory a/b/foo, before generating a pair of files:

```
#set($DELDIR="//----")
#set($MARKER="//----")
${DELDIR}a/b/foo
${MARKER}a/b/foo/A.java
class A {}
${MARKER} a/b/foo/B.java
class B {}
```

A limitation of the above approach is that directory names are hardwired. Two new variables are now available. If output-file is specified, you can access its value via the variable **OutputFileName**. So:

```
#set($MARKER="//----")
${MARKER}${OutputFileName}
. . .
```

Directs VM2T output to file ${**OutputFileName**}.

Another new Velocity variable, **Vm2tInFileName**, is the name (including path prefix) to the input prolog file. That is, if "a/b/c/d.pl" is the prolog database, **Vm2tInFileName** = "a/b/c/d". Using this variable may provide more flexibility in generating output. The following sequence deletes a directory (whose name is $**Vm2tInFileName**) before a pair of generated files are placed in it:

```
#set($DELDIR="//----")
#set($MARKER="//----")
${DELDIR}a/b/$Vm2tInFileName
${MARKER}a/b//$Vm2tInFileName/A.java
class A {}
${MARKER} a/b//$Vm2tInFileName/B.java
class B {}
```

Errors

VM2T will report errors in one of two ways:

1. If a fundamental error in VM2T logic is discovered, errors will be reported to Standard.err.
2. If errors due to VM2T processing (i.e., due to the use of an erroneous template), errors will be reported in file **error.txt**.

This change was required to integrate VM2T with CatLite tools.

Tool Architectural Overview

Figure V5 provides a high-level architectural overview of our M2T tool that illustrates how each component described above is connected.

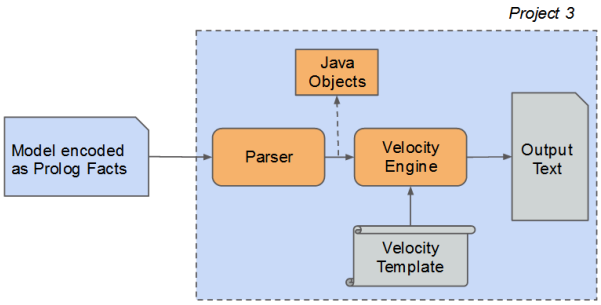


Figure V5. High-level architectural overview of our M2T tool. The dashed box represents the M2T portion of the MDE process.

In Figure V5, we see that the tool receives a model input encoded as Prolog tables from some external source upstream. These tuples are passed through the back-end into a Prolog parser that stores these Prolog table entries into a list of Java objects. This list is placed in the Velocity context where they are accessible by any calling Velocity template. The context and template are then fed into the Velocity engine to produce the output text specified by the user-written Velocity template code. Incidentally, the schematic of Figure V5 can be represented with a category; this is examined below in the subsection *A Note on M2T in Categories*.

The dashed box represents the boundary of our M2T tool while the orange nodes represent pieces of the tool that are automated. We see that the only thing our tool requires for M2T transformations is a user-written Velocity template. Many such examples are given in the *Tutorials* section of this report.

Running the Tool

Our tool can generate the output text either by command line invocation or in Eclipse. Our tool takes the following parameters:

```
Command line arguments (has to be in this order):
                prolog-file template-file
```

To use our tool via the command line, you simply need to pass the input model and the template to the release/vm2t.jar found in our Github project.

```
java  -jar  m2t-1.0.jar  prologModel.pl  velocityTemplate.vm
```

In order to import the project into Eclipse, download the latest version of the project in a zip file; you do not need to unzip the file. In Eclipse, go to *File* menu and select *Import*. From there, choose the *Existing Projects into Workspace* option. Then select the *Select archive file* and finish by selecting the downloaded zip file and clicking on *Finish* button. The tool requires that the workspace Java Compiler Compliance is set to at least 1.6. A tutorial video on this process can be found here.

In Eclipse, code generation occurs by creating a Java application run configuration. The Main class should point to the core.Main file (Figure V5.5), which requires the following program arguments:

```
prolog-file template-file
```

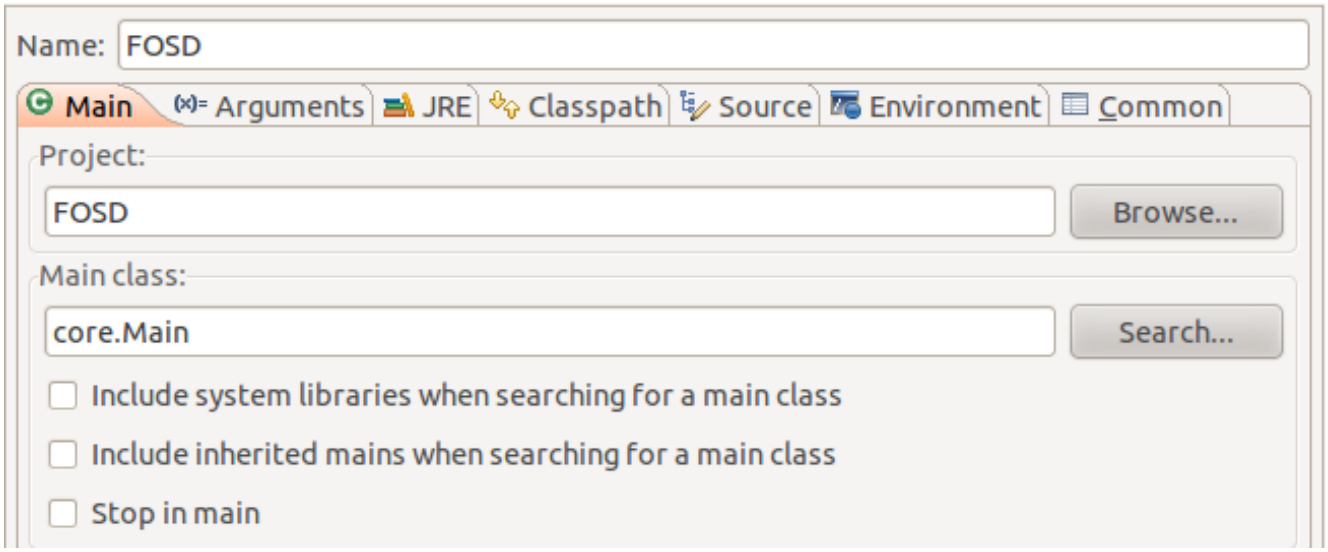


Figure V5.5: The run configuration for our tool

Figure V6 shows the program arguments to generate the Java code for the Family Finite State Machine example in Appendix A in Eclipse's Run Configuration menu.

