An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part V: Model-to-text Transformations

`For eMoflon Version 2.0.0`

For further information contact us at contact@emoflon.org.

The eMoflon team
Darmstadt, Germany (August 2015)

Contents

Part V:

Model-to-Text Transformations

`Approximate time to complete: 1h 30min`

`URL of this document: http://tiny.cc/emoflon-rel-handbook/part5.pdf`

Welcome to Part V of the eMoflon handbook, an introduction to bidirectional model-to-text transformations using Triple Graph Grammars (TGGs). If you're just joining us and haven't completed any of the previous parts, we recommend working through at least Part I for the required setup and installation instructions to ensure eMoflon is working correctly, and strongly encourage finishing Part IV to master the basics of TGGs. That part will be a key reference if you're ever unsure how to use a TGG feature. Apart from TGG fundamentals however, we have assumed as little as possible from any of the previous parts and include appropriate references where necessary.

Up until now in the handbook, we have created `LeitnersLearningBox`, a memorization tool that stores cards (with keywords on the front, and definitions on the back) in different partitions, which then move through the box based on a set of rules simulating how our short and long-term memory works. This metamodel was used in Part IV as a source language in a *bidirectional* TGG transformation, where each card was translated into an entry (with a sole content attribute storing all its information) in a `Dictionary` metamodel. In this part, we shall implement a second bidirectional transformation, this time a model-to-text transformation to establish a textual representation of the `Dictionary` metamodel. We'll use an `ANTLR` [?] parser and unparser, and TGGs to transform the parsed tree from ANTLR to an instance of the `Dictionary` metamodel.

Bidirectional Transformation

When establishing a model-driven solution, *model transformations* usually play a central and important role. They could be used for specifying dynamic semantics (as done for the rules of our learning box) or, more generally, for transforming a certain model to another model to achieve some goal (i.e., checking or guaranteeing consistency, adding or abstracting from platform details, ...).

There are many *types* of model transformations and [**?**, **?**] give a nice and detailed classification along a set of different dimensions. In this part, we shall explore some of these dimensions and learn how *model-to-text* transformations can be achieved with a nice mixture of *string grammars* and *graph grammars*.

For the rest of this part, a model transformation is denoted as:

$$\Delta : m_{src} \to m_{trg}$$

where the source model m_{src} is to be transformed to the target model m_{trg}. Let's review the four primary ways in which Δ can be classified.

Δ is *endogenous*, if m_{src} and m_{trg} conform to the same metamodel. All story driven models (SDMs) built in Part III for `LeitnersLearningBox` are examples of *endogenous* transformations. *Endogenous*

Δ is *exogenous*, if m_{src} and m_{trg} are instances of different metamodels. For example: A dictionary is used to learn new words (similar to a learning box), but is more suitable for use as a reference (i.e., one already knows the words, but may occasionally need a specific definition). In contrast, a learning box is geared towards the actual memorization process. Therefore, one could start with a learning box and, once all the words have been memorized, transform it into a personalised dictionary for future reference. If too many words become forgotten, the dictionary should be transformed back to a learning box. The learning box to dictionary transformation and vice-versa are therefore examples of *exogenous* transformations, and we implemented this in Part IV by using TGGs to transform our `LeitnersLearningBox` to a `Dictionary`. *Exogenous*

Δ operates *in-place* if m_{src} is *destructively* transformed to m_{trg}. The SDMs for our learning box (e.g., `grow` or `check`) are examples of *in-place* transformations as they perform destructive changes directly to a source model, transforming it into the target model. *In-place*

Finally, Δ is *out-place* if m_{src} is left intact and is unchanged by the transformation which creates m_{trg}. The learning box to dictionary transformation with TGGs is an example of an *out-place* transformation. *Out-place*

Although *endogenous* + *in-place* is the natural case for SDMs (as was the case for our learning box), *exogenous* and/or *out-place* transformations can also be specified with SDMs.

To twist your brain a bit, here are a few interesting statements:

- *Out-place* transformations can be *endogenous* or *exogenous*.
- *In-place* transformations can usually only be *endogenous*. *Exogenous* transformations are consequently, always *out-place*. Why?

It should be noted that Δ can be further classified as *horizontal* if m_{src} and m_{trg} are on the same *abstraction level*, or *vertical* if they are not. Unfortunately, this *abstraction level* dimension is a bit 'fuzzy,' but we will explore and work on these different levels by establishing a textual concrete syntax for `Dictionary`. We shall learn how TGGs can be used in combination with parser generators and template languages to implement model-to-text and text-to-model transformations that are typically *vertical* (text is normally on a lower abstraction level than a model). On the other hand, the overall learning box to dictionary transformation completed in the previous part (also with TGGs) is *horizontal* as the models represent the *same* information differently, and can thus be considered to be on the same abstraction level.

1 Setting up your workspace

Nowadays, *no one* writes a complex parser completely by hand. Although this is sometimes still necessary for syntactically challenging languages, most parsers can be quickly whipped up using context-free *string grammars*[1] that are typically written in Extended Backus-Naur Form (EBNF). ANTLR is a tool that can generate a parser from this compact specification for a host of target programming languages, including Java. Although ANTLR might not be the most efficient or powerful parser generator, it's open-source, well documented and supported, and allows for a pragmatic and elegant fallback to Java if things get nasty and we have to resort to some dirty tricks to get the job done. *EBNF*

To set up your workspace for the model-to-text transformation, you have two options: (1) Import a cheat package with everything already prepared (useful if you're just joining us), or (2) if you've worked through the previous part, continue with your existing workspace. Both options should work, but we have only tested and updated all screenshots for Option (1) and thus highly recommend this.

As some of you are just reading this handbook without actually getting your hands dirty with an implementation (beware: no pain, no gain!), we have included a screenshot of the dictionary metamodel that you get with both options in our visual (Fig. 1.1) and textual (Fig. 1.2) concrete syntax.

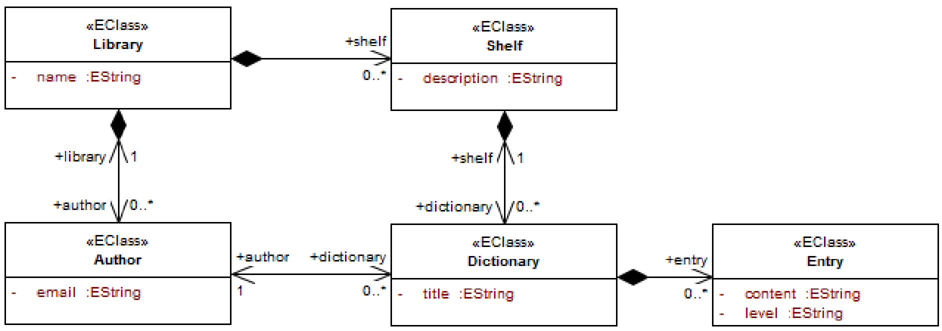


Figure 1.1: Metamodel for dictionaries (visual concrete syntax)

[1]For simple cases, *regular expressions* can also be used

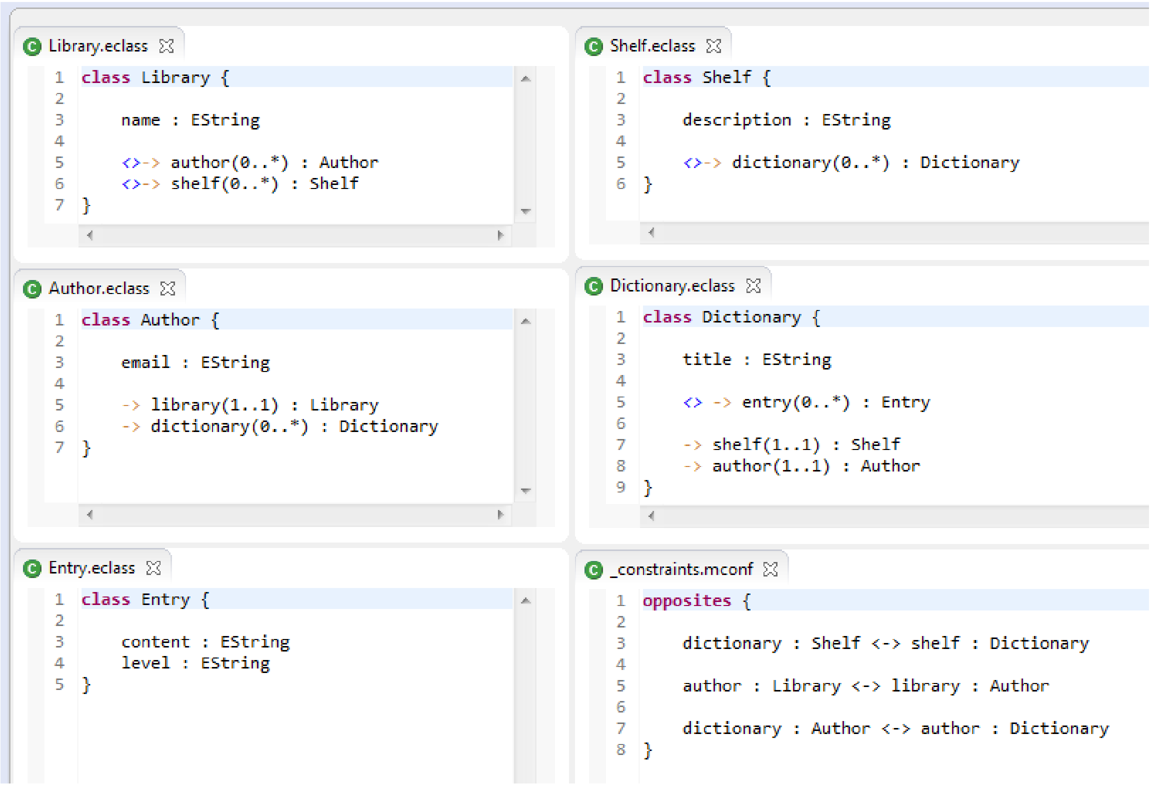


Figure 1.2: Metamodel for dictionaries (textual concrete syntax)

Option 1: Import a complete cheat package

- Import the Part V 'cheat package' by selecting "New" in the toolbar, and the cheat package in the concrete syntax of your choice (Fig. 1.3).

Option 2: Continue with the workspace from Part IV

- Use the same metamodel for `Dictionary` as completed in Part IV. Just make sure you haven't radically changed the dictionary metamodel (i.e., it still closely resembles the metamodel in either Fig. 1.1 or Fig. 1.2). Everything else should work fine using the exact same workspace but remember, your screen may look different than our screenshots.

We recommend reviewing the dictionary metamodel until you feel comfortable with what you'll be working with.

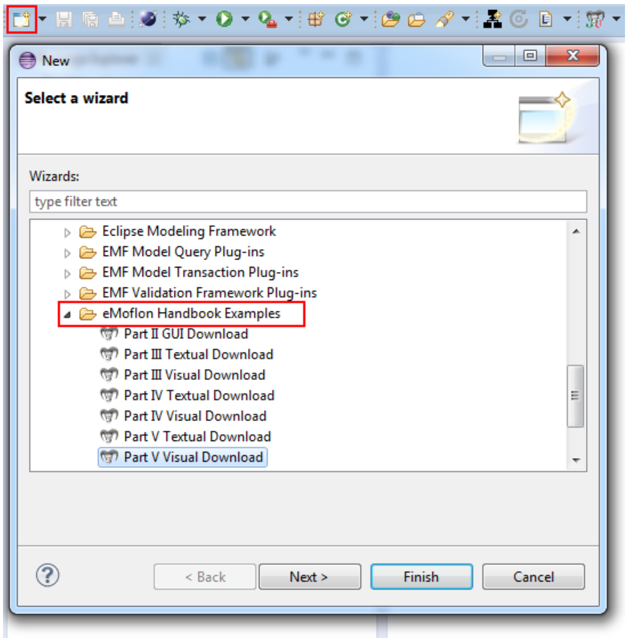


Figure 1.3: Load the cheat package for Part V into your workspace

`DictionaryLanguage` is only one of two metamodels that we'll be using to specify the TGG transformation. After all, TGGs typically require separate source and target metamodels. The second metamodel involved in the transformation will be eMoflon's standard `MocaTree` language.[2] It basically combines concepts from a filesystem (folders and files), XML (text-only nodes and attributes), and a general indexed containment hierarchy. It is provided by our Eclipse plugin and is automatically added to the build path, so it won't actually appear anywhere in your Eclipse workspace.

