Ingredients

Reading: Czarnecki and Helsen [1] survey model transformation technologies. This is a useful, if dated, state of the union regarding model transformation. Most of the technologies mentioned are still available, and the catalog of main characteristics of these technologies has not changed much since then.

Other than that we rely on TMF documentation: the MWE2 tutorial and the Xtend tutorial.

1 Model Transformations: Applications & Classification

⚠ Let us recall the distinction between syntax and semantics of a language:

> *syntax:* the principles and processes by which sentences are constructed in particular languages [Chomsky]
>
> *semantics:* the study of meanings [Merriam-Webster English dictionary]

So far we were focused with defining the syntax of DSLs in efficient ways. We worked with abstract and concrete syntax. We have seen tools that can transform syntax definitions into model editors. Now we are going to turn our attention more towards the semantics of modeling languages — so to implementing language processors.

The standard way to give semantics to programming (modeling) languages is writing an interpreter or a compiler. We will focus mostly on compilers, i.e. we will consider translating models to other languages.

Translating of models between languages is called *model transformation*. Model transformation is a form of meta-programming, so programming that treats other programs (models) as data. In that sense it is a more advanced programming activity than usual programming.

⚠ Applications of model transformation include [1]:

- **Generating** lower-level models and eventually code, from higher-level models
- **Mapping** and **synchronizing** among models at the same level or different levels of abstraction
- Creating a **query-based views** on a system

- Model **evolution** tasks such as model **refactoring**, model versioning, diffing and patching
- **Reverse engineering** of higher-level models from lower-level ones

From all these points the first one is most certainly most important, and of interest to general software development audience. The remaining ones are more internal MDD problems, that are addressed by vendors of MDD tools.

Figure 1 shows an overview of a model transformation set-up.

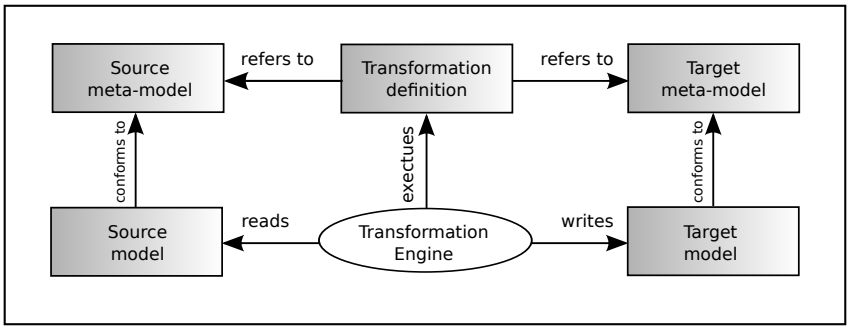

Figure 1: Overview of the model transformation languages and concepts. Figure from [1]

Remark. Program transformation and model transformation are very close, as models and programs are sometimes hard to distinguish. The main objective of program transformation has always been compilation (and optimization of code). Program transformations are based on mathematically-oriented concepts such as term rewriting, attribute grammars, and functional programming. Model transformations usually target object-oriented programming and adopt an object-oriented approach for representing and manipulating models. Model transformations, as they relate models to each other, are often expected to preserve traceability links, which is not done in program transformation [1].

2 Model-To-Text Transformations (M2T)

A large class transformations translate models to text. The text can be generated code, other models in textual syntax, or other textual artefacts such as reports or documentation.

Text is typically generated using *templates*, a technique extremely well established also in web development. A template can be thought of as the target text with holes for variable parts. The holes contain metacode (so code creating code), which ir run at template instantiation time to compute the variable parts [1].

In the following we show two simple M2T transformations that generate respectively HTML and SQL code from models adhering to the trip language. In TMF templates are written using the textual templates of the Xtend language (these templates are sometimes known under an older name, Xpand). Figure 2 shows the template generating a simple HTML report about our trips. Observe how concise is the template:

```
«IMPORT tripLoose»

«DEFINE main FOR TripModel»
«FILE this.name + ".html"»
<html><head><title>Report for Model "«name»"</title></head>
<body>
<h1>Trip Model Report for Model "«name»" </h1>
«FOREACH elements AS e»«EXPAND entry FOR e»«ENDFOREACH»
</body>
«ENDFILE»
«ENDDEFINE»

«DEFINE entry FOR NamedElement»
«ENDDEFINE»

«DEFINE entry FOR Car»
    <strong>Car</strong> «name»<br/>
«ENDDEFINE»
«DEFINE entry FOR Person»
    <strong>Person</strong> «name»<br/>
«ENDDEFINE»
«DEFINE entry FOR Trip»
    <strong>Trip</strong> «name»
    <ol>
    «FOREACH passengers AS p»<li>«EXPAND entry FOR p»«ENDFOREACH»</li>
    </ol>
«ENDDEFINE»
```

Figure 2: Xpand template generating a simple report about trips.