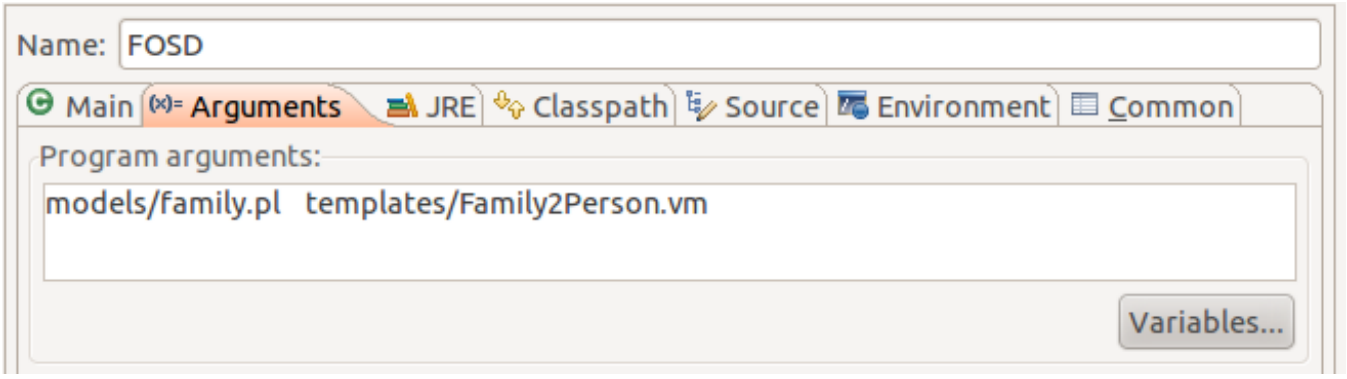


Figure V6: The program arguments for the Family Finite State machine example.

A Note on Generating Output for All Examples

A Python script (`src/generateExamples.py`) will automatically generate the output for all the examples included in the Eclipse Project. The following shows how to run the script via the command line:

```
python src/generateExamples.py
```

Note that the script requires the user to be in the Eclipse project directory and a Python installation.

A Note on M2T in Categories

On a more abstract level, our tool can be thought of as the arrow mapping between objects in a category. This is demonstrated in Figure V7 which shows a portion of MDE as a grouping of categories.

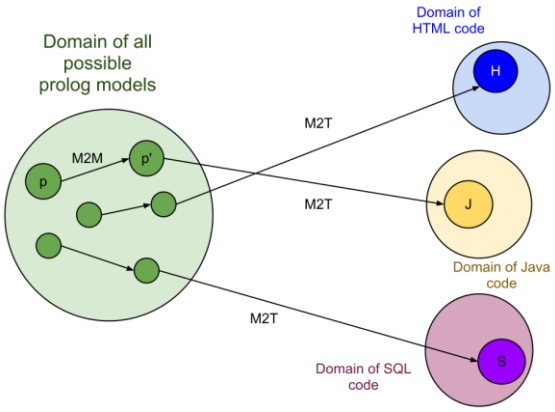


Figure V7. A portion of MDE as categories.

Shown in Figure V7 are four domains (large circles) containing instances of that domain (small circles): models encoded with Prolog (green), HTML (blue), Java (yellow) and SQL (purple). Additionally, we can see an example of a category given by:

$$P \rightarrow P' \rightarrow J$$

In the above category, one Prolog model P is taken to another Prolog model P' by performing a model-to-model (M2M) transformation. Then, P' is transformed to an instance of Java source code, J, by performing an M2T transformation. Our tool is the arrow mapping between the Prolog model domain to one of the remaining domains. More generally, our tool performs an M2T transformation, taking a specific Prolog model and mapping it to an instance of another, different domain (inter-domain transformation). These right-most instances can be thought of as the output code generated from our tool. Incidentally, the Prolog M2M transformation only maps between instances within the Prolog model domain (i.e. an endogenous transformation). To round out our understanding of M2T and its relation to categories, we can begin to consider some interesting properties of the figure above. For instance, we may want to know if the M2T mapping is injective (i.e. there is a single instance in the target domain to which the prolog model can map). This is important as it informs the shape of our categories. While this idea has not been considered in full detail, it may be of interest to the reader.

Tutorials

We present two examples. The paired input and template files are located in the examples folder in the Eclipse project. The first is relatively simple while the second is more complex. To handle the complexity of the second, the first example is heavily detailed.

Example 1: Class to Relational

Class to Relational describes a transformation from a class schema model to a relational database structure. For instance, a class with several attributes might be converted to a relational table where the class name is the name of the table and the attributes become the table's data fields. The metamodels of both the class schema model and the relational database model can be found here. Though simple, this is a practical example of how to learn our tool. As we are concerned with M2T transformations only, we begin with the Prolog tables that represent only the relational database model after an M2M transformation from the class model. Thus, for this example we take as input the Prolog model for a relational database (Class2Relational_2.pl) given below in Figure M3. It should be noted that the M2M transformation from the class schema model to the relational database model shown below is assumed to have been accurately performed.

```
% table(table, [tableId, classId, tableName]).
table(0, c1, family).
table(1, d4, members).
table(2, c2, person).
table(3, d4, emailAddress).


% table( column, [columnId, belongsTo, field, fieldType].
column(0, 0, primary, integer).
column(1, 0, name, string).
column(2, 1, reference, integer).
column(3, 1, memberId, integer).
column(4, 2, primary, integer).
column(5, 2, firstName, string).
column(6, 2, closestFriendId, integer).
column(7, 3, reference, integer).
column(8, 3, emailAddress, string).
```

Figure M3. Prolog tables.