Figure 1.4 is a visual depiction of this MocaTree model.[3] As you can see, the most important element is `Node`. Note that a single `Node` can store any number of `Attribute` or `Text` elements (subnodes), but only belongs to one `File`. If you look closer at `File`, you'll also notice that it belongs to a single `Folder`. `Folder` is able to store any number of `File`s or subfolders.

[2] MOCA stands for Moflon Code Adapter (not coffee, sorry.)

[3] If you are using the visual syntax, feel free to view a detailed metamodel by opening `dictionary.eap`, navigating to the `MocaTree` EPackage, and opening its diagram.

You can see that all elements inherit an `index` and `name` attribute. `Index` can be used to demand a certain *order* of nodes in a tree, otherwise not guaranteed by default (i.e., to enforce a hierarchy), while `name` can be any arbitrary string value.

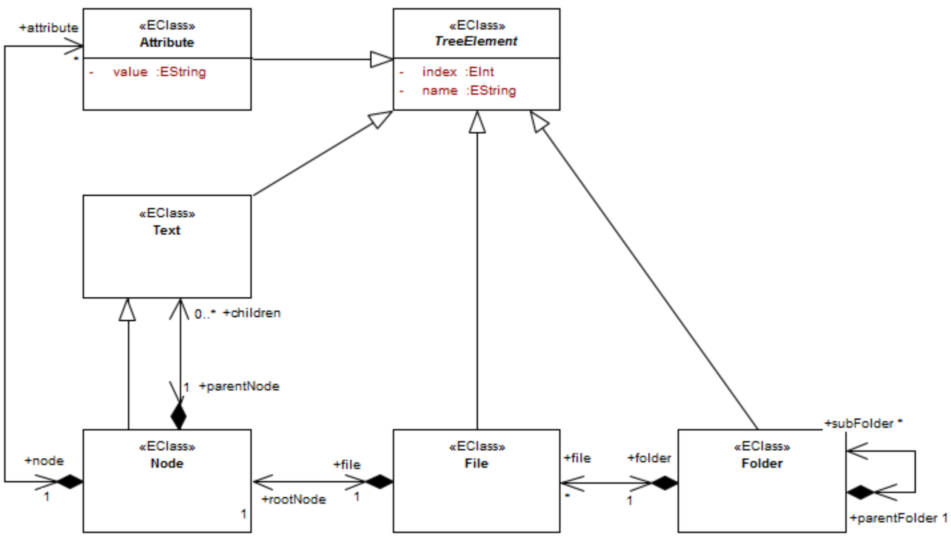


Figure 1.4: Visual depiction of the MocaTree metamodel

Enough chatting – let's begin by creating the TGG project that will implement our model-to-text transformation.

▷ Next [visual]
▷ Next [textual]

1.1 First steps

- ▶ From your Eclipse workspace, open the `Dictionary.eap` file in Enterprise Architect (EA). The project browser should closely resemble Fig. 1.5. As you can see, the project is already populated with `MocaTree` and other built-in metamodels in the `eMoflon Languages` working set.

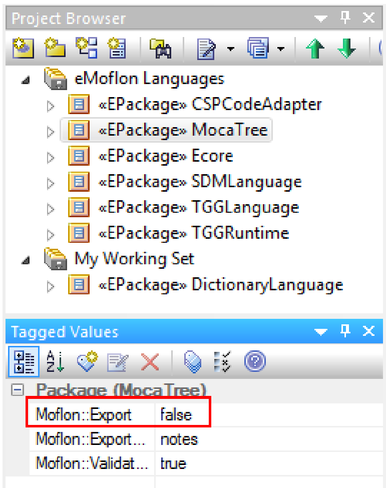


Figure 1.5: `MocaTree` is one of eMoflon's internal metamodels

If you inspect the tagged values[4] for these built-in languages, you'll notice that the `MocaTree` package has the `Moflon::Export` value set to `false`. This ensures that the package is *ignored* when exporting. As with all such standard metamodels (e.g., Ecore or our SDM metamodel) the `MocaTree` package in EA should be regarded as read-only, required in the EA project so that SDMs/TGGs can refer to the classes defined in the package.

- ▶ Despite `DictionaryLanguage` being contained in a different working set than `MocaTree`, the two metamodels are contained within the same EA project (EAP) which means you are able to create a new TGG using them both. Add a new package to `My Working Set` named `DictionaryCodeAdapter`.

- ▶ Select the package and add a new TGG schema diagram as depicted in Fig. 1.6. In the next dialogue window, set the source project as `MocaTree`, and the target project as `DictionaryLanguage`.

[4]The "Tagged Values" window can be opened by going to "View/Tagged Values" or by hovering over the `Tagged Values` tab immediately to the right of the project browser.

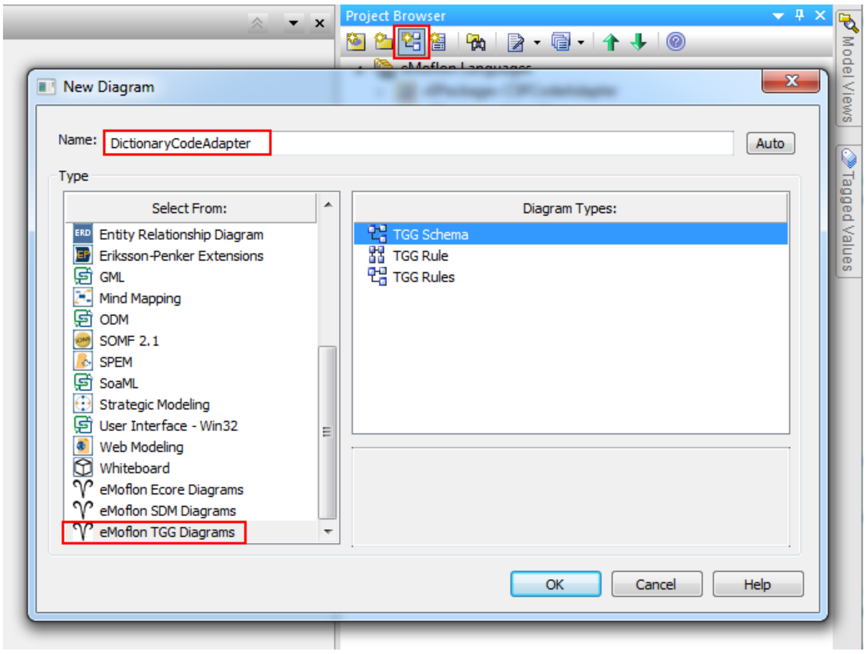


Figure 1.6: Create a new TGG schema diagram

- For the moment, add a single correspondence type to the new diagram now active in the editor (the TGG `schema`) between `Folder` and `Library`. Remember, you can get the classes by drag-and-dropping each element into the diagram, then quick-creating a new `TGG Correspondence Type` between them.[5] Your diagram should come to resemble Fig. 1.7.

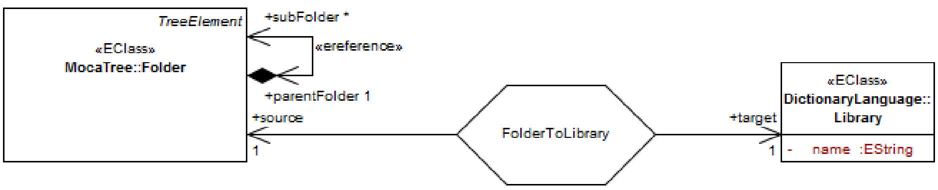


Figure 1.7: The first correspondence type for the transformation

[5]For details on the correspondence metamodel and how to create types, refer to Part IV, Section 3.

- ▶ Your complete project browser should now resemble Fig. 1.8, where `DictionaryCodeAdapter` is now explicitly listed as a `TGGSchemaPackage`.

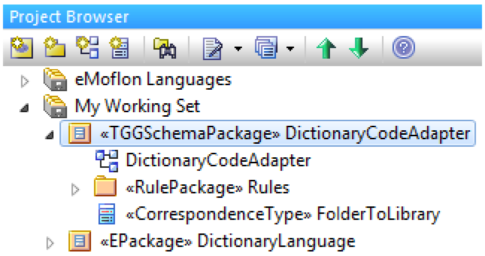


Figure 1.8: A fully prepared TGG project

- ▶ Validate and export your file via the eMoflon control panel,[6] then switch back to Eclipse and refresh the package explorer. A new `DictionaryCodeAdapter` project should appear in `My Working Set`.

[6]Activate via "Extensions/Add-in Windows"

1.2 First steps

In your Eclipse workspace, find and open `Dictionary/MOSL/_imports.mconf` (Fig. 1.9). You'll notice that it's already accessing the `MocaTree` and some other built-in metamodels – you're already able to start with both metamodels.

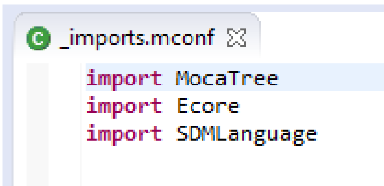


Figure 1.9: `Dictionary`'s imports file

- ▶ Let's create the TGG we'll use to transform `MocaTree` to `Dictionary`. Right-click on `MyWorkingSet`, and navigate to "New/ TGG."
- ▶ Name the package `DictionaryCodeAdapter`, setting the source as `MocaTree` and target as `DictionaryLanguage` (Fig. 1.10).
- ▶ A `schema.sch` file should automatically open in the editor. As a first step, let's add a correspondence type between `Folder` and `Library` as depicted in Fig 1.11.[7] Don't forget that you can use eMoflon's auto-completion feature here!
- ▶ Save and build your project. Confirm that the generated project has a solid black hexagon symbol overlaying the folder, indicating `DictionaryCodeAdapter` is a TGG project, and not just a standard Ecore project (the default project type).

[7]For details on this correspondence metamodel and how to create types, refer to Part IV, Section 3.

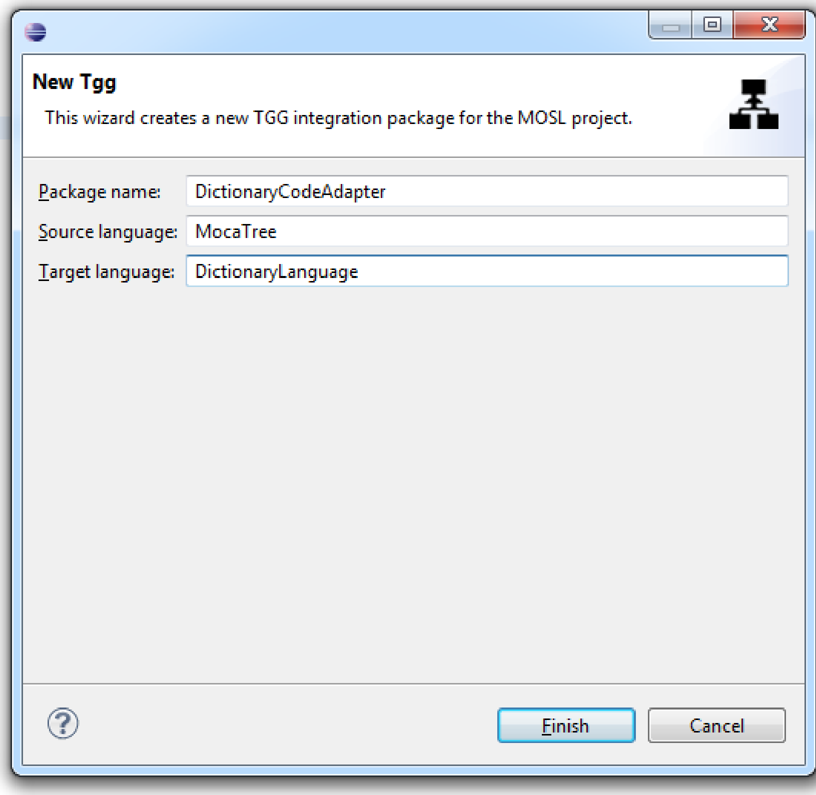


Figure 1.10: Settings for our TGG

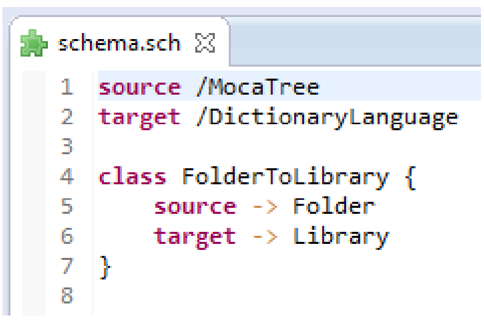


Figure 1.11: Our first correspondence type

1.3 Setting up the parser

Our convention is that the *code adapter* project we established in the previous step contains all tree-to-model transformation logic for the project. Although the transformation *could* be integrated directly in the corresponding metamodel (`DictionaryLanguage`), a separation makes sense here as there could be *different* code adapters for the *same* language.