This template generates the following HTML (up to white space) given the same input model as used in the previous lecture:

Output

```
<html><head><title>Report for Model "MyTrips"</title></head>
<body>
<h1>Trip Model Report for Model "MyTrips" </h1>

<strong>Car </strong> WVPolo<br/>
<strong>Car </strong> Trabant<br/>

<br/>

<strong>Person</strong> Andrzej<br/>
<strong>Person</strong> Helge<br/>
<strong>Person</strong> Thorsten<br/>
<strong>Person</strong> Joachim<br/>

<strong>Trip</strong> MDDTrip
<ol>
<li><strong>Person</strong> Helge<br/>
<li><strong>Person</strong> Andrzej<br/>
<li><strong>Person</strong> Thorsten<br/>
<li><strong>Person</strong> Joachim<br/>
```

```
</li>
</ol>
</body>
```

Let us briefly summarize the syntax of the template. The first line imports the meta-model of the language processed. Then we define the main template which is to be executed on elements of type TripModel. The object of this type can be referred to as `this` in the text of the transformation.

The body of the template is treated as a text file, which is verbatim output to the output file. Text in «French quotes»is the meta-code. Use code complete (ctrl-space) to switch to the meta-code mode. The `FILE` instruction opens a new file.

We do a polymorphic expansion for named elements. We bind the element processed to a variable *e* — so that we can access it in the contained `EXPAND` command. This command calls another expansion function. The call is polymorphic, so a different function will be called for objects of various concrete types. These polymorphic templates are given in the bottom of the figure. The transformation handles Cars, but silently ignores other vehicles.

The following template is very similar, but it generates an SQL code for inserting the trip data into a hypothetical relational database:

```
«IMPORT tripLoose»

«DEFINE insert FOR NamedElement»«ENDDEFINE»
«DEFINE insert FOR Person»
    INSERT INTO PERSONS VALUES ('«name»')
«ENDDEFINE»
«DEFINE insert FOR Vehicle»
    INSERT INTO CARS(regnr,seats) VALUES ('«name»','«nrOfSeats»')
«ENDDEFINE»
«DEFINE insert FOR Trip»
    INSERT INTO TRIPS (id,name) VALUES ('«name»','«name»')
    «FOREACH passengers AS p»
    INSERT INTO PASSENGERS (tripid,personid) VALUES ('«name»','«p.name»')
    «ENDFOREACH»
«ENDDEFINE»

«DEFINE main FOR TripModel»
«FILE this.name + ".sql"»
«FOREACH elements AS e»«EXPAND insert FOR e»«ENDFOREACH»
«ENDFILE»
«ENDDEFINE»
```

Figure 3: Xpand template to generate simple SQL.

3 Gluing Things with MWE2

The transformations, model readers, model serializers, validators, and all other model processing components need to be wired together to perform the task you need. This can be

done using a GPL like Java, but it is often easier in some kind of scripting language. Ant can be good for combining very coarse grained components, but since here individual components are Java classes, we need a language that is able to instantiate classes, initialize their parameters and invoke them in the right order. MWE2 is TMF's language for this purpose. It is a kind of *glue* language used to define Workflows.

MWE2 is not a overwhelmingly interesting language in itself, but you will need to know it when working with TMF. In the following I give a quick summary of MWE2, so that you can use it in your tools.

The main concept of MWE is the Workflow. Each script defines one workflow, which is like a model processing recipe, consisting of coarse grain steps.

Here is the simplest work flow (inspired by MWE2 official tutorial):

```
module episode13m2t.trip.generators.HelloWorld

Workflow {

    component = SayHello { message = "Hello!" }

}
```

The first line declares the name of the workflow module (and to which package it belongs). Then our workflow consists of creating, initializing and invoking exactly one component, called SayHello.

A component is a Java class implementing the IWorkflowComponent interface. So to run the above work flow, we need to provide an implementation of such Java class.