Here, we see that there are four tables to be inserted into a relational database: family, members, person and emailAddress. Recall, we are not concerned with the class schema model which we can assume had information regarding people, families and email addresses; we are only concerned with the Prolog tables representing the relational database model. Note that each table has a table ID. Next, we see that there is a table called column that contains a column ID, a data field and its type, and the table ID to which said data field belongs. With these two tables, we have enough information to convert the Prolog tables representing the relational database model into a .sql script that creates these four tables with the necessary data fields and keys. With our target source code in mind, we must now create a Velocity template to perform the M2T transformation. The Velocity template that produces our .sql script, Class2Relational_SQL_2.vm, is given in whole in Figure M4; detailed explanation of individual lines or portions of code follow. Note that the extracted and explained portions of the code will be formatted such that loops and conditional statements will be indented for clarity. These indentations are not however present in the overall template for reasons pertaining to the formatting of the output text. Additionally, each line in the template in Figure M4 is preceded by a number and colon (e.g. *3:*); this dentoes that it is line 3 of the template. ***These line numbers are shown for clarity/reference and should not be present in an actual template.***

```
1:  #set($MARKER="//----")
2:  ${MARKER}src/classToRelational_2.sql
3:  #foreach($tab in $tableS)

4:  CREATE TABLE ${tab.tableName}(
5:  ## Determine the number of fields in the current table ####
6:  #set($fieldCount=0)
7:  #foreach($col in $columnS)
8:  #if(${col.belongsTo} == ${tab.tableId})
9:  #set($fieldCount=$fieldCount + 1)
10: #end
11: #end
    ################################################################
12: #set($colCount = 0)
13: #foreach($col in $columnS)
14: #if(${col.belongsTo} == ${tab.tableId})
15: #set($colCount=$colCount + 1)
16: #if(${col.fieldType} == "integer")
17: #if(${col.field} == "primary")
18:   ${col.field} INT NOT NULL, PRIMARY KEY(${col.field})#if(${colCount} == ${fieldCount})
19: );#else,
20: #end
21: #else
22:    ${col.field} INT NOT NULL#if(${colCount} == ${fieldCount}));#else,
23: #end
24: #end
25: #end
26: #if(${col.fieldType} == "string")
27:    ${col.field} VARCHAR(50)#if(${colCount} == ${fieldCount}));#else,
28: #end
29: #end
30: #end
31: #end
32: #end
```

Figure M4. Velocity template for the Class to Relational model to text transformation example.

Line 1: `#set($MARKER="//----")`

This line uses Velocity's `#set` tag to set a variable called `MARKER` (denoted `$MARKER`) to the string "`//----`". While the #set tag can be used to set variables of any string or integer variable type, the string given (//----) has a special meaning for our tool; it says that whenever we see the variable $MARKER, we will output a new file at the location/path specified directly after that variable. This is made more clear after observing line 2 below.

Line 2: `${MARKER}src/classToRelational_2.sql`

This line says that when the compiler reaches this line, it should create a file called `classToRelational_2.sql` at the location/path `./src/` .

Line 3: `#foreach($tab in $tableS)`

This line says that we need to loop through each row in the Prolog table `table`. This allows us to access each row and all its associated information. For instance, recall that the Prolog table `table` (found in Figure M3) contains three fields: `tableId`, `classId` and `tableName`. Thus, if we want to access the tableName of the row in which we are currently accessing (the current loop) within the `table` table, we only need to use the variable `$tab.tableName`. In the final output, this will be replaced by the current rows table name. For example, if we were currently in the second loop (i.e. looking at the second row of `table`), the Velocity variable `$tab.tableName` would be replaced by string `members` in the text output file specified in line 2. With that, we have a nice segway to line 4.

Line 4: `CREATE TABLE ${tab.tableName}(`

Here we see that the text that will be written to the output file is the first portion of a standard SQL command to create a table. However, as stated earlier, the variable `${tab.tablename}` will be replaced with the current table's actual table name. As a quick note, you'll notice the use of brackets around the variable. This is to remove ambiguity as to what text is actually part of the variable, much like in bash scripts.

Lines 5-11:

```
## Determine the number of fields in the current table ####
#set($fieldCount=0)
#foreach($col in $columnS)
   #if(${col.belongsTo} == ${tab.tableId})
      #set($fieldCount=$fieldCount + 1)
   #end
#end
################################################################
```

As the comment suggests, this portion of the code determines how many fields are contained within the the current table. We do this by first setting a `fieldCount` variable. Next, using the #foreach tag, we loop through each row of the `column` table. Then we use the #if operator to determine if the `belongsTo` ID of the current column entry matches that of the current `table` entry. If yes/true, we increment the `fieldCount` variable with another call to `#set`. As always, each if and foreach statement are closed with the `#end` tag. Note that these lines generate no code and only compute a property value that we will use later.

Thus far, we have only produced one useful line of source code in our .sql script and that is `CREATE TABLE xxx(` . However, anyone familiar with SQL knows that this is an incomplete declaration. We must also include information about the fields that will be present within that table. The remaining lines deal with this issue.