To continue setting up the framework for our transformation, let's establish an ANTLR parser/unparser which will enable us to transform from tree-to-text (and vice versa).

▶ Right-click on the generated `DictionaryCodeAdapter` folder and navigate to "eMoflon/ Add Parser/Unparser" (Fig 1.12).[8]

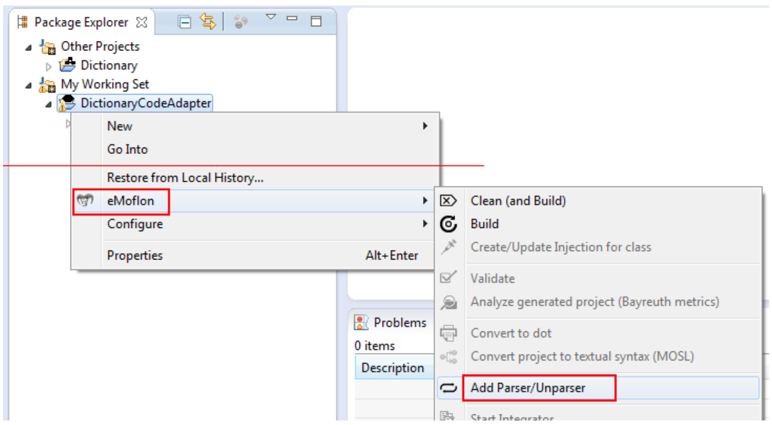


Figure 1.12: Adding a new parser/unparser to a project

▶ In the parser settings window, enter `dictionary` as the `File extension`, and confirm that the `Create Parser`, `Create Unparser`, and `ANTLR` options are selected as the corresponding technology for each case (Fig 1.13). Affirm by pressing `Finish`.

▶ If everything executed without error, parser and unparser stubs should be generated in the `src` package (Fig. 1.14). In addition, a new `in` folder should appear under `instances`.

[8] For presentation purposes, this context menu screenshot has been edited

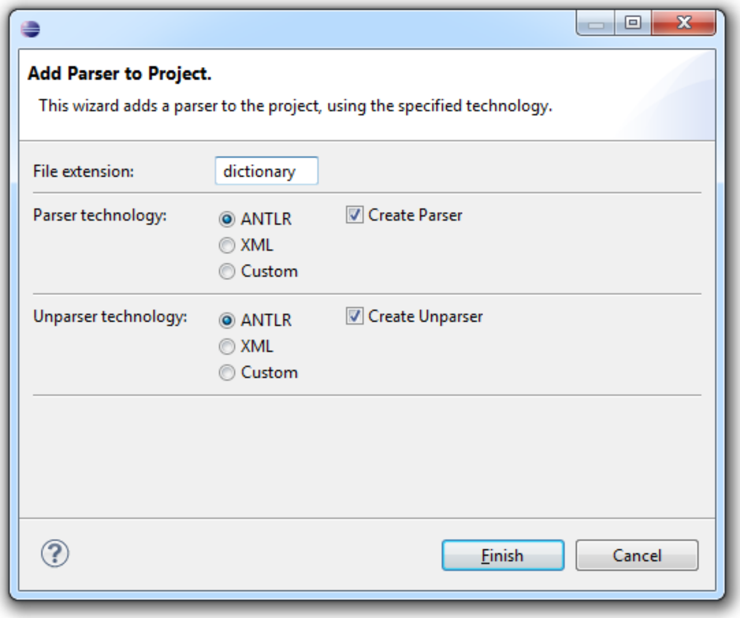


Figure 1.13: Parser/unparser settings

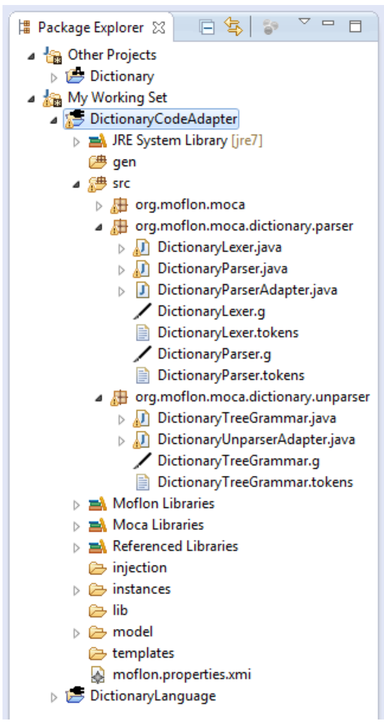


Figure 1.14: Generated stubs and derived files

2 Text-to-tree transformation

Now that our workspace is successfully prepared, let's discuss how the transformation will proceed. For reference, Fig. 2.1 depicts a small sample of the textual syntax that will specify a dictionary instance. As we shall see in a moment, the libraries and shelves containing each dictionary correspond to a folder structure, while the contents for a single dictionary are specified in a file.

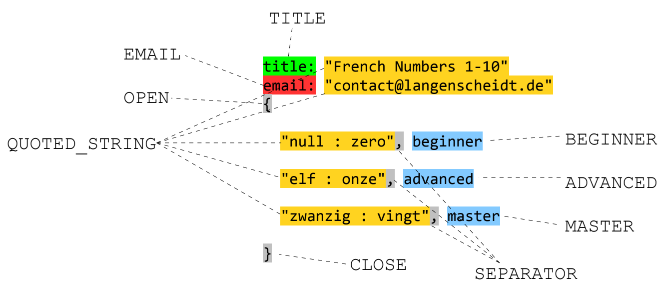
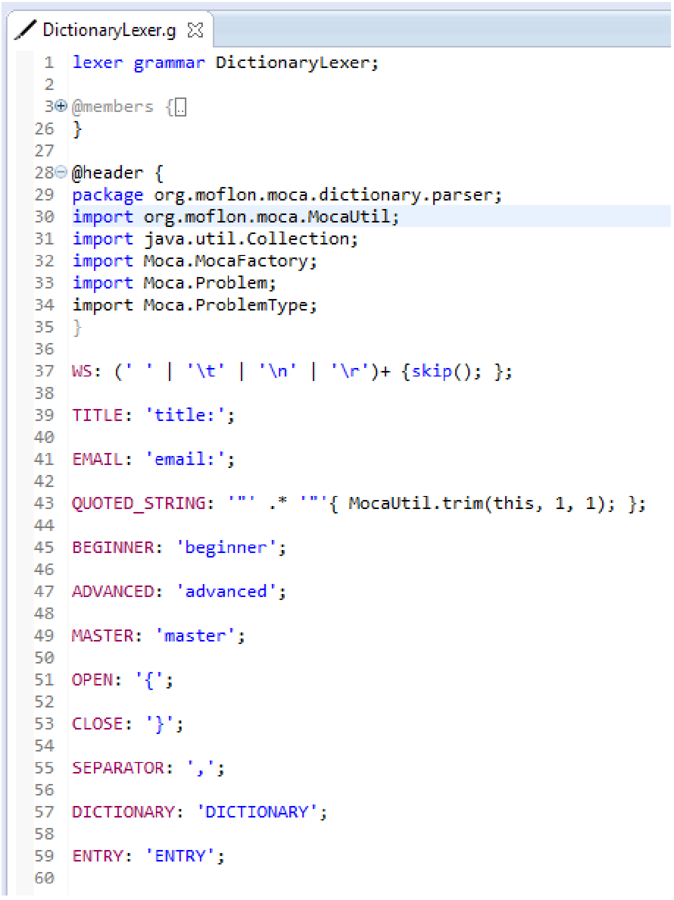


Figure 2.1: Identified tokens in a dictionary file

On the way to an instance model of our dictionary metamodel, the very first step is to create nice *chunks* of characters. This step is called *lexing* and it simplifies the comprehension of the complete text. Interestingly, human beings actually comprehend text in a similar manner; one recognizes whole words without "seeing" every individual character. This is the reason why you can siltl raed tihs sneentce alsomt eforftlsesly. A lexer recognizes these chunks or *tokens* and passes them on as a token stream to the *parser* that does the actual work of recognizing complex hierarchical and recursive structures.

To recognize the tokens as indicated in Fig. 2.1, `ANTLR` can automatically generate a lexer in Java from a compact specification. This is actually a DSL for lexing and is explained in detail in [**?**]. If you are unfamiliar with EBNF, and feel you may have problems understanding the lexer grammar, we suggest going through the documentation on `www.antlr.org`, or reading the relevant chapters in [**?**]. Otherwise, let's complete the *lexer* and *parser* grammars that will handle our project instances.

- ▶ Navigate to "DictionaryCodeAdapter/src/org.moflon.moca.dictionary-.parser" and edit `DictionaryLexer.g` until it matches Fig. 2.2.



```
DictionaryLexer.g
lexer grammar DictionaryLexer;

@members {
}

@header {
package org.moflon.moca.dictionary.parser;
import org.moflon.moca.MocaUtil;
import java.util.Collection;
import Moca.MocaFactory;
import Moca.Problem;
import Moca.ProblemType;
}

WS: (' ' | '\t' | '\n' | '\r')+ {skip(); };

TITLE: 'title:';

EMAIL: 'email:';

QUOTED_STRING: '"' .* '"'{ MocaUtil.trim(this, 1, 1); };

BEGINNER: 'beginner';

ADVANCED: 'advanced';

MASTER: 'master';

OPEN: '{';

CLOSE: '}';

SEPARATOR: ',';

DICTIONARY: 'DICTIONARY';

ENTRY: 'ENTRY';
```

Figure 2.2: Lexer grammar

- Don't forget to add `import org.moflon.moca.MocaUtil` to `@header`. Be vigilant to avoid any typos and mistakes!

- Save to compile the file, and ensure no errors persist before proceeding.

To briefly explain the two complicated-looking rules, note that the `WS` rule simply ignores white space. The '`skip()`' statement throws away the tokens matched as white space each time they're found in a stream. Similarly, `QUOTED_STRING` calls '`MocaUtil.trim(...)`', which trims a recognized token by removing the specified number of characters at its beginning and end. In this case, the token is everything between the '"' characters, as indicated by the '`.*`' symbol.

Now let's establish a parser to form a file's stream of tokens (as created by the lexer) into a *tree*. In this context, a tree is an acyclic, hierarchical, recursive structure as depicted in Fig. 2.3. Depending on what the tree is to be used for, it can be organized differently using extra *structural* nodes such as `DICTIONARY` or `ENTRY` which were not present in the textual syntax. These can be used to give additional semantics to the tree.

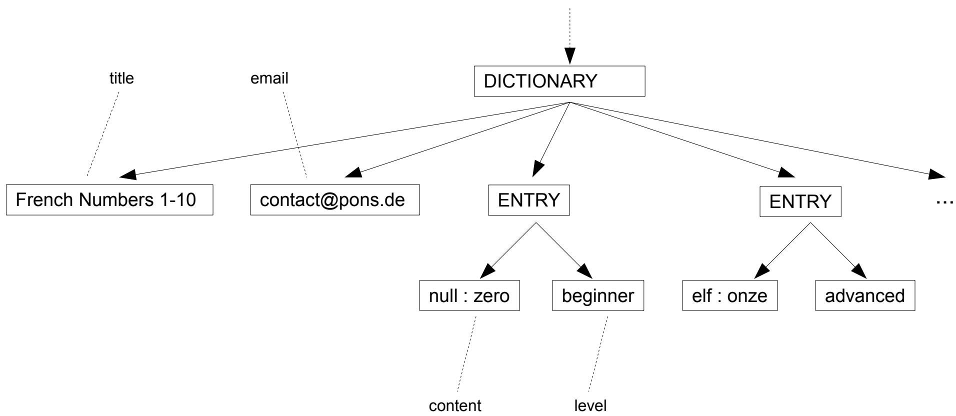


Figure 2.3: Abstract Syntax Tree (AST) of an input token stream

- ▶ From the same package, open and edit `DictionaryParser.g` until it matches Fig. 2.4. As with the lexer, avoid any mistakes, and ensure it compiles before proceeding.