Here is one possibility:

```
package org.xtext.example.loosetrip.generators;
import
  org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent;

public class SayHello implements IWorkflowComponent {

  private String message = "Hello World!";

  public void setMessage(String message)
  { this.message = message; }

  public String getMessage() { return message; }

  public void invoke(IWorkflowContext ctx)
  { System.out.println(getMessage()); }

  public void postInvoke() { }
```

```
  public void preInvoke() { }
}
```

Observe that the component implements the method invoke, and it provides an attribute message, with respective access functions. The workflow line activating this component roughly corresponds to:

```
SayHello h = new SayHello ();
h.setMessage (‘‘Hello!’’);
h.invoke(ctx);
```

where ctx is an object storing the state of execution of the workflow.

Now of course this is quite a heavy weight way to say 'Hello' to the world. The MWE2 starts to shine if you realize that the entire TMF infrastructure is based on Java. We will see that some classes can be (for example) implemented in Xtend, which is another programming language of TMF meant for transformation and model manipulation. Moreover, the framework provides many standard components, and some components are generated for your languages, thus you do not have to implement them in Java.

The following work flow reads a model in xtext syntax for trip and saves it in the xmi syntax. So this is already a kind of model transformation, implemented purely in MWE2, as it only relies on standard components:

```
module episode13m2t.trip.generators.M2T

Workflow {
    bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
        platformUri = "../"
        registerGeneratedEPackage = "tripLoose.TripLoosePackage"
        registerEcoreFile = "platform:/resource/episode10xtext.tripLoose/model/tripLoose.ec
    }

    component = org.eclipse.xtext.mwe.Reader {
        path = "input/"
        register = episode10xtext.tripLoose.xtext.TTripLooseStandaloneSetup { }

        load = {
            slot = "tripmodel"
            type = "TripModel"
            firstOnly = true
        }
    }

    component = org.eclipse.emf.mwe.utils.DirectoryCleaner {
        directory = "../episode13m2t.trip/model-gen/"
```

```
    }

    component = org.eclipse.emf.mwe.utils.Writer {
        modelSlot = "tripmodel"
        uri = "platform:/resource/episode13m2t.trip/model-gen/output.xmi"
    }
}
```

The first bean registers the metamodel for our language, and informs the state of the platformUri (the directory of current Eclipse's workspace). Then we call the standard Reader component. This reader reads all the models under the indicated path, and stores elements of type "TripModel" into a *slot* called "tripmodel". A slot is simply an index entry in a map from names to slots. The map of slots is the main component of the context, which is passed around to exchange information between components.

Then we use a directory cleaner, a standard component preparing space for generated artefacts. And finally we use a standard writer (with an xmi) extension, to save a file in the xmi format.

Note that in this workflow there is actually no transformation directly, but simply loading and saving the same model in a different representation.

Finally we show the workflow that is needed to call one of the two M2T transformations presented in the previous sections:

```
module org.xtext.example.loosetrip.generators.M2T

Workflow {

  bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
    platformUri = "../"
    registerGeneratedEPackage = "dk.itu.example.tripLoose.TripLoosePackage"
    registerEcoreFile = "platform:/resource/dk.itu.example.trip/model/tripLoose.ecore"
  }

  component = org.eclipse.xtext.mwe.Reader {
    path = "../org.xtext.example.loosetrip/m2/txt/"
    register = org.xtext.example.loosetrip.LooseTripStandaloneSetup {}

    load = { slot = "tripmodel"
             type = "TripModel" }
  }

  component = org.eclipse.emf.mwe.utils.DirectoryCleaner {
    directory = "../org.xtext.example.loosetrip/model-gen/"
  }
```

```
  component = org.eclipse.xpand2.Generator {

    metaModel = org.eclipse.xtend.typesystem.emf.EmfMetaModel {
      metaModelPackage = "dk.itu.example.tripLoose.TripLoosePackage"
      metaModelFile = "platform:/resource/dk.itu.example.trip/model/tripLoose.ecore" }

    metaModel = org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel {}
      //expand = "templates::DoHtmlReport::main FOREACH tripmodel"
      expand = "templates::DoSQL::main FOREACH tripmodel"

      outlet = { path = "model-gen" }
  }

  component = org.eclipse.emf.mwe.utils.Writer {
    modelSlot = "tripmodel"
    // file extension is important here.
    // ".strip" (so the declared xtext syntax extension) saves in textual syntax
     // ".xmi" or ".ecore" saves in xmi syntax.
     uri = "platform:/resource/org.xtext.example.loosetrip/model-gen/capitalized.strip"
  }
}
```

The only really new thing in this workflow, is a call to the `org.eclipse.xpand2.Generator`, which executes the M2T transformation in xpand.

In Eclipse execute the MWE2 workflows by choosing “Run as” from their context menu.

References

[1] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches, 2006.