Lines 12-32:

```
#set($colCount = 0)
#foreach($col in $columnS)
#if(${col.belongsTo} == ${tab.tableId})
#set($colCount=$colCount + 1)
```

```
#if(${col.fieldType} == "integer")
#if(${col.field} == "primary")
   ${col.field}   INT   NOT   NULL,   PRIMARY   KEY(${col.field})#if(${colCount}   ==
${fieldCount}) );#else,
#end
#else
   ${col.field} INT NOT NULL#if(${colCount} == ${fieldCount}));#else,
#end
#end
#end
#if(${col.fieldType} == "string")
   ${col.field} VARCHAR(50)#if(${colCount} == ${fieldCount}));#else,
#end
#end
#end
#end
#end
```

To begin, we first set a variable called `colCount` which will be used later to help correctly print the output file. Next, we loop through each entry/row in the `column` table. If the entry belongs to the current table (remember, we are still in the `#foreach($tab in $tableS)` loop as it was never closed), we increment the `colCount`. Next we must determine if the current column entry is an integer. If yes/true, we must use another if statement to determine if that integer is the primary key (as specified in the Prolog tables). If yes/true again, we can print the SQL statement that instantiates that field in the table such as[2]

```
fieldName INT NOT NULL, PRIMARY KEY(fieldName)
```

In the template, this is represented by the line:

```
${col.field} INT NOT NULL, PRIMARY KEY(${col.field})#if(${colCount} == ${fieldCount}) );#else,#end
```

Here is where it gets a bit complicated. At the end of a line that looks like that directly above, we have two options: one, we can end the statement with a comma which suggests more fields must be instantiated in the table, or two, we can end the statement with a right parenthesis. The latter suggests no more fields will be instantiated in that table. This is where that variable `colCount` comes into play. At the end of the line is an if statement that checks whether or not our colCount is equal to that of the fieldCount. If yes/true, we end the SQL statement with the right parenthesis and semi-colon. If no/false, we add a comma as we know there are more fields to follow. Note that while the formatting of the if-statement looks compact or sloppy, it is only for the preservation of proper formatting in the output file. Moving on, if the field is not an integer, it is a string (for now, only two data types were used to minimize complexity of the example). Using this information, we instantiate the string field in such a way that closely resembles that of instantiating an integer field. Thus, this ends the Velocity template. *Note that if the input Prolog tables contained field types consistent with standard query language, the use of if-statements in the template could be drastically reduced!* This is a good example of how the model informs the template.

[2] Note in Oracle schemas, “PRIMARY KEY” is sufficient.

After running the M2T tool, we obtain the following output .sql script:

```
CREATE TABLE family(
  primary INT NOT NULL, PRIMARY KEY(primary),
  name VARCHAR(50));
CREATE TABLE members(
  reference INT NOT NULL,
  memberId INT NOT NULL);
CREATE TABLE person(
  primary INT NOT NULL, PRIMARY KEY(primary),
  firstName VARCHAR(50),
  closestFriendId INT NOT NULL);
CREATE TABLE emailAddress(
  reference INT NOT NULL,
  emailAddress VARCHAR(50));
```

The above .sql script does exactly what we proposed it should do at the beginning of the example; namely, it builds the four tables present in the Prolog relational database model and includes the appropriate fields. If the Prolog model contained information about entries in the relational database tables, the template---and as a result, the script---could be modified to use that data to populate the database tables. This completes the model to text portion of the Class 2 Relational transformation.

Example 2: UML to Java

UML to Java describes a transformation from a UML class diagram to a Java source representation. For instance, if a class exists in the UML model, the skeleton of this class must be created in Java such that it contains the given attributes, constructors, methods, etc. In order to perform this transformation, a UML metamodel is required such that Prolog tuples can capture the model structure. The ATL zoo gives a nice UML metamodel from which we can base our transformations; this metamodel can be found here. As expected, the metamodel provides all the components needed to fabricate a UML class diagram such as class names, attributes, attribute types, inheritance, scope and more. Before transforming the model to Java code, the UML model information must be encoded as Prolog tables such that they capture the structure of the diagram in question (much like XML/XMI might also be used to represent the UML diagram). The UML model and the corresponding Prolog input tables for the example implemented are shown below in Figures M5 and M6 respectively.

Before continuing, it is important to observe the structure of the Prolog tables/facts in Figure M6 (`UML2Java.pl`); many tables are present including `class, namespace, attribute, operation` and `parameter.` The `class` table/facts contain information about the desired class name, visibility, inheritance and abstraction. Thus, this portion of the model will allow greater specificity in the generated java source code (i.e. class templates); with this information, we aren't limited to to public, concrete classes. The `namespace` and `package` tables allow for greater control over the location and package settings of the java source, respectively. The `attributes` table defines the visibility, type and ownership of data fields within classes. And finally, the `operation` tables combined with the `parameter` tables give all the information necessary to produce the structure of methods within classes (e.g. type, visibility, inputs and output variables).