You'll notice that the parser grammar is extremely similar to the lexer grammar, save for some *parser actions* following the `'->'` symbol. These actions control the construction of the resulting tree. Using this simple tree language, one can (1) abstract from tokens such as `'{'` or `'}'`, which are just *syntactical noise*[9] and (2) enrich the tree with structural nodes such as `ENTRY`, which add explicit structure to the tree. Refer to [**?**] and online resources for detailed explanations on the syntax and semantics of the parser grammar supported by `ANTLR`.

[9]Irrelevant content for our model

DictionaryParser.g

```
parser grammar DictionaryParser;

options {
  language = Java;
  tokenVocab = DictionaryLexer;
  output = AST;
}

@members {
}

@header {
package org.moflon.moca.dictionary.parser;
import java.util.Collection;
import Moca.MocaFactory;
import Moca.Problem;
import Moca.ProblemType;
}

// Parser Rules:
main: title email? OPEN entry+ CLOSE -> ^(DICTIONARY title email? entry+);

title: TITLE QUOTED_STRING -> QUOTED_STRING;

email: EMAIL QUOTED_STRING -> QUOTED_STRING;

entry: QUOTED_STRING SEPARATOR level -> ^(ENTRY QUOTED_STRING level);

level: (BEGINNER | ADVANCED | MASTER );

```

Figure 2.4: Parser grammar

▶ Before taking our lexer and parser for a spin, navigate to "src/org.moflon.tie"[10] and open `DictionaryCodeAdapterTrafo.java`. We need to update the file so that it will work with our specific project, so add the highlighted areas in Fig. 2.5 to the file.

```java
import DictionaryCodeAdapter.DictionaryCodeAdapterPackage;
import DictionaryLanguage.DictionaryLanguagePackage;

import MocaTree.Folder;
import java.io.File;

public class DictionaryCodeAdapterTrafo extends IntegratorHelper {

    public DictionaryCodeAdapterTrafo() throws IOException {

    public static void main(String[] args) throws IOException {
        // Set up logging
        BasicConfigurator.configure();

        // Text to tree
        Folder folder = MocaMain.getCodeAdapter().parse(new File("instances/in/myLibrary"));
        eMoflonEMFUtil.saveModel(folder, "instances/fwd.src.xmi");

        // Forward Transformation
        DictionaryCodeAdapterTrafo helper = new DictionaryCodeAdapterTrafo();
        helper.performForward("instances/fwd.src.xmi");

        // Backward Transformation
        helper = new DictionaryCodeAdapterTrafo();
        helper.performBackward("instances/bwd.src.xmi");

        // Tree to text
        MocaMain.getCodeAdapter().unparse("instances/out", (Folder) helper.getSrc());
    }

    public void performForward(String source) {

    public void performBackward(String target) {
}
```

Figure 2.5: Edit `DictionaryCodeAdapterTrafo` to run the transformation

You can see that this main method is essentially the driver for a complete transformation, executing four stages for a forward and backward transformation. In a nutshell, each folder in "instances/in/myLibrary" is taken as the root of a tree, and their folder and file structures will be reflected as a hierarchy of (children) nodes in the tree. For each file, the framework will search for a registered parser that is responsible for the particular file, pass the content onto the parser, then plug in the tree generated from the parser as a single subtree of the corresponding file node in the overall tree.

In this example, the framework uses our parser on `.dictionary` files, the file extension we specified when creating the lexer and parser stubs (Fig. 1.14). Of course, this method (in the generated `parserAdapter`) can be overridden

[10]TIE stands for *Tool Integration Environment*

to register e.g., multiple file extensions, or peek into the actual file content and base its parsing decision on what it finds.

- ▶ To prepare some input for the framework, navigate to "Dictionary-CodeAdapter/instances/in" and create the filesystem depicted in Fig. 2.6.

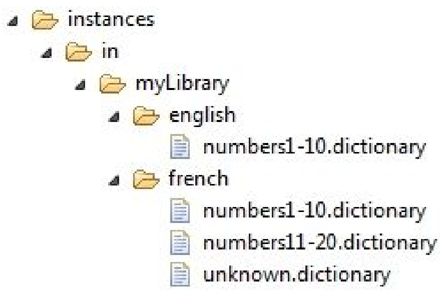


Figure 2.6: Input directory structure

- ▶ Complete each of the four `.dictionary` files with the contents in Table 1.[11] Be vigilant to ensure there are no mistakes with symbols such as colons or commas!

As you can see, the input folder is structured as a single library, `myLibrary`, and split into two languages, english and french, each containing some dictionaries. Reviewing Fig. 2.4, you can see that the structure of these files conforms to the parser's `main` rule: it first lists the dictionary's `title`, may or may not contain an `author`, and contains all `entry` elements between a pair of `OPEN` and `CLOSE` brackets.

[11] If you copy and paste this data, be careful as your .pdf reader may add some invisible characters to the file that ANTLR will not detect and ignore as white space.

english/numbers1-10.dictionary:

```
title: "numbers1-10"
email: "contact@langenscheidt.de"
{
  "null : zero", beginner
  "eins : one", beginner
  "zwei : two", beginner
  "drei : three", beginner
  "vier : four", beginner
  "fuenf : five", beginner
  "sechs : six", beginner
  "sieben : seven", beginner
  "acht : eight", beginner
  "neun : nine", beginner
  "zehn : ten", beginner
}
```

french/numbers1-10.dictionary:

```
title: "numbers1-10"
email: "contact@pons.de"
{
  "null : zero", beginner
  "eins : un/une", beginner
  "zwei : deux", beginner
  "drei : trois", beginner
  "vier : quatre", beginner
  "fuenf : cinq", beginner
  "sechs : six", beginner
  "sieben : sept", beginner
  "acht : huit", beginner
  "neun : neuf", beginner
  "zehn : dix", beginner
}
```

french/numbers11-20.dictionary:

```
title: "numbers11-20"
email: "contact@pons.de"
{
  "elf : onze", advanced
  "zwoelf : douze", advanced
  "dreizehn : treize", advanced
  "vierzehn : quatorze", advanced
  "fuenfzehn : quinze", advanced
  "sechzehn : seize", master
  "siebzehn : dix-sept", master
  "achtzehn : dix-huit", master
  "neunzehn : dix-neuf", master
  "zwanzig : vingt", master
}
```

french/unknown.dictionary:

```
title: "unknown"
{
"unbekannt : unknown", beginner
}
```

Table 1: Four input `.dictionary` files

- Once you have saved each file, right click on `DictionaryCodeAdapterTrafo.java` and navigate to “Run As/Java Application” to run the transformation. Don’t worry about the error messages – they’re related to the unparser which we haven’t implemented yet; You should have received at least one success message indicating your transformation worked.

- Refresh the `instances` folder. Despite being (mostly) unimplemented, the transformation still partly completed, generating several files in the process (Fig. 2.7).

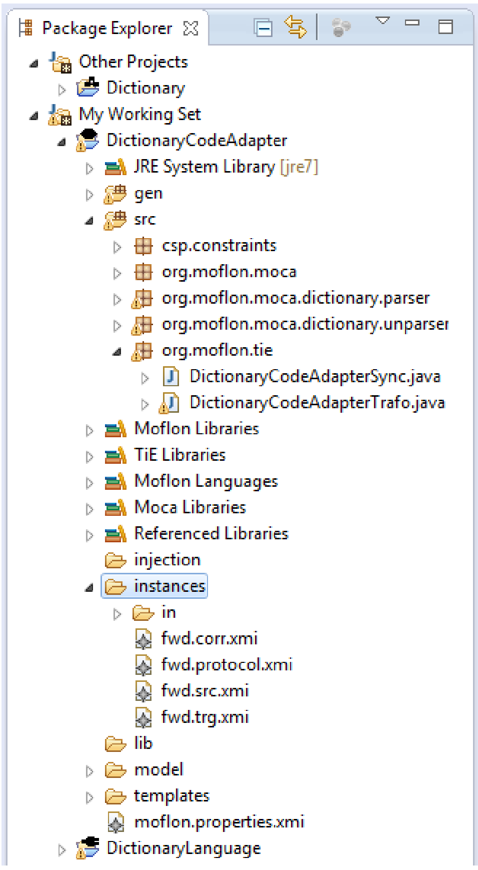


Figure 2.7: Result of the first TGG execution

Let’s go over what each of these files are. First, `fwd.src.xmi` is the direct result of the `myLibrary` filesystem input, which was parsed into a `MocaTree` instance by our ANTLR parser.

While the parser is the only implemented piece of our transformation, TGGMain still used `fwd.src.xmi` in a forward transform, producing the correspondence model `fwd.corr.xmi` (paired with `fwd.protocol.xmi`), and the (currently empty) `Dictionary` target result, `fwd.trg.xmi`.

► Open `fwd.src.xmi` and compare the contents to Fig. 2.8. Reflect on the directory-type structure of the tree, where each `File` and its contents appear as `Nodes`.[12] This file is important to understand! The filesystem was transformed into a corresponding hierarchy of `Folders` and `Files`. The actual *text* content of each file is then transformed to a subtree using a registered, suitable parser. The resulting subtree from the parser is then connected to the existing tree by setting its `DICTIONARY` root as the single child node of a `File`.

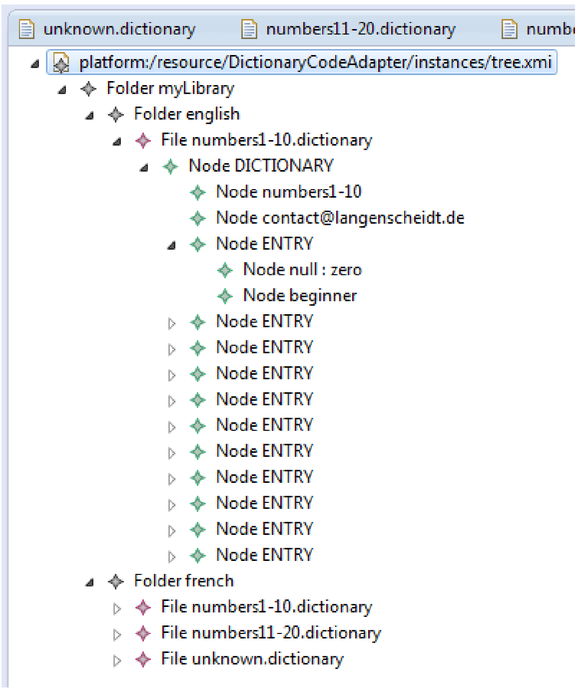


Figure 2.8: A `MocaTree` created by the framework using our parser

If everything executed without errors, well done! Let's continue with the transformation to a `Dictionary` instance by specifiying some TGG rules.

[12]Refer to Fig 1.4 for the metamodel of this structure

3 Tree-to-model transformation with TGGs

Our goal in this section is to break down the `MocaTree` to `Dictionary` transformation into smaller, modular steps. More precisely, we want separate rules for transforming a `Folder` into its appropriate container element (i.e., `Library` or `Shelf`), then individual rules to handle whatever `File` and `Node` elements they contain.

Let's briefly look at the models we'll be working with. We start with Fig. 3.1,[13] where our root input folder, `myLibrary`, contains two subfolders with at least one dictionary `File` each. Each dictionary has one equivalent dictionary root `Node` with at least two children representing the title and first `ENTRY`, along with an unknown number of additional nodes. Of the remaining nodes, there may be one that stores the dictionary author's contact information. All the rest will be `ENTRY` nodes with two children representing its content and level information.