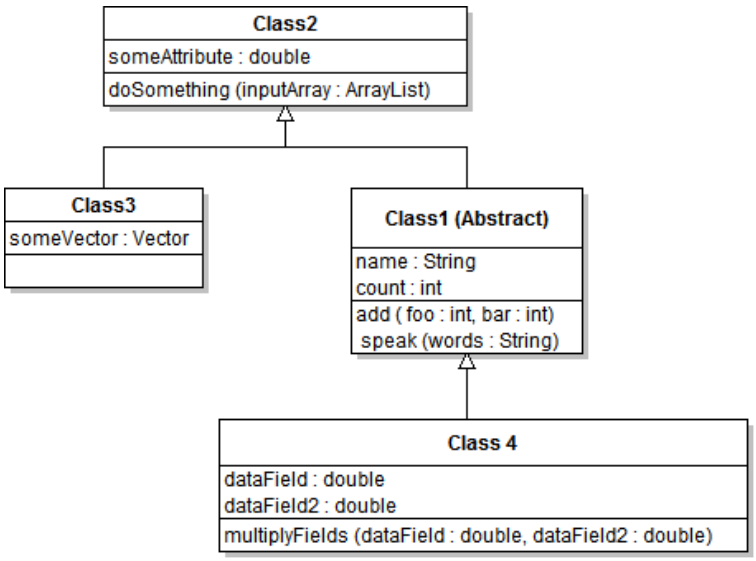


Figure M5. UML class diagram that we would like to transform into Java source code (i.e. generate Java class templates).

```
% table( class, [id, name, visibility, parentId, isAbstract]).
class(1, Class1, public,  2, true).
class(2, Class2, public,  0, false).
class(3, Class3, private, 2, false).
class(4, Class4, public,  1, false).

% table( namespace, [id, name, ownerId]).
namespace(1, util, 1).
namespace(2, util, 2).
namespace(3, util, 3).
namespace(3, util, 4).

% table( package, [id, name, ownerId]).
package(1, java, 1).
package(1, java, 2).
package(1, java, 3).
package(1, java, 4).

% table( attr, [id, name, type, pid, ownerScope, visibility,
changeability, ownerId]).
attr(1, name,          String, 1, ,       public, true, 1).
attr(2, count,         int,    1, static, public, true, 1).
attr(3, someAttribute, double, 2, static, public, true, 2).
attr(4, someVector,    Vector, 3,       , public, true, 3).
attr(5, dataField,     double, 4,       , public, true, 4).
attr(5, dataField2,    double, 4,       ,private, true, 4).

% table( operation, [id, name, type, pid, ownerScope, visibility,
ownerId]).
operation(1, add,           int,    1, static, private, 1).
operation(2, speak,         void,   1,       , public,  1).
```

```
operation(3, doSomething,    void,   2,       , public,  2).
operation(4, multiplyFields, double, 4,       , private, 4).

% table( parameter, [id, name, type, operationId]).
parameter(1, foo,         int,       1).
parameter(2, bar,         int,       1).
parameter(3, words,       String,    2).
parameter(4, inputArray,  ArrayList, 3).
parameter(5, dataField,   double,    4).
parameter(6, dataField2,  double,    4).
```

Figure M6. Input Prolog tables representing the UML model. Each table is an element from the UML metamodel

Now, in order to transform the prolog representation of the UML diagram into the representative Java classes, we must develop a Velocity template that reads and utilizes all of the aforementioned information from the given input tables. A correct template for such a transformation is given in Figure M7 (`UML2Java.vm`). However, as the complete template is quite long, we split it into chunks and analyze each portion. *Note that each portion will be headed/delimited by a gray comment that is also present within the complete template code. This should make cross referencing the template chunks with the whole template simple while avoiding the insertion of artificial line numbers into the template code for reference where there should be none, such as in the preceding example, Class to Relational*. Soon after, the aggregate template (Figure M7) should be more easily understood. Note that each chunk is indented for clarity though the overall template contains no such formatting.

```
## SET VARIABLES ########################
#set($MARKER="//----")
##
#foreach($class in $classS)
${MARKER}src/UML2Java/${class.name}.java
## DETERMINE IF CLASS INHERITS ##########
#set($extends=0)
#if(${class.parentId} != "0")
#set($extends=1)
#set($superId=${class.parentId})
#end

## DECLARE THE PACKAGE ################
#foreach($namespace in $namespaceS)
#foreach($package in $packageS)
#if(${namespace.ownerId} == ${class.id})
#if(${package.ownerId} == ${class.id})
package ${namespace.name}.${package.name};
#end
#end
#end ##foreach
#end ##foreach

## FOR CURRENT CLASS, GET NUMBER OF ATTRIBUTES ####################
#set($attrSize=0)
#foreach($attr in $attrS)
#if(${attr.ownerId} == ${class.id})
```

```
#set($attrSize=$attrSize + 1)
#end
#end
#set($count1=1)
## DECLARE THE CLASS AND ATTRIBUTES ##############################
#if($extends == 0)
${class.visibility}#if(${class.isAbstract} == true) abstract#end class
${class.name} {
#else
${class.visibility}#if(${class.isAbstract} == true) abstract#end class
${class.name} extends #foreach($class in $classS)#if(${class.id} ==
$superId)${class.name}#end#end {
#end

#foreach($attr in $attrS)
#if(${attr.ownerId} == ${class.id})
${attr.visibility} ${attr.ownerScope} ${attr.type} ${attr.name};
#end
#end

## BUILD THE CONSTRUCTORS ###################
/**
* Empty constructor for class ${class.name}
*/
${class.visibility} ${class.name} () {
}

/**
* Constructor for class ${class.name}
*/
${class.visibility} ${class.name} (#foreach($attr in $attrS)#if(${attr.ownerId} ==
${class.id})#if($count1 < $attrSize)${attr.type} ${attr.name}, #else${attr.type}
${attr.name}#end #set($count1=$count1 + 1)#end#end) {
#foreach($attr in $attrS)
#if(${attr.ownerId} == ${class.id})
this.${attr.name} = ${attr.name};
#end
#end
}

## BUILD THE GET/SET METHODS ##########################
// Get/Set methods
#foreach($attr in $attrS)
#if(${attr.ownerId} == ${class.id})
public ${attr.type}
get${attr.name.substring(0,1).toUpperCase()}${attr.name.substring(1)}() {
return this.${attr.name};
}

public void
set${attr.name.substring(0,1).toUpperCase()}${attr.name.substring(1)}(${attr.type}
${attr.name}) {
this.${attr.name} = ${attr.name};
return;
```

```
}
#end
#end

## BUILD THE CLASS METHODS #########################
// Class methods
#foreach($operation in $operationS)
#if(${operation.ownerId} == ${class.id})
#set($params=0)
#set($count1=1)
#foreach($parameter in $parameterS)
#if(${parameter.operationId} == ${operation.id})
#set($params=$params + 1)
#end
#end
${operation.visibility} ${operation.ownerScope} ${operation.type}
${operation.name}(#foreach($parameter in $parameterS)#if(${parameter.operationId}
== ${operation.id})#if($count1 < $params)${parameter.type} ${parameter.name},
#else${parameter.type} ${parameter.name}#end #set($count1=$count1 + 1)#end#end) {
// Fill in method implementation details
}

#end
#end
}
#end
```

Figure M7. Velocity template to transform input Prolog tables to Java source code (class skeletons).