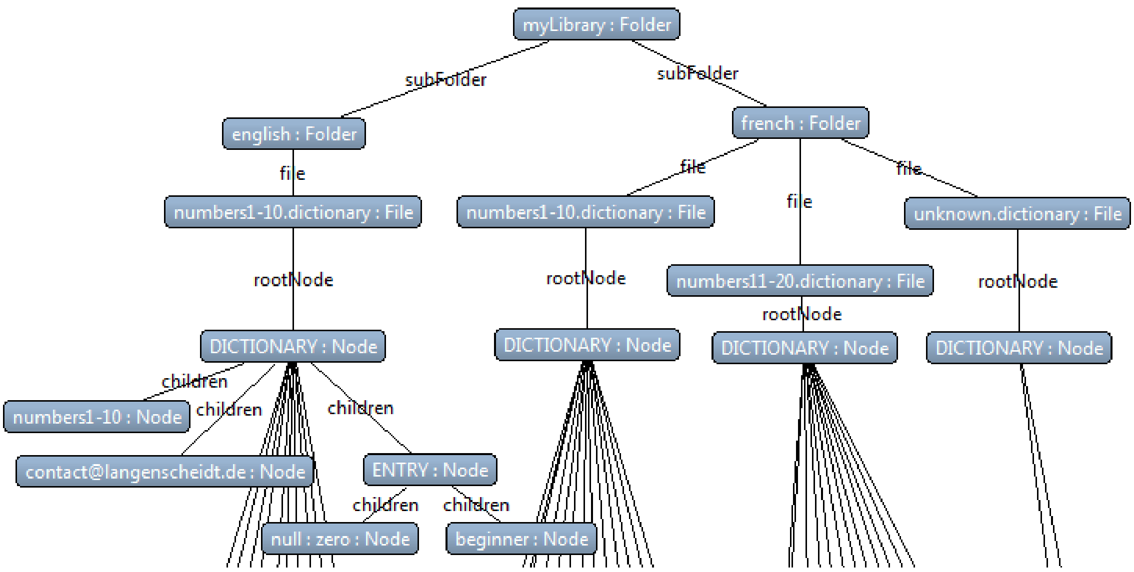


Figure 3.1: The `MocaTree` model of our input directory

[13]You can view this model in your Eclipse editor by placing the contents of `fwd.src.xmi` into eMoflon's Graph Viewer, as introduced in Part II, Section 4.

Our transformation intends to finish with a `Dictionary` model resembling Fig. 3.2, where the root `myLibrary` has four children, one for each shelf and author. These elements will likely pair up, sharing a child `Dictionary` element containing an unknown number of entries.

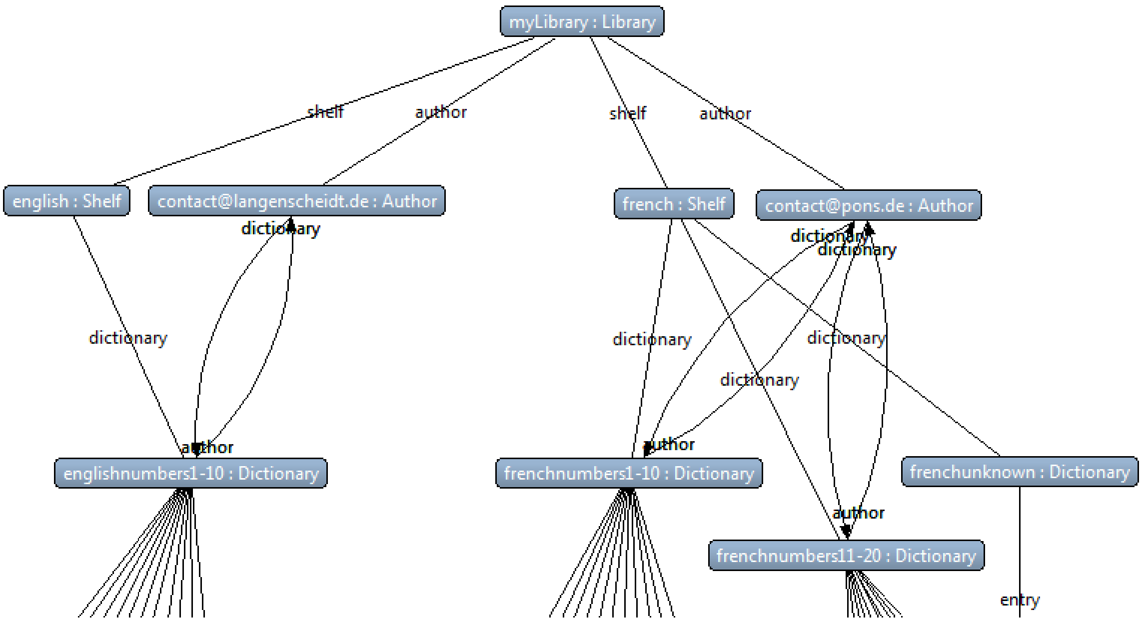


Figure 3.2: The final `Dictionary` target model

As you can see, it's important to keep in mind the flexibility that the rules for this transformation will require. While our example model is small enough to count the number of entries our rules will need to account for, future models may of course vary. Just like SDM patterns, it's key to avoid situation-specific TGG rules.

▷ Next [visual]
▷ Next [textual]

3.1 The visual transformation rules

As a quick note before you start, remember that we have assumed a basic understanding of TGGs and the different ways of using EA productively to create rules. If you find this section challenging, we recommend first working through Part IV to cover TGG fundamentals.

FolderToLibraryRule

- Return to your open project in EA and expand the `<<Rules Package>>`, then open the `Rules` diagram. Create a new rule named `FolderToLibraryRule` and double-click its element to open the rule diagram. Complete it as depicted in Fig. 3.3.

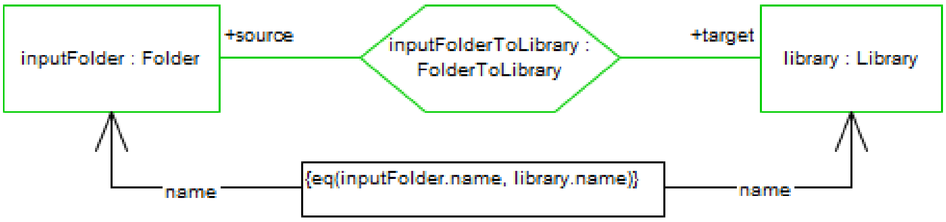


Figure 3.3: `FolderToLibraryRule`

- This is a simple rule that creates and connects equivalent `Folder` and `Library` instances. We're able to use this entire rule as context for the next rule, which will handle the creation of shelves. Select `inputFolder`, `inputFolderToLibrary`, and `library`, then use the eMoflon control panel to `derive` a new rule (Fig. 3.4). Name this `ForAllShelfRule`.

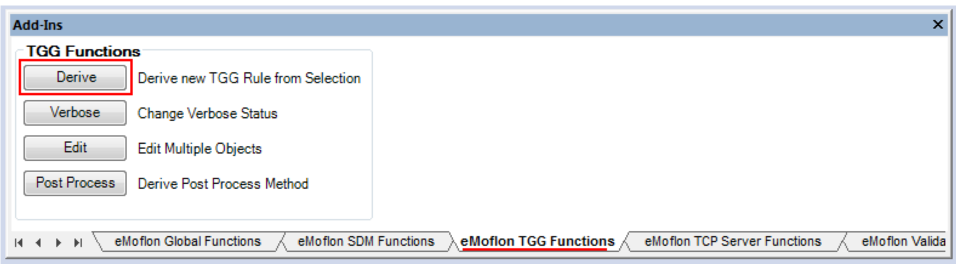


Figure 3.4: Deriving a new rule with eMoflon's control panel

ForAllShelfRule

The derivation procedure will open a new diagram with context elements from the first rule. This new rule is similar to `FolderToLibraryRule`, except that it will connect new (green) elements to existing (black) containers.

▶ Complete the rule as depicted in Fig. 3.5. You'll need to create a new `FolderToShelf` correspondence type in either the schema (as we did in the beginning), or on-the-fly by selecting `Create New Correspondence Type` in the quick-link dialogue (Fig. 3.6).

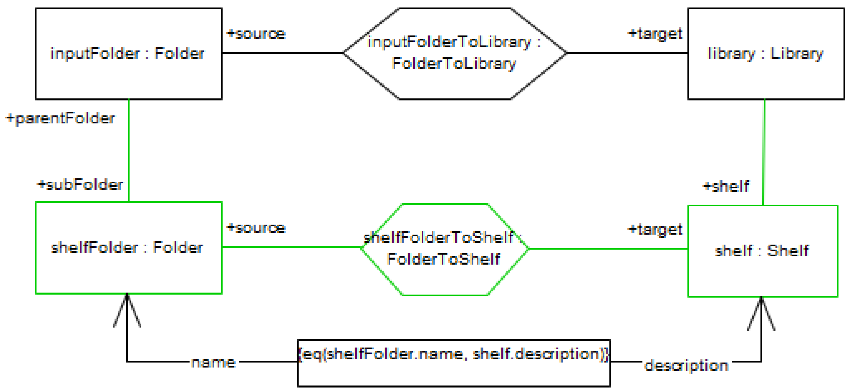


Figure 3.5: `ForAllShelfRule`

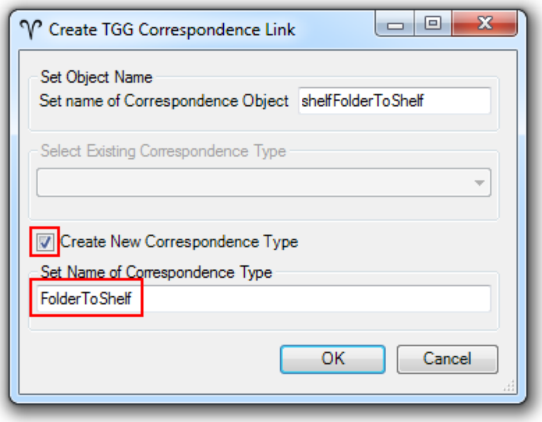


Figure 3.6: Creating a `FolderToShelf` correspondence type on-the-fly

NodeToDictionaryRule

▶ With the container `shelf` now assumed to exist, we are ready to handle dictionary `File` elements. Analogously to how you began the previous rule, select `shelfFolder`, `shelfFolderToShelf`, and `shelf`, and derive `NodeToDictionaryRule` with this context.

▶ Complete it as depicted in Fig. 3.7. As you can see, this rule creates a `dictionaryNode` and and its equivalent `dictionary`, and handles the first node in the tree structure. Nearly every element is used to correctly set the `dictionary` and `dictionaryFile` names in two constraints.

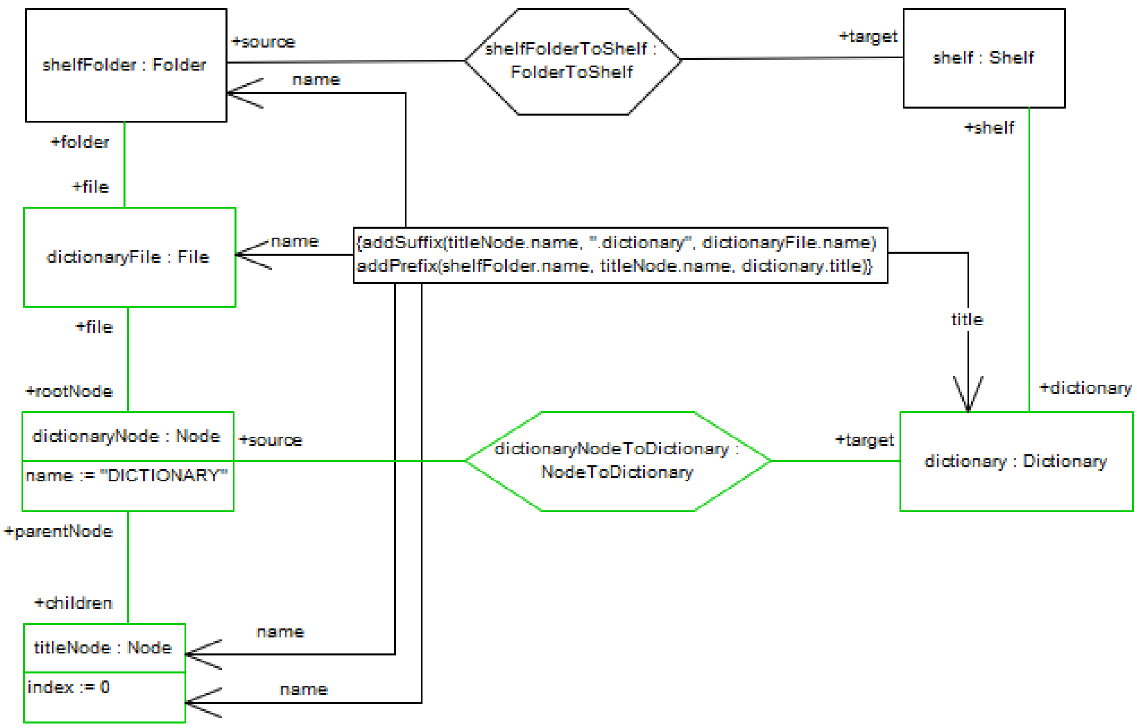


Figure 3.7: `NodeToDictionaryRule`

▶ Please note that the attribute constraint in `titleNode` is required in order to ensure that the node with the title information is always the first child in the tree (index = 0).