The first portion of the Velocity template deals with setting variables:

```
## SET VARIABLES ##########################
#set($MARKER="//----")
##
#foreach($class in $classS)
${MARKER}src/UML2Java/${class.name}.java
   ## DETERMINE IF CLASS INHERITS #######
   #set($extends=0)
   #if(${class.parentId} != "0")
       #set($extends=1)
       #set($superId=${class.parentId})
   #end
```

First we set the variable `$MARKER` as the string "`//----`". Every time this variable is seen, an output file is created at the location/path that directly follows. Now that we've set the marker variable, we must loop through each class in the `class` table; this corresponds to each row in the `class` table. This is done by using the Velocity tag `#foreach`. NOTE: the `#foreach` tag must be followed with a `#end` tag. By doing so, we can extract all the information pertaining to that class and build the java class skeleton. Before continuing, we must first determine if the current class in the loop (given by the variable `$class`) is a subclass. If it is, we set the `$extends` variable to 1. Now that some of the class specific variables are set, we must move on to setting the namespace and package of the current class in the loop:

```
## DECLARE THE PACKAGE #################
#foreach($namespace in $namespaceS)
   #foreach($package in $packageS)
      #if(${namespace.ownerId} == ${class.id})
         #if(${package.ownerId} == ${class.id})
            package ${namespace.name}.${package.name};
         #end
      #end
   #end
#end
```

Though it looks complicated, this portion of the template has a very simple function. First, it loops through each namespace and package entry in the namespace and package tables. If the package and namespace's ownerId equals the classId of the current class in the class loop (recall that we hadn't closed the first #foreach that loops through the class), we use the namespace and package names to declare the package within the java class.

Next, we need to determine how many attributes the current class contains. This will help in printing the constructor of the current java class:

```
## FOR CURRENT CLASS, GET NUMBER OF ATTRIBUTES ####################
#set($attrSize=0)
#foreach($attr in $attrS)
   #if(${attr.ownerId} == ${class.id})
      #set($attrSize=$attrSize + 1)
   #end
#end
```

The above template chunk loops through each entry in the attr table and increments a counter `$attrSize` when an attribute's ownerId equals the current classes ID--basically, we are counting how many fields should exist within the current class; just as in the *class to relational* example above, this attribute counter will be used in building/printing the class constructor. More specifically, it will be used to determine if the constructor argument list should be closed with a right parenthesis (i.e. there are no more attributes in the class) or with a comma (i.e. more attributes to be declared in the constructor argument list). So far, we have only generated the package information of the java class. The next step is declaring the class itself and its given data fields:

```
## DECLARE THE CLASS AND ATTRIBUTES ##############################
#if($extends == 0)
   ${class.visibility}#if(${class.isAbstract}    ==    true)    abstract#end    class
${class.name} {
#else
   ${class.visibility}#if(${class.isAbstract}    ==    true)    abstract#end    class
${class.name}     extends     #foreach($class     in     $classS)#if(${class.id}     ==
$superId)${class.name}#end#end {
#end
## ATTRIBUTES ############
#foreach($attr in $attrS)
   #if(${attr.ownerId} == ${class.id})
```

```
      ${attr.visibility} ${attr.ownerScope} ${attr.type} ${attr.name};
   #end
#end
```

Walking through the above template chunk makes its function clear. First we determine if the current class extends another class. If not, we print the standard class declaration (i.e. `public class className { )` using the information given in the `class` table (e.g. name which can be extracted with `$class.name`). You'll notice that there are embedded `#if` statements within these class declarations. This is to insert the word `abstract` into the declaration if the current class is defined as abstract within the Prolog table inputs.

The attributes section is straightforward. Each attribute entry in the `attr` table is looped through. If the attribute's owner ID is equal to the current class's ID, we print the attribute and all its associated values (e.g. visibility, type, etc) to the java class file. The resulting output will look very familiar and of the form:

```
public static String attributeExample;
```

Next we move to printing the constructors:

```
## BUILD THE CONSTRUCTORS ##################
/**
 * Empty constructor for class ${class.name}
 */
${class.visibility} ${class.name} () {
}

/**
 * Constructor for class ${class.name}
 */
${class.visibility} ${class.name} (#foreach($attr in $attrS)#if(${attr.ownerId} ==
${class.id})#if($count1 < $attrSize)${attr.type} ${attr.name}, #else${attr.type}
${attr.name}#end #set($count1=$count1 + 1)#end#end) {
#foreach($attr in $attrS)
   #if(${attr.ownerId} == ${class.id})
      this.${attr.name} = ${attr.name};
   #end
#end
}
```

The top portion of the template is simple and creates the empty constructor for the current class. The bottom portion is not as simple. Here we first access the class's visibility and name to provide the usual format of a class constructor. However, the difficulty comes when we're trying to print the parameters of the non-empty constructor. First, we must loop through each of the attributes and determine which one belongs to the current class by checking if the attribute ownership ID matches that of the current class ID. If it does, we add it to the argument list as usual by first accessing the attribute type then its name. After each iteration of the loop, we iterate the variable called `$count1`. When `$count1` is equal to `$attrSize` (as found using the #if statement) the argument portion of the constructor is closed with a right bracket, ")". If this condition is not satisfied, the attribute loop continues but appends a comma (",") in anticipation of another argument to follow. Once this is finished, we must again loop through each of

the attributes, check if they belong to the current class, and set them in the constructor with the standard `this.fieldName = fieldName` syntax. With that, our constructors have been built.

Next, we must include get and set methods within the class so we can access the data fields. This is done with the following portion of the template of Figure M7:[3]

```
## BUILD THE GET/SET METHODS ############################
// Get/Set methods
#foreach($attr in $attrS)
  #if(${attr.ownerId} == ${class.id})
public ${attr.type} get${attr.name.substring(0,1).toUpperCase()}${attr.name.substring(1)}() {
        return this.${attr.name};
    }

public       void       set${attr.name.substring(0,1).toUpperCase()}${attr.name.substring(1)}(${attr.type}
${attr.name}) {
              this.${attr.name} = ${attr.name};
        return;
    }
  #end
#end
```

Since we want a get and set method for each attribute, we start this portion with a `#foreach` loop that loops through each attribute in the `attr` table. If the attribute ID matches the class ID, we begin to print the standard structure of a set and get method using the attribute's data (e.g. name and type) in the standard java form. You'll notice that when accessing the attribute's name, some standard java operations are called and used on the name string, such as `toUpperCase()`. This is to ensure that the method name adheres to the standard java practice of using camel notation. After the methods are described, the method bodies are developed by accessing the information in the `attr` table (such as `$attr.name` which prints the name of the attribute) in such a way that resembles standard get and set methods within a java class.

Finally, we end with printing the class methods themselves given by the `operation` table:

```
## BUILD THE CLASS METHODS ##########################
// Class methods
#foreach($operation in $operationS)
#if(${operation.ownerId} == ${class.id})
#set($params=0)
#set($count1=1)
#foreach($parameter in $parameterS)
#if(${parameter.operationId} == ${operation.id})
#set($params=$params + 1)
#end
#end
${operation.visibility} ${operation.ownerScope} ${operation.type}
${operation.name}(#foreach($parameter in $parameterS)#if(${parameter.operationId} ==
```

[3] Sadly, there is no line-continuation character in Velocity, so everything that is to appear on one line must be on a single VTL line.

```
${operation.id})#if($count1 < $params)${parameter.type} ${parameter.name},
#else${parameter.type} ${parameter.name}#end #set($count1=$count1 + 1)#end#end) {
// Fill in method implementation details
}

#end
#end
}
#end ##end to the class foreach loop and end of template
```

Again, building the portion of the template that consists of printing the class methods seems daunting. However, we've seen a very similar portion of template code for printing the class constructor. The big portion of Velocity code with all the embedded `#if`'s and `#foreach`'s is really just a way to print the method arguments in a nice way (i.e. when there are no more parameters that belong to the current method, cap the printed line with a right bracket ")"; otherwise, print a comma "," in anticipation of more arguments). However this time, instead of comparing the number of parameters counted so far in the argument list to a variable called `$attrSize` (such as in the constructor), we use the variable `$params`, which is determined *before* we start building the method. The final `#end` statement is closes the very first `#foreach` loop that was looping through each class in the `class` tables. As such, the template is finished and we are ready to run the tool to perform the M2T transformation.

Once the tool is run (which is as simple as running Main.java with the necessary arguments), four java classes will be produced, each corresponding to a class in the UML class diagram given in Figure M5. In order to minimize space, only one of the java class---`Class1.java`---is shown:

```
package util.java;

public abstract class Class1
extends Class2 {

public String name;
public static int count;

/**
* Empty constructor for
class Class1
*/
public Class1 () {
}

/**
* Constructor for class
Class1
*/
public Class1 (String name,
int count) {
this.name = name;
this.count = count;
}
```

```
// Get/Set methods
public String getName() {
return this.name;
}

public void setName(String
name) {
       this.name = name;
return;
}
public int getCount() {
return this.count;
}

public void setCount(int
count) {
this.count = count;
return;
}

// Class methods
private static int add(int
foo, int bar) {
// Fill in method
implementation details
}

public  void speak(String
words) {
// Fill in method
implementation details
}
}
```

We see that all information in the input model's tables are considered; the class is abstract and extends `Class2`, it contains the proper data fields, it has the appropriate constructors and, finally, it contains all necessary methods. Note that the fact that this class is abstract is purely illustrative. It is not meant to show the actual structure of an abstract class but rather that abstraction can be placed in the class definition.

With this example we see the power of M2T transformations. If the UML class diagram is large and complex, the M2T transformation given drastically reduces development time as the java classes are created quickly, automatically and error-free.

Additional Examples

Additional examples---including the input file, Velocity template and output code---are given in the *Appendix* at the end of this report. Some of these examples include transforming a finite state machine from a prolog model to both java source code and the Graphviz language. If necessary these examples in the appendix provide the user more experience using templates. If additional examples are required,

the user can find 20 input files and their corresponding templates for M2T transformation within the tool package itself (again found at https://github.com/amshali/FOSD). Each of these transformations can be run from Eclipse (or the command line) using the procedure outlined in the subsection above called *Running the Tool*.

Conclusion

We presented an overview of *Model to Text* (M2T) transformations along with our tool that automates much of the M2T process. Our tool, a customization of the Velocity template engine, has several benefits: *it requires no Java coding, automatically handles Prolog input, allows you to split a single template into multiple output files, and negates the need to download the Velocity engine.* All that is required is a single user-created template file that specifies the transformation, written in the *Velocity Template Language* (VTL).

There are some limitations to our tool. For instance, VTL does not handle complex conditional statements or logic containing *AND* or *OR* conjunctions. The workaround is to set a variable that evaluates the complex logic expression and then test that variable's value in the conditional statement. Additionally, formatting the output layout (indents, spacing, etc) can also be difficult as each space in the template is printed verbatim. This is made more complicated as spaces within VTL loops will begin to compound, pushing all desired text to the side---a neurosis that becomes clearer when experimenting with Velocity. The user is forced to sacrifice template readability for output text readability. Finally, our current Prolog parser does not support the capability to join tables. This functionality would require additional Java coding by the end user. (Another way is to start with a different set of Prolog tables where these joins have been computed. Such computations would be written in Prolog and executed prior to the use of our tool).

In the end, these limitations---which can be circumvented---are outweighed and outnumbered by the advantages and functionality of the tool:

1. Accelerated creation of redundant or boiler-plate code
2. No back-end Java coding required
3. Minimization of bug propagation
4. Enhanced project maintenance through use of templates for code generation (i.e template provides single point of change)

As a result, our tool and its use in M2T transformations decreases development time and cost while providing the end user easy access to maintenance and versatility in the source code.