- Note that we could have also included another `node` here to handle the author, along with a third, fourth, or even tenth `node` for a dictionary's entries, but that would mean the pattern would absolutely have to match to a single author and ten entry elements, which may not always exist. Instead, we'll create separate rules for each of these which can be called as many (or as few) times as necessary.

ForAllEntryRule

- Let's handle `entry nodes` first. Create and complete `ForAllEntryRule` as depicted in Fig. 3.8.

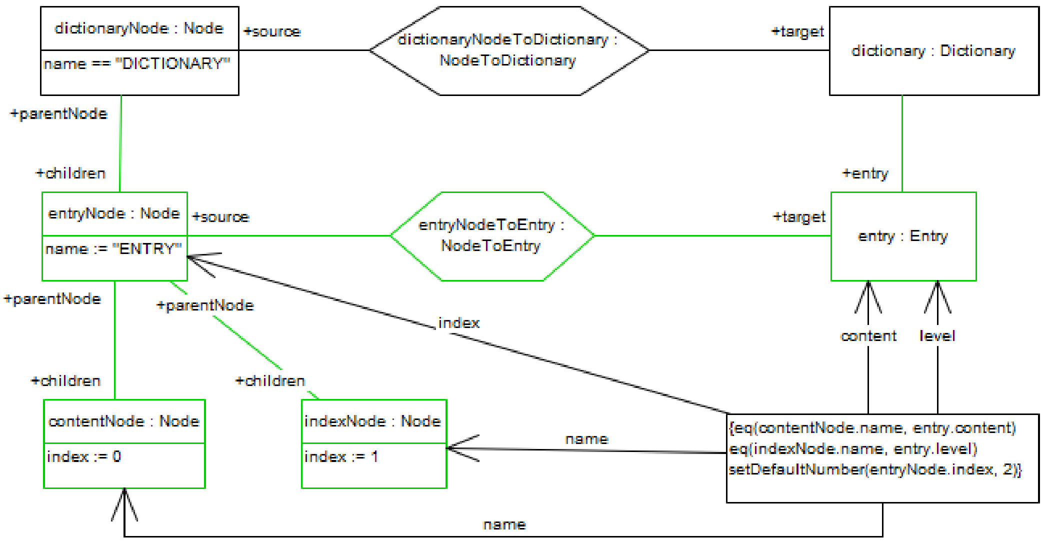


Figure 3.8: `ForAllEntryRule`

You can see that for every `entryNode`, `contentNode` and `indexNode` child elements are also created. When transforming from a tree to dictionary, these are identified by their 0 and 1 indices. As such, the rule's first two constraints are used to ensure that this information is not lost, guaranteeing their correct positions in the tree when transforming back.

The final constraint however, is one we haven't used before. If you re-examine your source `tree.xmi` model, you'll notice that every entry has a different `index` value. This prevented us from setting an attribute constraint on `entryNode` but, as long as its index wasn't 0 indicating a `titleNode` (as constrained in the previous `NodeToDictionaryRule`), it didn't matter. Unfortunately, this missing information means any new `entryNodes` created in the backward transformation have a default 0 index value, and *could* be mistaken by the rule. By using `setDefaultNumber`, we have declared that any created default `index` attributes must be set to 2.

AuthorRule

- We want to create a rule to handle authors next, so double-click the anchor in the top left of the diagram to return to `NodeToDictionaryRule`.
- We can begin this rule by deriving from `DictionaryNode`, `dictionary`, and `shelf`, but we'll need to add a fourth context element, `library`, in accordance with the `Dictionary` metamodel, where each `author` is connected to its `dictionary` and the `library` instance. Derive and create `AuthorRule` as shown in Fig. 3.9.

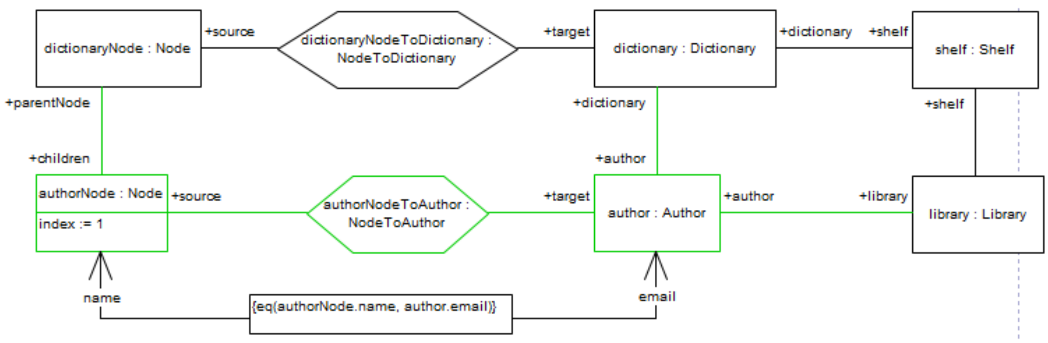


Figure 3.9: `AuthorRule`

Handling all `author` instances however, can't be realised with a single rule. Here we have specified that, for every `authorNode` the rule finds, an `author` instance should be created. This would be fine if we had unique authors for *every* dictionary `File`, but if you take look at both of the french `numbers` files, you'll notice they both have the same contact information. This means that our `Dictionary` will have two identical `author` instances for one `library`.

Some users may be okay with this, and not care about redundant information so long as all the correct information is there, but others may prefer a more concise structure. How can we refine this rule so that it's easy to handle both cases?

eMoflon's visual syntax has a cool *refinement* feature which enables you to adjust specific elements in a rule, without having to redraw an entire diagram exactly as before, save for one or two minor differences. Given that we want rules to handle either *always* creating an `author`, or checking for an existing one first, (both which will be identical to `AuthorRule` except for the binding and reference links on the matched `authorNode`), this feature is exactly what we need.

- ▶ Return to the `Rules` diagram. Since we're no longer implementing `AuthorRule` directly, we need to make it `abstract`. Select the rule, then hit `alt + enter` to open its properties dialogue.
- ▶ Switch to the `Details` tab, and select `Abstract` from the list of properties (Fig. 3.10). Affirm and close by pressing `OK`.

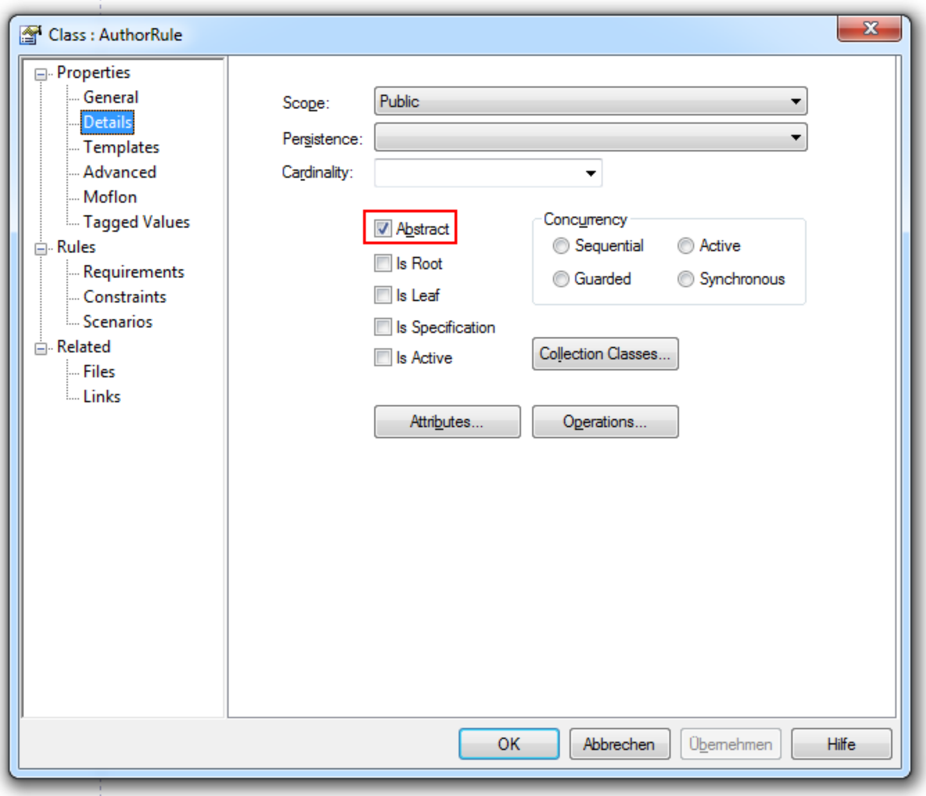


Figure 3.10: Declaring `AuthorRule` as abstract

Now we can develop our two rules. The key idea when building refinements is to imagine the new rules being placed directly over the pattern they inherit from, similar to a transparency sheet. These rules will execute `AuthorRule` exactly, except for whatever modifications you make (which "cover" the original element).

Let's make the rule that handles an already existing author first. Inspecting `AuthorRule`, we still want the rule to match a new `authorNode` and create a link between `author` and `dictionary`, but the `author` element, and the link connecting it to `library` should already exist, (i.e., be 'black').

ExistingAuthorRule

- In `AuthorRule`'s diagram, select `author` and `library`, and press `Derive`. Enter `ExistingAuthorRule` as the rule's name but given that we want to refine the selected elements, not use them as context elements, be sure to select the `exact copy` option (Fig. 3.11).

Figure 3.11: Deriving a refinement rule

- The rule diagram will open in the editor, with the elements in the same place you copied them from. Complete the rule by changing `author` and the link to `library` to `Check Only` as depicted in Fig. 3.12.

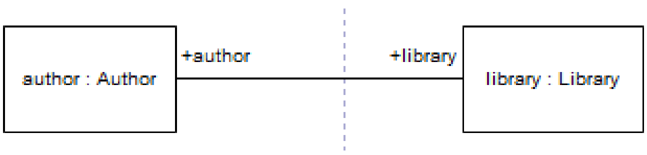


Figure 3.12: `ExistingAuthorRule`

- If you're having difficulty visualising the entire rule with this minor modification, note that EA allows you to drag and drop all elements from the basis rule as links so they're represented in the diagram. Figure 3.13 represents the entire `ExistingAuthorRule`. This how the rule would have looked without using rule refinement.

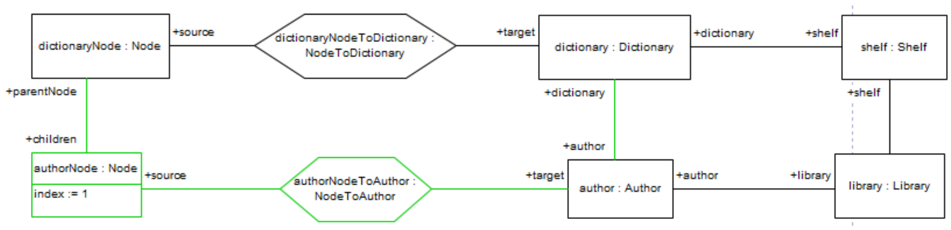


Figure 3.13: `ExistingAuthorRule` (flattened)

NewAuthorRule

- Return to `AuthorRule` and derive this time a copy of `authorNode` and `library` into a rule called `NewAuthorRule`. We'll explain why further in the next section, but for now just leave it as it is, and don't change anything. As we have defined `AuthorRule` as abstract, we need this concrete rule to handle new authors. The diagram should come to resemble Fig. 3.14.

Figure 3.14: Completed `NewAuthorRule`

- Of course, we could have left `AuthorRule` concrete and used just two rules, but having an explicit abstract rule, with two concrete implementations of the possibilities, is clearer.

- Return to the `Rules` diagram one last time. In order to ensure the new rules refine `AuthorRule`, quick-link from each to the root rule, choosing `Create Refinement Link` from the context menu. Your diagram should now resemble Fig. 3.15.

- You're nearly done! Make sure everything is saved, and validate your TGG.

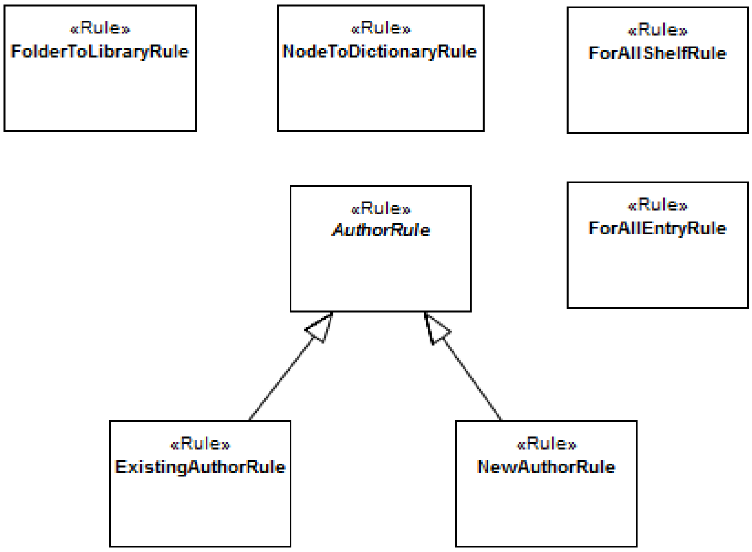


Figure 3.15: Final `Rules` diagram

3.2 The textual transformation rules

Please note that we have assumed a basic understanding of TGGs, specifically those about constructing each scope in a rule, and how to use Eclipse efficiently with eMoflon's auto-completion feature. If you find this section challenging, we recommend first working through Part IV to cover TGG fundamentals.

FolderToLibraryRule

- ▶ Return and expand the `DictionaryCodeAdapter` TGG package and right-click on the `Rules` folder. Create your first rule by navigating to "New/TGG Rule," naming it `FolderToLibraryRule`.
- ▶ All this rule needs to do is create a `Folder` (i.e., "myLibrary") together with its equivalent `Library`, and establish a correspondence link between them. Edit your rule until it resembles Fig. 3.16.

schema.sch | FolderToLibraryRule.tgg

```
rule FolderToLibraryRule {
    source {
        ++ inputFolder : Folder
    }

    correspondence {
        ++ inputFolder <- folderToLibrary : FolderToLibrary -> library
    }

    target {
        ++ library : Library

    }

    constraints {[
        eq(inputFolder.name, library.name)
    ]}
}
```

Figure 3.16: The TGG transformation begins with this rule.

ForAllShelfRule

- ▶ Let's use some elements from the previous rule to help us define how to handle creating shelves for our library. Copy and paste the required context elements from `FolderToLibraryRule` in a new `ForAllShelfRule`, adding a new `shelfFolder` and `shelf` as depicted in Fig. 3.17.

```
schema.sch    FolderToLibraryRule.tgg    ForAllShelfRule.tgg
rule ForAllShelfRule {
    source {
        inputFolder : Folder {
            ++ -subFolder-> shelfFolder
        }

        ++ shelfFolder : Folder
    }

    correspondence {
        inputFolder <- folderToLibrary : FolderToLibrary -> library
        ++ shelfFolder <- folderToShelf : FolderToShelf -> shelf
    }

    target {
        library : Library {
            ++ -shelf-> shelf
        }

        ++ shelf : Shelf

    }

    constraints {[
        eq(shelfFolder.name, shelf.description)
    ]}
}
```

Figure 3.17: `ForAllShelfRule`

▶ Add the new correspondence type to your `schema` (Fig. 3.18).

```
schema.sch    FolderToLibraryRule.tgg    ForAllShelfRule.tgg
source /MocaTree
target /DictionaryLanguage

// Add your classes here (template available)
class FolderToLibrary {
    source -> Folder
    target -> Library
}

class FolderToShelf {
    source -> Folder
    target -> Shelf
}
```

Figure 3.18: Updated TGG `schema`

Now that we can assume the primary library and shelf containers exist, we can handle the dictionary `File` elements. We know from our generated tree model that a dictionary will always have a title node, but we're unsure if an author will be included, and there's no way to know how many entries are involved. Therefore, we should create at least three different rules to handle this stage of the transformation.

NodeToDictionaryRule

▶ Create a rule named `NodeToDictionaryRule` as indicated in Fig. 3.19.

schema.sch NodeToDictionaryRule.tgg

```
rule NodeToDictionaryRule {
    source {

        shelfFolder : Folder

        ++ dictionaryFile : File {
            ++ -folder-> shelfFolder
            ++ -rootNode-> dictionaryNode
        }

        ++ dictionaryNode : Node {
            dictionaryNode.name := "DICTIONARY"
            ++ -children-> titleNode
        }

        ++ titleNode : Node {
            titleNode.index := 0
        }
    }

    correspondence {
        shelfFolder <- folderToShelf : FolderToShelf -> shelf
        ++ dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
    }

    target {
        shelf : Shelf

        ++ dictionary : Dictionary {
            ++ -shelf-> shelf
        }

    }

    constraints {[
        addSuffix(titleNode.name, ".dictionary", dictionaryFile.name)
        addPrefix(shelfFolder.name, titleNode.name, dictionary.title)
    ]}
}
```

Figure 3.19: `NodeToDictionaryRule` handling only `titleNodes`

- As you can see, this rule demands that a `shelfFolder` and `shelf` already exist before executing, implying that this rule can only be called after executing `ForAllShelfRule`. An attribute constraint is used with `titleNode` to ensure that the correct child `Node` is matched from `dictionaryNode`, and not accidentally to an author or entry node, which will have different indices.

- This rule also imposes two constraints for attribute manipulation. We need to add the name of the shelf as a prefix to the title node's name to get the dictionary's title (i.e., "english" + "numbers1-10"). Similarly, the second constraint appends `.dictionary` to `titleNode.name` to get the file name of the dictionary.

ForAllEntryRule

- Let's handle `Entry` elements next. Create `ForAllEntryRule` so that it closely resembles Fig. 3.20.

You can see that this rule has three attribute constraints, one of which we haven't encountered before. The first two `eq` constraints guarantee that an `entryNode`'s content and index values remain consistent with its equivalent `entry` in a `dictionary`. The final constraint is to ensure that any new `entryNodes` created in the backward transformation have index values set to 2.

Without this final constraint, all new `entryNodes` would have a default 0 index value, and could be mistaken as `titleNodes` as described in the previous `NodeToDictionaryRule`, causing the entire transformation to fail.

```
rule ForAllEntryRule {
    source {
        dictionaryNode : Node {
            ++ -children-> entryNode
        }

        ++ entryNode : Node {
            entryNode.name := "ENTRY"
            ++ -children-> contentNode
            ++ -children-> indexNode
        }

        ++ contentNode : Node {
            contentNode.index := 0
        }

        ++ indexNode : Node {
            indexNode.index := 1
        }
    }

    correspondence {
        dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
        ++ entryNode <- nodeToEntry : NodeToEntry -> entry
    }

    target {
        dictionary : Dictionary {
            ++ -entry-> entry
        }

        ++ entry : Entry

    }

    constraints {[
        eq(contentNode.name, entry.content)
        eq(indexNode.name, entry.level)
        setDefaultNumber(entryNode.index, 2)
    ]}
}
```

Figure 3.20: `ForAllEntryRule`

The last thing we need to specify is how to handle `authors`. Transforming an `authorNode` to an `author` isn't as simple as an `entryNode`, where you create an `entry` every time. Instead, we have to account for the possibility of a single author for multiple dictionaries in a `Library`. While some users may not care about having redundant information, why not also provide a rule for users who want to enforce unique authors in a `Library`?

NewAuthorRule

▶ Create `NewAuthorRule`, and complete it as depicted in Fig. 3.21. This is a one-to-one correspondence rule, where every `authorNode` creates a new `author`. If this rule is used in a transformation, one might end up with multiple authors with the same email address.

```
rule NewAuthorRule {
    source {
        dictionaryNode : Node {
            ++ -children-> authorNode
        }

        ++ authorNode : Node {
            authorNode.index := 1
        }
    }

    correspondence {
        dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
        ++ authorNode <- authorNodeToAuthor : NodeToAuthor -> author
    }

    target {
        dictionary : Dictionary {
            -shelf-> shelf
        }

        shelf :  Shelf

        library : Library {
            -shelf-> shelf
        }

        ++ author : Author {
            ++ -dictionary-> dictionary
            ++ -library-> library
        }

    }

    constraints {[
        eq(authorNode.name, author.email)
    ]}
}
```

Figure 3.21: Creating new authors in `NewAuthorRule`

ExistingAuthorRule

▶ Similarly, create `ExistingAuthorRule` as specified in Fig. 3.22. You should be able to copy and paste the majority of the previous rule. In fact, the only thing you need to change are two small characters in front of `author` and its `dictionary` reference, forcing the rule to find an existing `author`, if possible.

schema.sch | NewAuthorRule.tgg | ExistingAuthorRule.tgg

```
rule ExistingAuthorRule {
    source {
        dictionaryNode : Node {
            ++ -children-> authorNode
        }

        ++ authorNode : Node {
            authorNode.index := 1
        }
    }

    correspondence {
        dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
        ++ authorNode <- authorNodeToAuthor : NodeToAuthor -> author
    }

    target {
        dictionary : Dictionary {
            -shelf-> shelf
        }

        shelf :  Shelf

        library : Library {
            -shelf-> shelf
        }

        author : Author {
            ++ -dictionary-> dictionary
            -library-> library
        }

    }

    constraints {[
        eq(authorNode.name, author.email)
    ]}
}
```

Figure 3.22: Checking for existing authors in `ExistingAuthorRule`

► Great work! You have now specified five different rules to handle a bidirectional text-to-model transformation! For confirmation, your final schema and package explorer should now resemble Fig. 3.23.

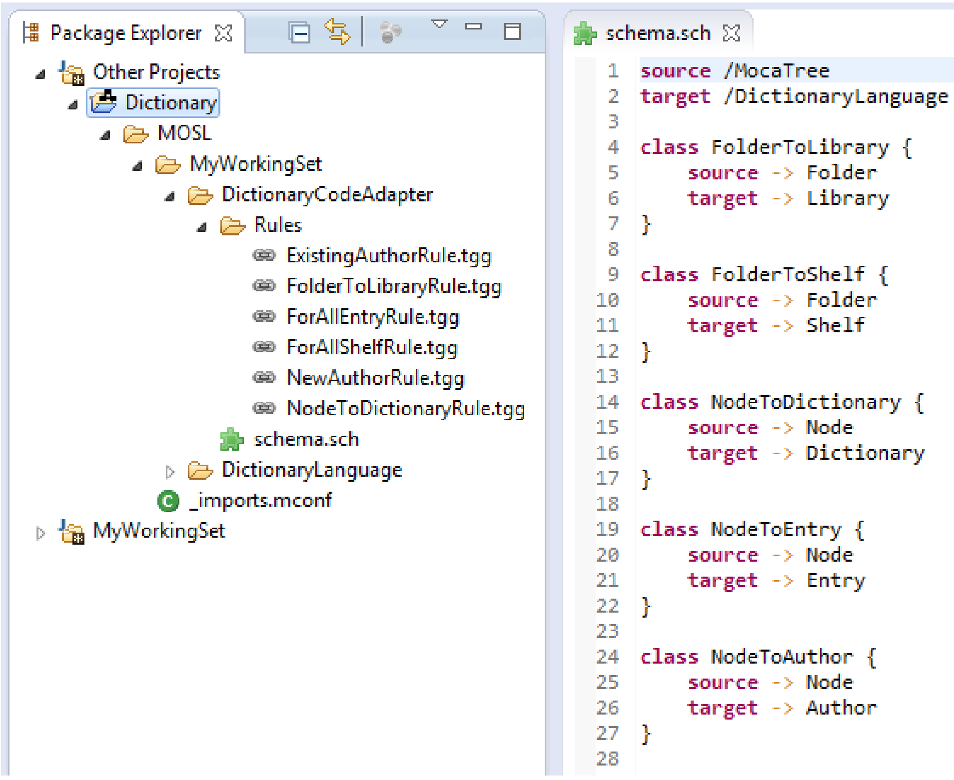


Figure 3.23: Your final `rules` project structure and `schema`

► Given that everything has been done correctly, and MOSL hasn't reported any errors, build your TGG transformation. If problems arise, be sure to double-check your files for spelling or other mistakes.

3.3 Additional author handling

▶ If your project's build succeeded, run the transformation[14] again and examine the successful forward output, `fwd.trg.xmi`, a little closer (Fig. 3.24).

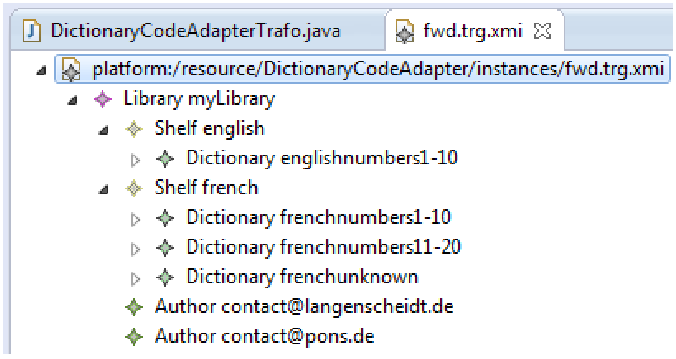


Figure 3.24: `Dictionary` result of the forward transformation

▶ Your output may or may not resemble ours. In fact, there's a 50/50 chance that either a single or two `contact@pons.de` authors are created! At run-time, the transformation has a choice between two rules to apply to a matched `authorNode`. The resulting choice is entirely random, meaning that your output is likely to be different each time you run the transformation. For a deterministic transformation, you would need to force a preferred decision.

There are two ways to do this:

1. At run-time, allowing users to decide for themselves what they would prefer to use, or
2. At design-time, making the decision a part of the TGG specification.

[14]If you haven't already, read Part IV, Section 6 for details on to run a transformation

Option 1: Run-time decision

The advantage with this option is that you give users the choice of what they prefer. Some users don't mind having multiple authors, while others might prefer a minimalist design. They can easily change their preference possibly on a case-by-case basis, by implementing a TGG rule *configurator*.

- Implement and set the configurator to be used for the transformation as depicted in Fig. 3.25. Note how the possible alternatives are filtered using an `isRule` predicate to compare the name of each alternative with the preferred rule `NewAuthorRule` in this case. As this degree of freedom concerning the creation or possible reuse of authors only arises in the forward direction, we do not need a similar configurator for the backward transformation.

```java
import org.moflon.tgg.algorithm.configuration.Configurator;
import org.moflon.tgg.algorithm.configuration.RuleResult;

public class DictionaryCodeAdapterTrafo extends SynchronizationHelper {
    public DictionaryCodeAdapterTrafo() throws IOException {
    public static void main(String[] args) throws IOException {

    public void performForward(String source) {
        try {
            loadSrc(source);
        } catch (IllegalArgumentException iae) {
            System.err.println("Unable to load " + source + ", " + iae.getMessage());
            return;
        }

        setConfigurator(new Configurator() {
            @Override
            public RuleResult chooseOne(Collection<RuleResult> alternatives) {
                Optional<RuleResult> preferred = alternatives.stream()
                        .filter(rr -> rr.isRule("NewAuthorRule"))
                        .findAny();
                return preferred.orElse(Configurator.super.chooseOne(alternatives));
            }
        });

        integrateForward();

        saveTrg("instances/fwd.trg.xmi");
        saveCorr("instances/fwd.corr.xmi");
        saveSynchronizationProtocol("instances/fwd.protocol.xmi");

        System.out.println("Completed forward transformation!");
    }

    public void performBackward(String target) {
}
```

Figure 3.25: Setting the configurator to control the run-time decision

- Save and run the transformation a few times, using the integrator to confirm your preference is enforced each time. You should now *always* get two `contact@pons.de` authors using this configurator. Try to change your preference and see the effect!

Option 2: Design-time decision

It is also possible to set a preference as part of the actual design of the transformation – users will not be able to modify this. In our example, this preference can be enforced using a NAC which checks to see if there is already an author with the same email in the library.

- ▶ Open and update either `NewAuthorRule` (visual) as shown in Fig. 3.26 or edit the target domain in Eclipse (textual) as depicted in Fig. 3.27.

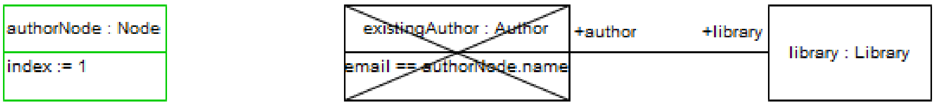


Figure 3.26: Adjust `NewAuthorRule` by adding a NAC

```
target {
    dictionary : Dictionary {
        -shelf-> shelf
    }

    shelf :  Shelf

    library : Library {
        -shelf-> shelf
    }

    ! existingAuthor : Author {
        author.email == authorNode.name
        -library-> libary
    }

    ++ author : Author {
        ++ -dictionary-> dictionary
        ++ -library-> library
    }

}
```

Figure 3.27: Add a NAC to `NewAuthorRule`

- ▶ Save and rebuild the TGG, then run the transformation a few times. Confirm your preference is enforced each time – With this NAC, the configurator won't be given the chance to decide anymore!

4 Tree-to-text transformation

We've finally reached the last step, transforming our tree result, `fwd.trg.xmi` back into a filesystem with `.dictionary` files identical to our original input, the `myLibrary` filesystem.

Note that in an actual application, we would do something useful with the model before transforming it back to text, or the dictionary might have been produced from a learning box, i.e., the textual syntax representation wouldn't exist yet. One of the coolest things about ANTLR is that the same parsing technology that we used in Section 2 can be used to *unparse* the tree.

Analogously to parsing text with a lexer and parser grammar to produce a tree, a tree is unparsed to text using a *tree grammar* and *templates*. A tree grammar is similar to EBNF, consisting of rules (`main`, `entry`) that each match a tree fragment and evaluate a template, as opposed to rules that match text fragments and build a tree. For further details concerning tree grammars, we refer to [**?**] and the ANTLR website `www.antlr.org`.

- ▶ Expand "src/org.moflon.moca.dictionary.unparser", open `DictionaryTreeGrammar.g`, and edit the contents as depicted in Fig. 4.1.

- ▶ Next, open `DictionaryUnparserAdapter.java` (Fig 4.2). You'll notice that this file contains a (commented out) `StringTemplateGroup` method for retrieving a group of templates and needs to be implemented. The comments explain how to use either a folder containing different template files, or a single file containing all templates. The latter is better for numerous small templates, while the former makes sense when the templates contain a lot of static text.

- ▶ For this small example, a single file with all templates is ideal. Uncomment line 44 (the option for a group file) and remove the line throwing an `UnsupportedOperationException`.

DictionaryTreeGrammar.g

```
tree grammar DictionaryTreeGrammar;

options {
  ASTLabelType = CommonTree;
  output       = template;
}

// Tokens used internally by Moca
// ID: ('a'..'z' | 'A'..'Z')+;
// STRING: ( ID | ('0'..'9') )+;
// ATRIBUTE: Used as an imaginary...

tokens {
  ID;
  STRING;
  ATTRIBUTE;
}

@members {
}

@header {
}

// tree grammar rules:
main: ^('DICTIONARY' name=STRING author=STRING? entries+=entry+)
-> dictionary(name={$name}, author={$author}, entries={$entries});

entry: ^('ENTRY' entryLabel=STRING level=STRING)
-> entry(entry={$entryLabel}, level={$level});

```

Figure 4.1: Tree grammar for the unparser

```
@Override
protected StringTemplateGroup getStringTemplateGroup() throws FileNotFoundException
{
  //TODO provide StringTemplateGroup ...
  // ... from folder "dictionary" containing .st files
  // return new StringTemplateGroup("dictionary", "templates/dictionary");
  // ... from group file Dictionary.stg
  // return new StringTemplateGroup(new FileReader(new File("./templates/Dictionary.stg")));
  throw new UnsupportedOperationException("Creation of StringTemplateGroup not implemented yet ...");
}
```

Figure 4.2: Two options of how to store templates

- Create a template file by navigating to the empty “templates” folder of your adapter project, and creating a new file named `Dictionary.stg` (as demanded in `DictionaryUnparserAdapter.java`). Complete it as specified in Fig. 4.3.

Dictionary.stg

```
group dictionary;

dictionary(name, author, entries) ::= <<
title: "<name>"
<if(author)>email: "<author>"<endif>
{
    <entries; separator="\n">
}
>>

entry(entry, level) ::= <<
"<entry>", <level>
>>
```

Figure 4.3: The `dictionary` template

- Copy and paste `fwd.trg.xmi` into “instances,” naming the new file `bwd.src.xmi`. This will be the backward transformation’s input file.

- Save and run your transformation again – there should no longer be an error message in the console. Inspect and compare your input and output folders and their containing files (Fig. 4.4). Are they the same? As we abstracted from some aspects such as sorting all entries in the transformation, the generated files are probably not absolutely identical. If this is required (we assume for the example that it is not) then the generated tree can either be *normalized* as required before generating text, or the additional aspect of order can be modelled explicitly in the transformation (using indices in the tree and `next` edges in the model).

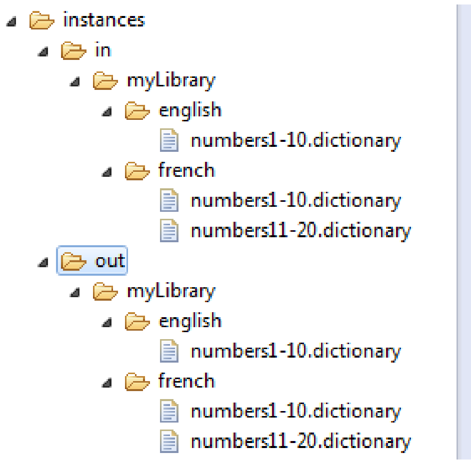


Figure 4.4: The final input and output filesystems

► If everything succeeded, your transformation is now complete in both directions! Feel free to play around with changing some files such as a the unparser template, or the content of the original files. How are the changes propagated through the transformation? How about implementing an SDM to refactor or extend the library model in some useful way before transforming it to text?

5 Conclusion and next steps

Dual congratulations are required here. First, great work on completing your first *bidirectional* model-to-text transformation with TGGs! TGGs might take some getting used to, but remember that you get a backwards transformation and synchronization for free.

Second, if you've really worked through *every* part of this handbook, go get yourself a nice cold beer – you've earned it! It has been a long journey, but you can now consider yourself to be an eMoflon master.

The only part remaining in this handbook is Part VI: Miscellaneous, full of reference documentation and one or two small features that we just weren't able to include with the example. You'll also find tips and tricks which may prove helpful when embarking on your own tooling project with eMoflon.

I suppose we must now say our sad goodbyes. We hope you enjoyed the example and handbook as much as we have enjoyed developing eMoflon (and this handbook). Our tool is constantly evolving, so don't forget to check for updates and new information at `www.emoflon.org`. Finally, if you have any suggestions, questions, feedback or corrections (especially about those screenshots – they get outdated so quickly!), you can reach us anytime at `contact@moflon.org`.

Cheers!

Glossary

Bidirectional Model Transformation Consists of two unidirectional model transformations, which are consistent to each other. This requirement of consistency can be defined in many ways, including using a TGG.

EBNF Extended Backus-Naur Form; Concrete syntax for specifying context-free string grammars, used to describe the context-free syntax of a string language.

Endogenous Transformations between models in the same language (i.e., same input/output metamodel).

Exogenous Transformations between models in different languages (i.e., different input/output metamodels).

In-place Transformation Performs destructive changes directly to the input model, thus transforming it into the output model. Typically *endogenous*.

Out-place Transformation Source model is left intact by the transformation that creates the output model. Can be *endogenous* or *exogenous*.