Appendix A: Family Finite-State Machine model

This example uses a set of Prolog facts (`fsm-family.pl`) that we used early in the semester, shown below. The *node* facts represent a state machine with 1 *Start* node, 1 *Stop* node, and 4 intermediate nodes representing the states *Ready*, *Drink*, *Eat*, and *Family*. The *edge* nodes represent connections between the 6 nodes.

```
%             table(node,
[id,name,type]).
node(1, Start, start).
node(2, Ready, state).
node(3, Drink, state).
node(4, Eat, state).
node(5, Family, state).
node(6, Stop, stop).

%       table(       edge,
[id,startsAt,endsAt]).
edge(1, Start, Ready).
edge(2, Ready, Drink).
edge(3, Drink, Drink).
edge(4, Eat, Drink).
edge(5, Drink, Eat).
edge(6, Ready, Eat).
edge(7, Eat, Eat).
edge(8, Drink, Family).
edge(9, Eat, Family).
edge(10, Family, Stop).
```

We have constructed two examples using this set of Prolog facts. The first is to automatically generate a graph representing the FSM that is similar to the hand-drawn one used in the slides. To do this, we leverage Graphviz (http://www.graphviz.org/), a tool that take a graph encoded in a domain-speicifc language and outputs a graph image. Below is our template (`graphviz-fsm.vm`) that generates the Graphviz code describing the FSM.

```
digraph fsm {
  node [shape=circle];

#foreach($node in $nodeS)
  #if ($node.type == "start")
    ${node.name}        [label        =
"${node.name}", shape=doublecircle];
  #elseif ($node.type == "stop")
    ${node.name}        [label        =
"${node.name}", shape=doubleoctagon];
  #else
    ${node.name}        [label        =
"${node.name}"];
  #end
#end
```

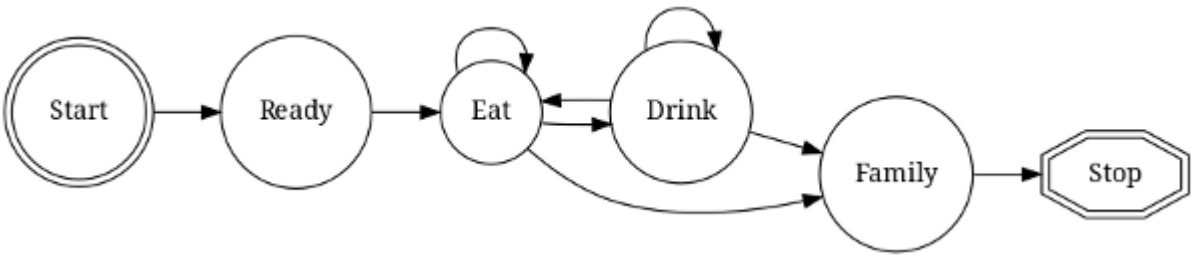
```
#foreach($edge in $edgeS)
  ${edge.startsAt} -> ${edge.endsAt};
#end
}
```

Below is the output of the template. For this example, the Graphviz language should be fairly straightforward. We declare a digraph with all the necessary nodes and labels. After the node declarations, the nodes are connected exactly as they are in the input model.

```
digraph fsm {
node [shape=circle];
rankdir=LR;
Start [label = "Start",
shape=doublecircle];
Ready [label = "Ready"];
Drink [label = "Drink"];
Eat [label = "Eat"];
Family [label = "Family"];
Stop [label = "Stop",
shape=doubleoctagon];

Start -> Ready;
Drink -> Drink;
Eat -> Drink;
Drink -> Eat;
Ready -> Eat;
Eat -> Eat;
Drink -> Family;
Eat -> Family;
Family -> Stop;
}
```

Below is the generated code run by Graphviz. As can be seen, it is a clean representation of the data model.



Our other example using the FSM Prolog model is to generate simple Java skeleton code that might be used as a baseline for executing the FSM. Our template (`java-fsm.vm`) is shown below.

```
#set($MARKER="//----")
${MARKER}Current.java
package fsm;

class Current {
  public static Current current = new
Start();
#foreach($node in $nodeS)
  public void goto${node.name}() {}
#end
}

#foreach($node in $nodeS)
${MARKER}${node.name}.java
package fsm;

class $node.name extends Current {
#foreach($edge in $edgeS)
  #if ($edge.startsAt == $node.name)

  public void goto${edge.endsAt}() {
    current = new ${edge.endsAt}();
  }
  #end
#end
}
#end
```

For our set of inputs, we generate seven classes. The first is a “Current” class that holds the node that is currently being executed. The other six subclass Current and define methods that allow correct state transitions.

Appendix B: Static Method Call

For this example we return to this Graphviz language but present a new data model. The model in this case is a Prolog file which describes a number of classes and the methods that they have in addition to the information which shows what static methods calls there are between different methods. An example of such a Prolog model of method call (`method-call-1.vm`) is shown below:

```
% Classes
% table(class, [ id, name ]).
class(c1, ClassA).
class(c2, ClassB).
class(c3, ClassC).

% Methods
% table(method, [id, classid, name]).
method(m1, c1, noCalls).
```

```
method(m2, c1, recursive).
method(m3, c2, back).
method(m4, c2, forth).
method(m5, c3, callsEveryone).
method(m6, c3, everyoneCalls).

% Invocations
% table( invocation, [caller, callee]).
invocation(m2, m2).
invocation(m2, m6).
invocation(m3, m4).
invocation(m3, m6).
invocation(m4, m3).
invocation(m4, m6).
invocation(m5, m2).
invocation(m5, m3).
invocation(m5, m4).
invocation(m5, m5).
invocation(m5, m6).
```

The template (`graphviz-method-call.vm`) for this transformation is shown below:

```
digraph G {
  concentrate=true;
  overlap=false;
  splines=true;
  ranksep=0.7;

#foreach($class in $classS)
  subgraph cluster_${class.id} {
    color=grey;
    labeljust="l";
    node [style=dotted, shape=box];
    label="${class.name}";
  #foreach($method in $methodS)
    #if(${method.classid}            ==
${class.id})

    ${method.id}
[label="${method.name}"];
    #end
  #end

  }
#end
#foreach($invok in $invocationS)
  $invok.caller -> $invok.callee ;
#end
}
```

Our model-to-text tool will convert this model into a text which is the encoding of the method call graph. The output of our tool is shown in Appendix A. We then use the graphviz tool to convert that text into an image which is shown below: