An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part IV: Triple Graph Grammars

For eMoflon Version 2.16.0

File built on 13th August, 2016

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

The eMoflon team
Darmstadt, Germany (August 2016)

Contents

Part IV:

Learning Box to Dictionary *and* back again with TGGs

URL of this document: https://emoflon.github.io/eclipse-plugin/beta/handbook/part4.pdf

If you're just joining us in this part and are only interested in bidirectional model transformations with *Triple Graph Grammars* then welcome! In fact to keep things simple, we shall assume for the tool-related parts of the handbook that you're starting (over) with a clean slate.

To briefly review what we have done so far in the previous parts of this handbook: we have developed Leitner's learning box by specifying its *abstract syntax* and *static semantics* as a *metamodel*, and finally implementing its *dynamic semantics* via Story Driven Modeling (programmed graph transformations). If the previous sentence could just as well have been in Chinese[1] for you, then consider work through Parts I—III. If you're only in interested in TGGs, however, and you have the appropriate background (or don't care) then that's also fine.

Even though SDMs are crazily cool (don't you forget that!), it is deeply unsatisfactory implementing an inherently *bidirectional* transformation as two unidirectional transformations. If you critically consider the straighforward solution of specifying forward and backward transformations as separate SDMs, you should be able to realise at least the following problems:

Productivity: We have to implement two transformations that are really quite similar, *separately*. This simply doesn't feel productive. Wouldn't it be nice to implement one direction such as the forward transformation, then get the backward transformation for free? How about deriving forward *and* backward transformations from a common joint specification?

[1] Replace with Greek if you are chinese. If you are chinese but speak fluent Greek, then we give up. You get the point anyway, right?

Maintainability: Another maybe even more important point is that two separate transformations often become a pain to maintain and keep *consistent*. If the forward transformation is ever adjusted to produce a different target structure, the backward transformation must be updated appropriately to accommodate the change, and vice-versa. Again, it would be great if the language offered some support.

Traceability: Finally, one often needs to identify the reason why a certain object has been created during a transformation process. This increases the trust in the specified transformation and is essential for working with systems that may actually do some harm (i.e., automotive or medical systems). With two separate transformations, *traceability* has to be supported manually (and that seriously sucks)!

Incrementality and other cool stuff: Traceability links can also be used to propagate changes made to an existing pair of models *incrementally*, i.e., without recreating the models from scratch. This is not only more efficient in most cases (small change and humongous models), but is also sometimes necessary to avoid losing information in one model that simply cannot be recreated with the other model. Finally, having a declarative (meaning here direction agnostic, i.e., neither forward nor backward) makes it easier to derive all kind of different transformations including a model generator (we'll actually get to use this later).

Our goal in this part is, therefore, to investigate how Triple Graph Grammars (TGGs), a cool *bidirectional* transformation language, can be used to address these problems. To continue with our running example, we shall transform `LeitnersLearningBox`, a partitioned container populated with unsorted cards that are moved through the box as they are memorized,[2] into a `Dictionary`, a single flat container able to hold an unlimited number of entries classified by difficulty (Figure 0.1).

To briefly explain, each card in the box has a keyword on one side that a user can see, paired with a definition hidden on the opposite side. We will combine each of these to create the keyword and content of a single dictionary entry, perhaps assigning a difficulty level based on the card's current position in the box. We also want to be able to transform in the opposite direction, transforming each entry into a card by splitting up the contents, and inserting the new element into a specific partition in the box. After a short introduction to TGGs and setting up your workspace correctly, we shall see how to develop your first bidirectional transformation!

[2] For a detailed review on Leitner's Learning Box, see Part II, Section 1

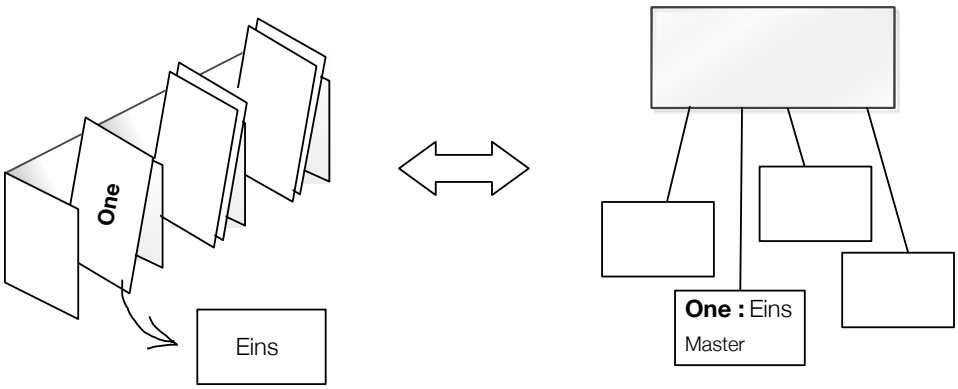

Figure 0.1: Transforming Leitner's learning box into a dictionary

1 Triple Graph Grammars in a nutshell

Triple Graph Grammars [5, 6, 3] are a declarative, rule-based technique used to specify the simultaneous evolution of three connected graphs. Basically, a TGG is just a bunch of rules. Each rule is quite similar to a *story pattern* and describes how a graph structure is to be built-up via a precondition (LHS) and postcondition (RHS). The key difference is that a TGG rule describes how a *graph triple* evolves, where triples consist of a source, correspondence, and target component. This means that executing a sequence of TGG rules will result in source and target graphs connected via nodes in a third (common) correspondence graph.

Graph Triples

Please note that the names "source" and "target" are arbitrarily chosen and do not imply a certain transformation direction. Naming the graphs "left" and "right", or "foo" and "bar" would also be fine. The important thing to remember is that TGGs are *symmetric* in nature.

So far, so good! Except you may be now be asking yourself the following question: "What on earth does all this have to do with bidirectional model transformation?" There are two main ideas behind TGGs:

(1) A TGG defines a consistency relation: Given a TGG (a set of rules), you can inspect a source graph S and a target graph T, and say if they are *consistent* with respect to the TGG. How? Simply check if a triple $(S \leftarrow C \rightarrow T)$ can be created using the rules of the TGG!

If such a triple can be created, then the graphs are consistent, denoted by: $S \Leftrightarrow_{TGG} T$. This consistency relation can be used to check if a

given bidirectional transformation (i.e., a pair (f, b) of a unidirectional forward transformation f and backward transformation b) is correct. In summary, *a TGG can be viewed as a specification of how the transformations* should *behave ($S \Leftrightarrow_{TGG} f(S)$ and $b(T) \Leftrightarrow_{TGG} T$).*

(2) The consistency relation can be operationalized: This is the surprising (and extremely cool) part of TGGs – correct forward *and* backward transformations (i.e., f and b) can be derived automatically from every TGG [1, 2]!

In other words, the description of the simultaneous evolution of the source, correspondence, and target graphs is *sufficient* to derive a forward and a backward transformation. As these derived rules explicitly state step-by-step how to perform forward and backward transformations, they are called *operational* rules as opposed to the original TGG *declarative* rules specified by the user. This derivation process is therefore also referred to as the *operationalization* of a TGG.

Operationalization

Before getting our hands dirty with a concrete example, here are a few extra points for the interested reader:

- Many more operational rules can be automatically derived from the $S \Leftrightarrow_{TGG} T$ consistency relation including inverse rules to *undo* a step in a forward/backward transformation [4],[3] and rules that check the consistency of an existing graph triple.
- You might be wondering why we need the correspondence graph. The first reason is that the correspondence graph can be viewed as a set of explicit traceability links, which are often nice to have in any transformation. With these you can, e.g., immediately see which elements are related after a forward transformation. There's no guessing, no heuristics, and no interpretation or ambiguity.

  The second reason is more subtle, and difficult to explain without a concrete TGG, but we'll do our best and come back to this at the end. The key idea is that the forward transformation is very often actually *not* injective and cannot be inverted! A function can only be inverted if it is *bijective*, meaning it is both *injective* and *surjective*. So how can we derive the backward transformation?

  eMoflon sort of "cheats" when executing the forward transformation and, if a choice had to be made, remembers which target element was

[3] Note that the TGGs are symmetric and forward/backward can be interchanged freely. As it is cumbersome to always write forward/backward, we shall now simply say forward.

chosen. In this way, eMoflon *bidirectionalizes* the transformation on-the-fly with correspondence links in the correspondence graph. The best part is that if the correspondence graph is somehow lost, there's no reason to worry because the *same* TGG specification that was used to derive your forward transformation can also be used to reconstruct a possible correspondence model between two existing source and target models.[4]

This was a lot of information to absorb all at once, so it might make sense to re-read this section after working through the example. In any case, enough theory! Grab your computer (if you're not hugging it already) and get ready to churn out some wicked TGGs!

[4]We refer to this type of operational rule as *link creation*. This turns out to be harder than it appears and support for link creation in eMoflon is currently still work in progress.

2 Setting up your workspace

To start any TGG transformation, you need to have source and target metamodels. Our example will use the `LeitnersLearningBox` metamodel (as completed in Parts II and III) as the transformation's source, and a new `DictionaryLanguage` metamodel as its target.

Even if you've worked through the previous parts of this handbook, we still recommend that you start with a fresh workspace (and even a fresh Eclipse if possible). If you insist on keeping your stuff from the previous parts then ok—but you're on your own, and anything can go wrong (and the universe might implode, etc).

2.1 Get Eclipse

Navigate to `https://www.eclipse.org/downloads/` and download the latest `Eclipse Modeling Tools`.[5] Do not try to use another Eclipse package as it probably won't work.

2.2 Install eMoflon

Install eMoflon as an Eclipse plugin from this update site.[6] When installing, make sure you choose to install *all* available features.

If you don't already have Graphviz dot installed on your system then install it: `http://www.graphviz.org/Download.php`

2.3 Install your initial workspace

To get started in Eclipse, press the "Install, configure and deploy Moflon" button and navigate to "Install Workspace". Choose "Handbook Example (Part 4)" (Figure 2.1). It contains the `LeitnersLearningBox` and `DictionaryLanguage` metamodels, which we'll be using. By the way, don't freak out if your workspace looks slightly different than our screenshots – things often change faster than we can update this handbook. If the difference is important and confusing, however, please send us an email: `contact@moflon.org`

[5] We have tested this part of the handbook for `Eclipse Mars.2 (4.5.2)` running on Mac OS X Yosemite and Windows 8.

[6] `http://www.emoflon.org/fileadmin/download/moflon-ide/eclipse-plugin/beta/update-site2/`

If everything went well, you should now have two projects in your workspace. Go ahead and explore the metamodels (`/model/*.ecore`, `/model/*.aird`). The generated code (`/gen`) is standard EMF code.

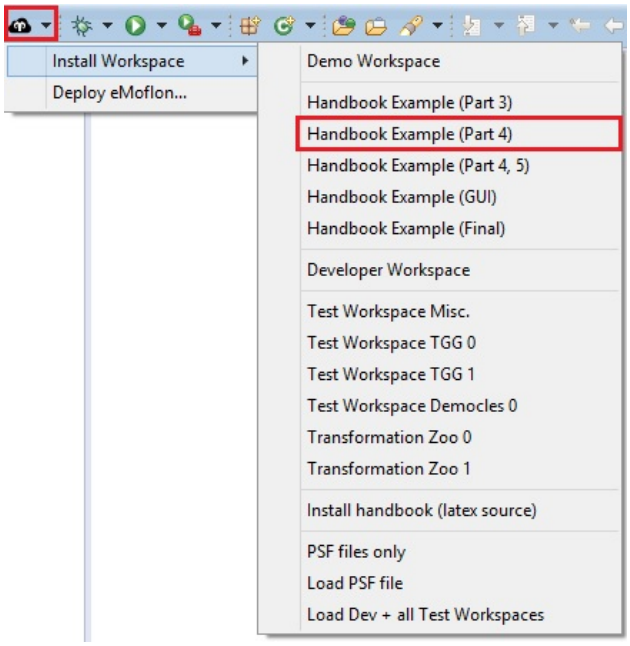

Figure 2.1: Initialize your workspace

If you ever want to create your own "repository" projects, eMoflon provides a wizard ("Create new repository project" in the eMoflon toolbar) for this, which creates the expected project structure with an empty ecore file that you can fill out.

3 Creating your TGG schema

Now that we have our source and target metamodels, we're going to start modeling the correspondence component of our envisioned triple language. This correspondence, or *link metamodel*, specifies *correspondence types*, which will be used to connect specific elements of the source and target metamodels. These correspondence elements can also be thought of as *traceability links*. While the link metamodel is technically just a good old metamodel like any other, eMoflon provides a special wizard and project infrastructure (nature, builder) for so called "integration" projects.

Link Metamodel
Correspondence Types

The overall metamodel triple consisting of the relevant parts of the source, link, and target metamodels is called a *TGG schema*. A TGG schema can be viewed as the (metamodel) triple to which all *new* triples must conform. In less technical lingo, it gives an abstract view on the relationships (correspondence) between two metamodels or domains. A domain expert should be able to understand why certain connected elements are related, irrespective of how the relationship is actually established by TGG rules, just by looking at the TGG schema.

TGG Schema

In our example schema, we will start by creating a link between our source `Box` and target `Dictionary` to express that these two container elements are related.

- Choose the "Create new integration project" wizard from the eMoflon task bar, enter the project name as `LearningBoxToDictionaryIntegration`, and click `Finish` (Figure 3.1).
- Open the schema file `src/org/moflon/tgg/mosl/Schema.tgg`, and import both repository projects by adding import statements as depicted in Figure 3.2. As both projects are eMoflon projects (they don't *have* to be but it makes things simpler) you can use a nifty little template provided by the editor for "eMoflon imports".
- Now state which project is to be source (please choose `LearningBoxLanguage`) and target (choose `DictionaryLanguage`) by stating the root packages in the `#source` and `#target` scopes of the schema (Figure 3.2). As with (almost) everything, the editor will try to help you with a reasonable selection if you hit "Ctrl + Space".
- Finally, create a first correspondence type named `BoxToDictionary` connecting `Boxes` and `Dictionaries` as indicated in Figure 3.2.

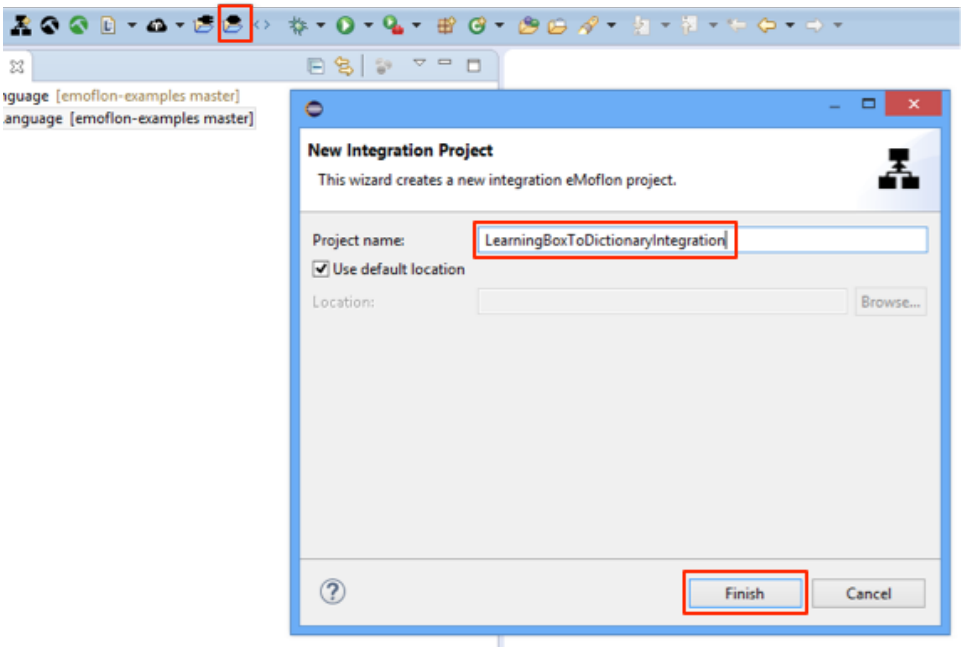

Figure 3.1: Create a new TGG integration project

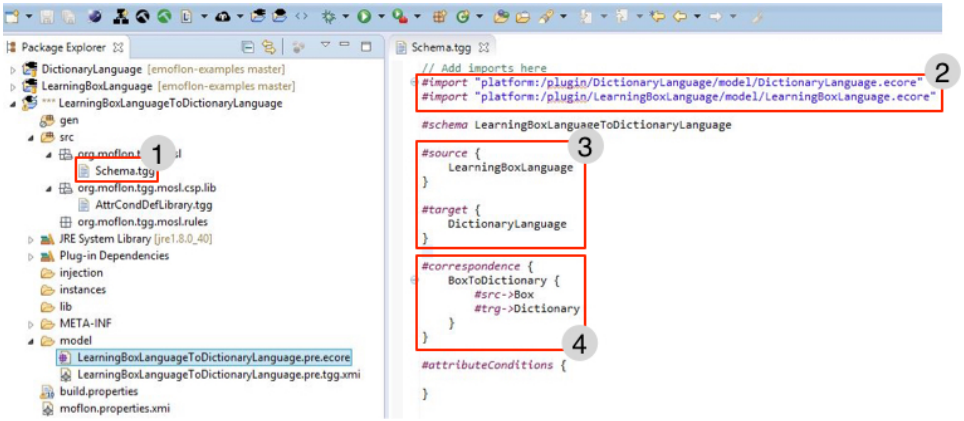

Figure 3.2: Add a first correspondence type

4 Specifying TGG rules

With our correspondence type defined in the TGG schema, we can now specify a set of *TGG rules* to describe a language of graph triples.

As discussed in Section 1, a TGG rule is quite similar to a story pattern, following a *precondition, postcondition* format. This means we'll need to state:

- What must be matched (i.e., under which conditions can a rule be applied; 'black' elements)
- What is to be created when the rule is applied (i.e., which objects and links must exist upon exit; 'green' elements)

Note that the rules of a TGG only describe the simultaneous *build-up* of source, correspondence, and target models. Unlike story diagrams, they do not delete or modify any existing elements. In other words, TGG rules are *monotonic*. This might seem surprising at first, and you might even think this is a terrible restriction. The intention is that a TGG should only specify a consistency relation, and *not* the forward and backward transformations directly, which are derived automatically. In the end, modifications are not necessary on this level but can, of course, be induced in certain operationalizations of the TGG.

Monotonic

Let's quickly think about what rules we need in order to successfully transform a learning box into a dictionary. We need to first take care of the `box` and `dictionary` structures, where `box` will need at least one `partition` to manipulate its `card`s. If more than one is created, those partitions will need to have appropriate `next` and `previous` links. Conversely, given that a `dictionary` is unsorted, there are no counterparts for partitions. A second rule will be needed to transform `card`s into `entries`. More precisely, a one-to-one correspondence must be established (i.e., one `card` implies one `entry`), with suitable concatenation or splitting of the contents (based on the transformation direction), and some mechanism to assign difficulty levels to each `entry` or initial position of each `card`.

4.1 BoxToDictionaryRule

- Select the created subpackage `src/org.moflon.tgg.mosl.rules` (Step 1 in Figure 4.1) and click on the wizard "Create new TGG rule" (Step 2). In the dialogue that pops up, enter `BoxToDictionaryRule` as the name of the TGG rule to be created. Open the newly created file.

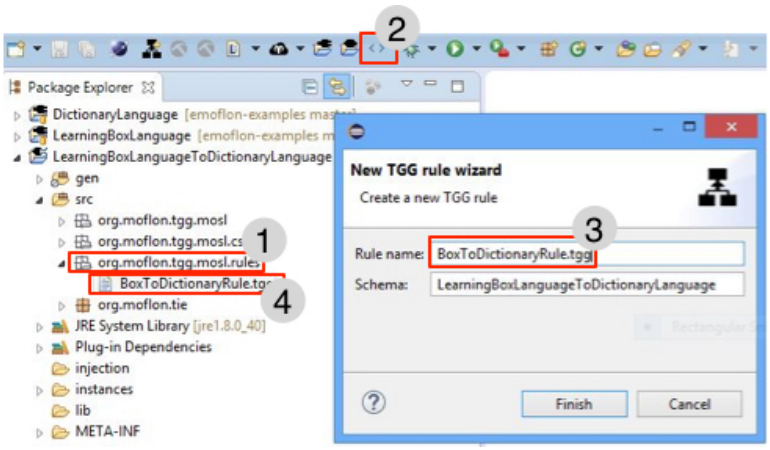

Figure 4.1: Creating a TGG rule

- Ensure that you open the `PlantUML` view (Step 1 in Figure 4.2) as it provides a helpful visualisation of the current TGG rule opened in the editor. To prevent slowing you down, the view is only updated when you save *and* change your cursor position. Elements in the source domain have a yellow background, while target elements have a peach background. Basic UML object diagram syntax is used, extended with colours to represent created elements (green outline) and context elements (black outline). `BoxToDictionaryRule` is a so called"island" rule and only has created elements. For presentation purposes, correspondence links are abstracted from in the visualisation and are depicted as bands (e.g., between `box` and `dictionary`). The rule we want to specify creates a minimal dictionary (just a `Dictionary` node), and a minimal box, which is a bit more interesting as it consists already of three correctly connected partitions. Any structure less than this is not a valid learning box.

- The textual syntax we use is pretty straightforward: every domain has a scope (`#source`, `#target`, and `#correspondence`), and there is a final scope for so called attribute conditions, which express how attribute values relate to each other. The domain scopes contain again scopes for each *object variable* in the rule (e.g., `box`) and these scopes

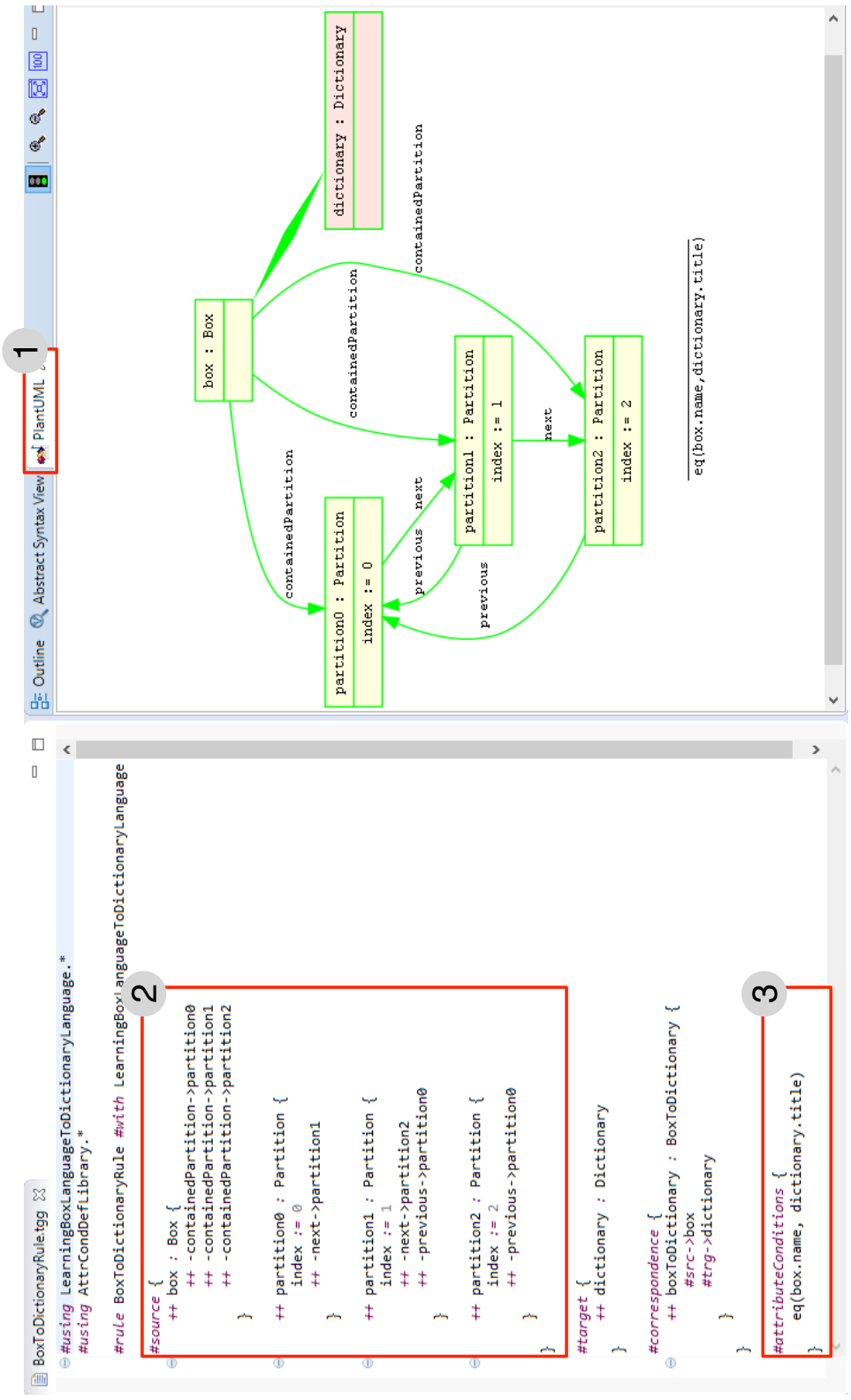

Figure 4.2: Complete TGG rule diagram for `BoxToDictionaryRule`

then contain outgoing *link variables* (e.g., `-containedPartition->`), as well as inline attribute conditions (e.g., `index := 0`). Go ahead and specify the source scope as depicted in Step 2 of Figure 4.2. Notice how the `++` operator is used to mark object and link variables as created or context variables. Try removing and adding it and see how the visualisation changes. As always, let the editor help you by pressing "Ctrl+Space" as often as possible.

- Specify the correspondence and target scopes accordingly and make sure your rule (and its visualisation) closely resembles Figure 4.2.
- To complete our first rule, specify an attribute condition to declare that the name of the box and the title of the dictionary are always to be equal. As depicted in Step 3 Figure 4.2, this can be accomplished using an `eq` condition. We provide a standard library of such attribute conditions (press `F3` on the constraint to jump to the library file), but it is also possible to extend this library with your own attribute conditions. We'll see how to do this in a moment.

Fantastic work! The first rule of our transformation is complete! If you are in hurry, you could jump ahead and proceed directly to Section 5: TGGs in Action. There you can transform a box to a dictionary and vice-versa, but please be aware that your specified TGG (with just one rule) will only be able to cope with completely empty boxes and dictionaries. Handling additional elements (i.e., cards in the learning box and entries in the dictionary) requires a second rule. We intend to specify this next.

4.2 CardToEntryRule

Our next goal is to be able to handle `Card` and `Entry` elements. The new thing here is that it will require a pre-condition – you should not be able to transform these child elements (cards and entries) unless certain structural conditions (there parents exist and are related) are met. In other words, we need a rule that demands an already existing `box` and `dictionary`. It will need to combine 'black' and 'green' variables!

- As we'll be connecting cards and entries, we need a new correspondence type. Open the TGG schema and add a new correspondence type `CardToEntry` connecting a `Card` and an `Entry`.
- Create a new TGG rule with `CardToEntryRule` its name, and specify the rule as depicted in Figure 4.3.

Your diagram should now resemble Figure 4.3. We're not done yet though, we still need to handle attributes! To understand why, consider the current rule and ask yourself *which* partition is meant. Exactly! This is currently not specified, so *any* partition can be taken when applying the rule. eMoflon would actually collect all applicable rules (one for every partition) and consult a configuration component to decide which rules should be taken. The default component just chooses randomly, but you could override this and e.g., pop up a dialogue asking the user which partition is preferred. Although this could be an interesting solution, we'll see how to fully specify things using extra attribute conditions.

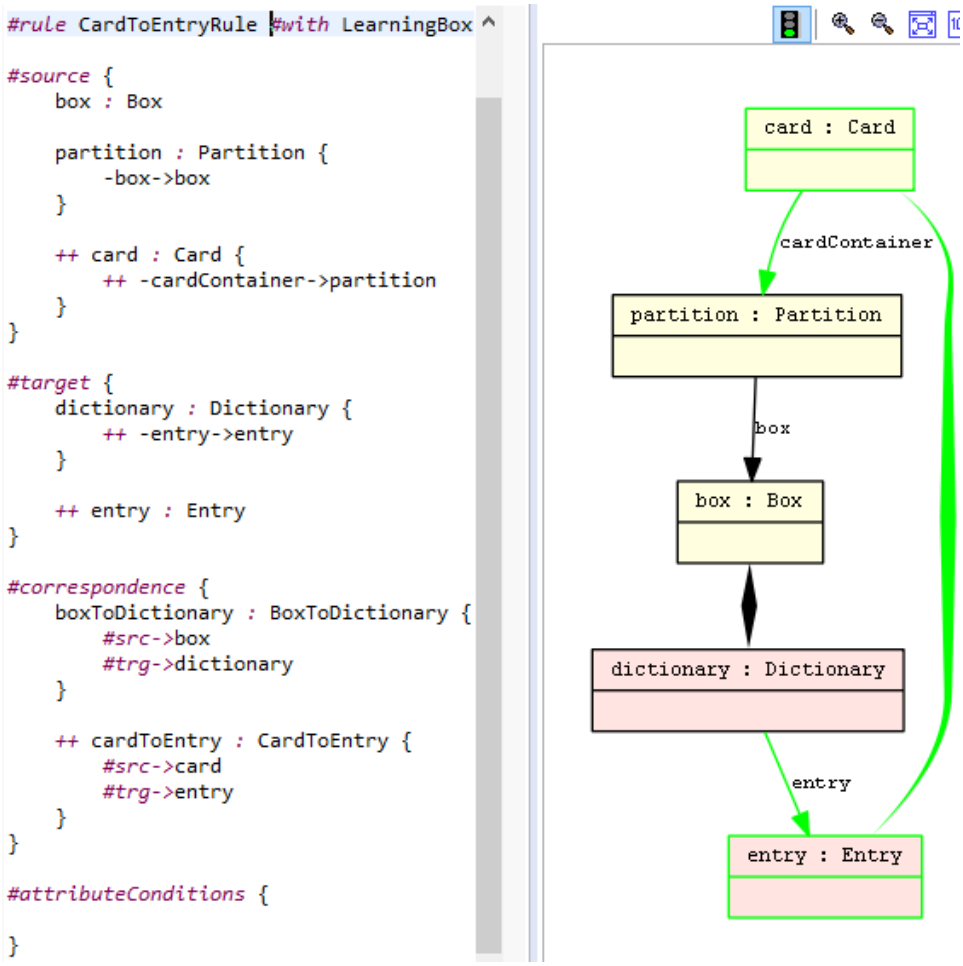

Figure 4.3: `CardToEntryRule` without attribute manipulation

On the way to handling all attributes, let's first start with the relatively easy case of specifying how every `entry.content`, `card.back`, and `card.face` relate to each other. We should probably combine the front and back of

each card as a single content attribute of an entry and, in the opposite direction, split the content into `card.back` and `card.face`.

So let's define `entry.content` as: `<word>:<meaning>`. `card.back` should therefore be `Question:<word>` and similarly, `card.face` should be `Answer:<meaning>`.

- Luckily, we have two library attribute conditions, `addPrefix` and `concat` to help us with this. Add the following to your rule:

  ```
  addPrefix("Question ", word, card.back)
  addPrefix("Answer ", meaning, card.face)
  concat(":", word, meaning, entry.content)
  ```

  "Question " and "Answer " are EString literals, `word` and `meaning` are local variables, and `card.face`, `card.back`, and `entry.content` are attribute expressions (this should be familiar from SDM story patterns).

- Our final task is now to specify where a new `card` (when transformed from an `entry`) should be placed. We purposefully created three partitions to match the three difficulty levels, but if you check the available library constraints, there is nothing that can directly implement this specific kind of mapping. We will therefore need to create our own attribute condition to handle this.

- Open the TGG schema, and define a new attribute condition in the `#attributeConditions` scope. Name it `IndexToLevel`, and enter the values given in Figure 4.4.

  ```
  #attributeConditions {

      #userDefined
      indexToLevel(0:EInt, 1:EString){
          #sync: BB, BF
          #gen:
      }

  }
  ```

  Figure 4.4: Specifying a new attribute condition

- Please note that this is just a *specification* of a custom attribute condition – we still need to actually implement it in Java! As we're so close to finishing this TGG rule, however, let's complete it first before we work out the exact meaning of the mysterious adornments (those

funny `BB`, `BF`, ...) and parameters (`0:EInt` and `1:EString`) of the attribute condition. For now, just make sure you enter the exact values in Figure 4.4.

- We can now use this new attribute condition just like any of the library attribute conditions. Add the following to `CardToEntryRule` to express that the relationship between the index of the partition containing the new card, and the level of the new entry, is defined by our new attribute condition:

  ```
  indexToLevel(partition.index, entry.level)
  ```

If everything has been done correctly up to this point, your project should save and build (hit the hammer symbol in the eMolfon task bar). The generated code will have some compilation errors (Step 1 in Fig. 4.5) as Eclipse does not know where to access the generated code for the imported source and target ecore files (these could also be supplied from jars or installed plugins). In our case the generated code is in the respective source and target projects so let's communicate this to Eclipse.

- Open the `MANIFEST.MF` file (Step 2 in Fig. 4.5) and choose the `Dependencies` tab (Step 3).
- Choose both source and target projects (Step 4) as dependencies and click `OK`. All compilations errors should now be resolved.

Great work! All that's left to do is implement the `indexToLevel` constraint, and give your transformation a test run.

4.3 Implementing IndexToLevel

Our TGG still isn't yet complete. While we've declared and actually used our custom `indexTolevel` attribute condition, we haven't actually implemented it yet. Let's quickly review the purpose of attribute conditions.

Just like patterns describing *structural* correspondence, *attribute conditions* can be automatically *operationalized* as required, e.g., for a forward transformations. Even more interesting, a set of attribute conditions might have to be ordered in a specific way depending on the direction of the transformation. Enforcing the conditions might involve checking existing attribute values, or setting these values appropriately.

For built-in *library* attribute conditions such as *eq*, *addPrefix* and *concat*, you do not need to worry about these details and can just focus on expressing what should hold. Everything else is handled automatically.

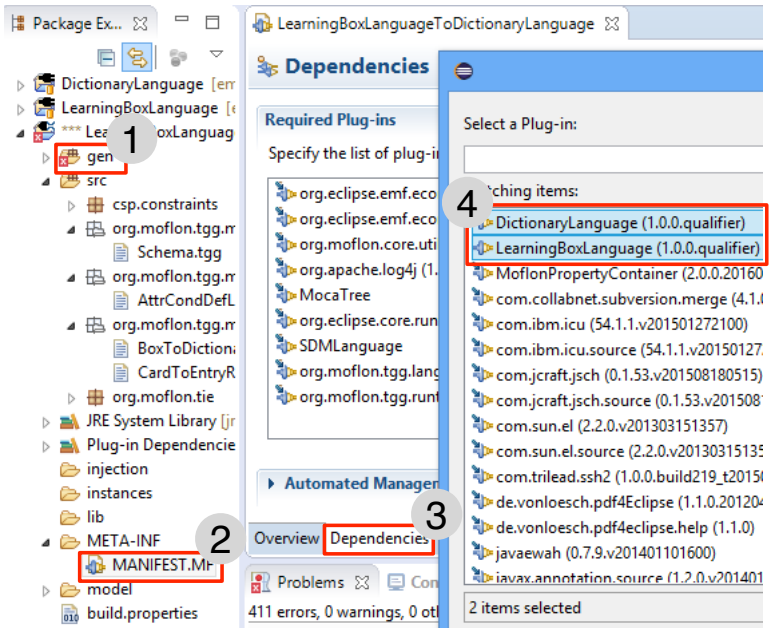

Figure 4.5: Add source and target projects as dependencies

In some cases however, a required attribute condition might be problem-specific, such as our *indexToLevel*. There might not be any fitting combination of library attribute conditions to express the consistency condition, so a new attribute condition type must be declared and implemented.

There is a list of *adornments* in the declaration which specify the cases for which the attribute condition can be operationalized. Each adornment consists of a `B` (bound) or `F` (free) variable setting for each argument of the attribute condition. This might sound a bit complex, but it's really quite simple, especially in the context of our example:

BB indicates that the `partition.index` and `entry.level` are both *bound*, i.e., they already have assigned values. In this case, the *operation* must check if the assigned values are valid and correct.

BF indicates that `partition.index` is *bound* and `entry.level` is *free*, i.e., the operation must determine and assign the correct value to `entry.level` using `partition.index`.

FB would indicate that `partition.index` is *free* and `entry.level` is *bound*, i.e., the operation must determine and assign the correct value to `partition.index` using `entry.level`.

FF would indicate that both `partition.index` and `entry.level` are *free* and we have to somehow generate consistent values out of thin air.

As `partition` is a context element in the rule (the partition is always bound in whatever direction), **FF** and **FB** are irrelevant cases and we do not need to declare or implement what they mean. For the record, note that adornments can be declared as either `#gen` or `#sync`. The reason is that it might make sense to restrict some adornments (typically **FF** cases) to only when generating models. Using **FF** cases for synchronisation might possibly makes sense, but most of the time it would be weird to generate random values during a forward or backward synchronisation.

At compile time, the set of attribute conditions for every TGG rule is "solved" for each case by operationalizing all constraints and determining a feasible sequence in which the operations can be executed, compatible to the declared adornments of each attribute condition. If the set of attribute conditions cannot be solved, an exception is thrown at compile time.

Now that we have a better understanding behind the construction of attribute conditions, let's implement `indexToLevel`.

- Locate and open `IndexToLevel.java` under "src/csp.constraints" in `LearningBoxToDictionaryIntegration`.
- As you can see, some code has been generated in order to handle the current unimplemented state of `IndexToLevel`. Use the code depicted in Figure 4.6 to replace this default implementation.[7]

To briefly explain, the `levels` list contains difficulty level at positions 0, 1, or 2 in the list, which correspond to our three partitions in the learning box. You'll notice that instead of setting "master" to 2, it has rather been set to match the first 0th partition. Unlike an `entry` in `dictionary`, the position of each `card` in `box` is *not* based on difficulty, but simply how it has been moved as a result of the user's correct and incorrect guesses. Easy cards are more likely to be in the final partition (due to moving through the box quickly) while challenging cards are most likely to have been returned to (and currently to be at) the starting position, i.e., the 0th partition.

In the `solve` method, the index of the matched partition in the rule is first of all normalised (negative values do not make sense, and we handle all partitions after partition 2 in the same way). A switch statement is then used, based on whichever adornment is currently the case, to enforce or check the condition.

[7] Depending of course on your pdf viewer, copy and pasting this code should work.

```
package csp.constraints;

import java.util.Arrays;
import java.util.List;
import org.moflon.tgg.language.csp.Variable;
import org.moflon.tgg.language.csp.impl.TGGConstraintImpl;

public class IndexToLevel extends TGGConstraintImpl {

    private static List<String> levels =
      Arrays.asList(new String[] {"master","advanced","beginner"});

    public void solve(Variable var_0, Variable var_1) {
        int index = ((Integer) var_0.getValue()).intValue();
        int normalisedIndex = Math.min(Math.max(0, index), 2);
        String bindingStates = getBindingStates(var_0, var_1);

        switch (bindingStates) {
        case "BB":
            String level = (String) var_1.getValue();
            setSatisfied(levels.get(normalisedIndex).equals(level));
            break;
        case "BF":
            var_1.bindToValue(levels.get(normalisedIndex));
            setSatisfied(true);
            break;
        }}}
```

Figure 4.6: Implementation of our custom `IndexToLevel` constraint

For `BB` we check if the normalised index of the partition corresponds to the difficulty level of the card. For `BF`, the normalised index is used to set the appropriate difficulty level of the card.

5 TGGs in action

Before we can execute our rules, we need to create something for the TGG to transform. In other words, we need to create an instance model[8] of either our target or our source metamodel! Since dictionaries are of a much simpler structure, let's start with the backwards transformation.

- Navigate to `DictionaryLanguage/model/` and open `DictionaryLanguage.ecore`. Expand the tree and create a new dynamic instance of a `Dictionary` named `bwd.src.xmi` (choose the EClass `Dictionary`, right-click, and select `Create dynamic instance`). Make sure you persist your instance as `LearningBoxToDictionaryIntegration/instances/bwd.src.xmi` (as depicted in Figure 5.1).

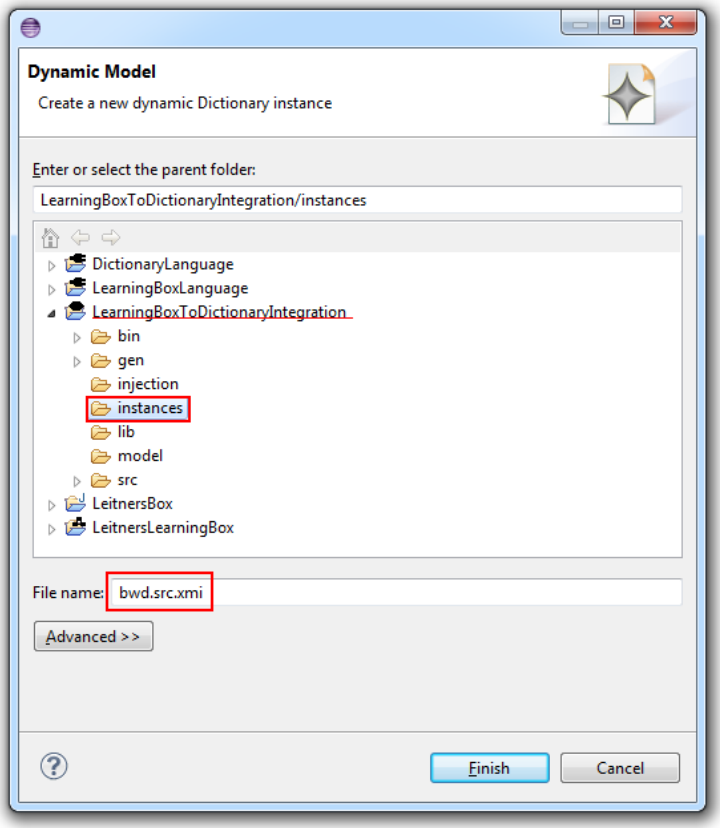

Figure 5.1: Create a dynamic instance of `Dictionary`

[8] For a detailed review how to create instances, refer to Part II, Section 3

- Open the new file and edit the `Dictionary` properties by double-clicking and setting `Title` to `English Numbers` in the `Properties` tab below the window.

- Create three child `Entry` objects. Don't forget the syntax we decided upon for each `entry.content` in the `CardToEntryRule` when setting up the constraints! Be sure to set this property according to the format `<word>:<meaning>`. Give each `entry` a different difficulty `level`, e.g., `beginner` for `One:Eins`, `advanced` for `Two:Zwei`, and `master` for `Three:Drei`. Your instance should resemble Figure 5.2. After this works you should of course play around with the model and add all kind of cards to see what happens.

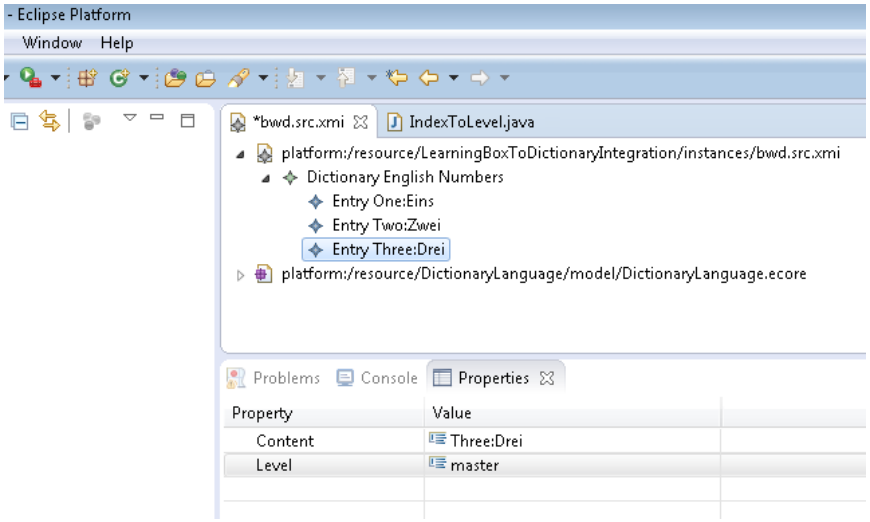

Figure 5.2: Fill a `Dictionary` for the transformation

- eMoflon generates default stubs to execute the transformation. Navigate to "LearningBoxToDictionaryIntegration/src/org.moflon.tie" and open `LearningBoxToDictionaryIntegrationTrafo.java`.

- As you can see, this file is a driver for forward and backward transformations, transforming a source `box` to a target `dictionary`, and backward from a `dictionary` to a `box`. As this is good old Java code, you can adjust everything as you wish. To follow this handbook, however, do not change anything for the moment.

- Right-click the file in the Package Explorer and navigate to "Run as. . . /Java Application" to execute the file.

- Did you get one error message, followed by one success message in the eMoflon console window (Figure 5.3) below the editor? Perfect! Both of these statements make sense – our TGG first attempted the forward transformation but, given that it was missing the source (`box`) instance, it was only able to perform a transformation in the backwards direction. The stub checks for `fwd.src.xmi` and `bwd.src.xmi` files per convention (all this can of course be edited and changed).

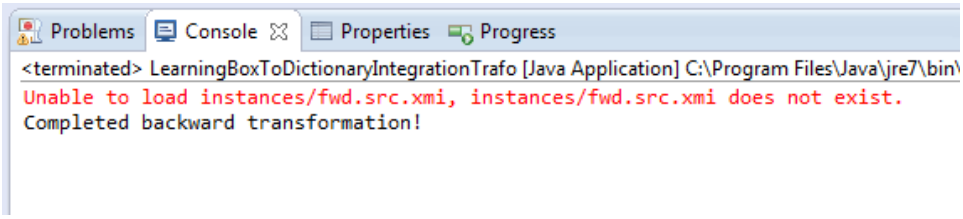

Figure 5.3: Running the backward transformation

- Refresh the integration project's `instances` folder. Three new `.xmi` files should have appeared representing your backward triple. While you created the `bwd.src.xmi` instance, the TGG generated `bwd.corr.xmi`, the correspondence graph between target and source, `bwd.protocol.xmi`, a directed acyclic graph of the rule applications that lead to this result, and `bwd.trg.xmi`, the output of the transformation.

- If you open and inspect `bwd.trg.xmi` you'll find that it's a `Box` of `English Numbers`. Expand the tree and you'll see our `Dictionary` in its corresponding `Box` format containing three `Partitions` (Figure 5.4). Double click each `card` and observe how each `entry.content` was successfully split into two sides. Also note how the cards were placed in the correct (at least according to our `indexToLevel` attribute condition) partition.

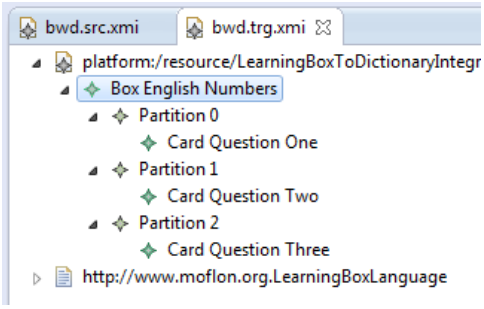

Figure 5.4: Result of the *backwards* transformation

- Congratulations! You have successfully performed your first *backward* transformation using TGGs! To understand better what happened, eMoflon provides two important tools: the first is the correspondence model containing all established links between source and target elements, the second is the so called protocol of the transformation.

- Go ahead and open `bwd.corr.xmi`. This model contains the created correspondence model, and references both source and target models. Using eMoflon's `Abstract Syntax View`, you can drag in a selection of elements and explore (right-click on a node in the view to choose if direct neighbours or all transitive neighbours should be added to the view) how the models are connected. You can remove elements from the view, zoom in and out, and also pick from a choice of standard layout algorithms. Figure 5.5 shows all three complete models in the view with all correspondence links highlighted. You can choose elements in the view to see their attribute values either as a tool tip, or in the standard properties view in Eclipse.

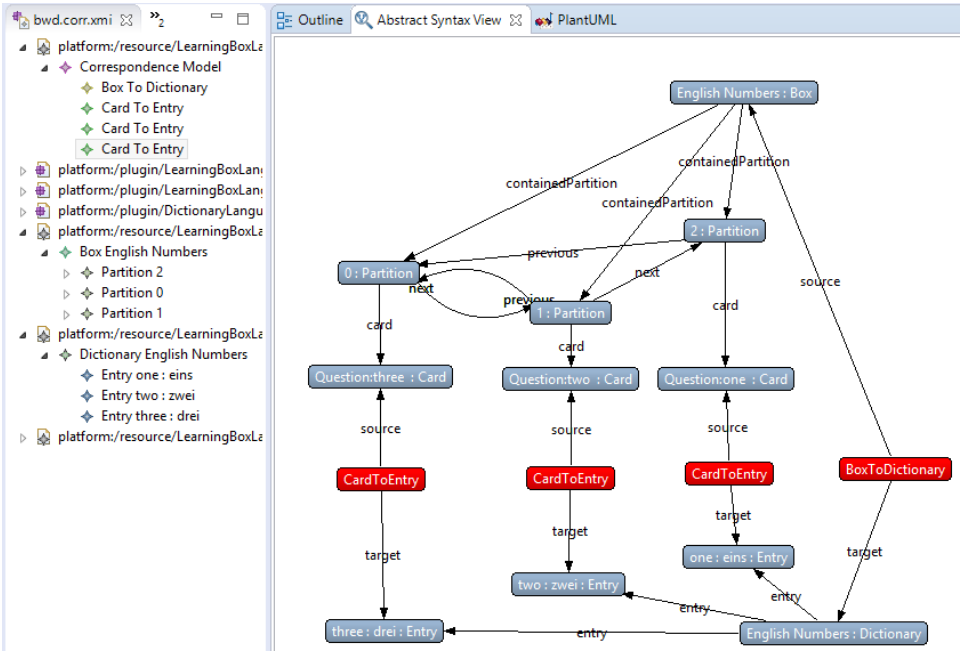

Figure 5.5: Inspecting the correspondence models

- eMoflon provides a special visualisation for protocol models, so go ahead and open `bwd.protocol.xmi`. This time around, use the PlantUML view to inspect the protocol. If you select the root element (`Precedence Structure`), you'll see a tree of *rule applications*, showing you which rules were applied to achieve the result. As `CardTo-`

`EntryRule` can only be applied after `BoxToDictionaryRule`, an application of the latter is a parent for all other rule applications. These are all children on the same level, as the exact order of application is irrelevant in this case. This, in general, directed acyclic graph of rule applications is depicted in Figure 5.6.

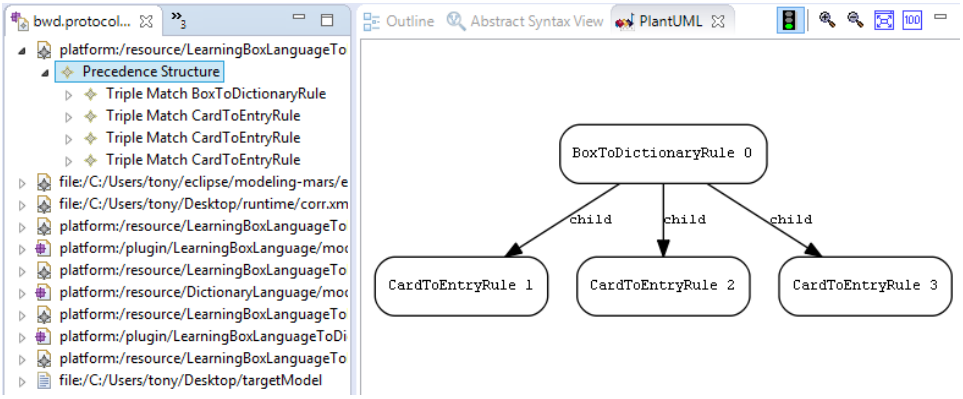

Figure 5.6: Inspecting the protocol

- To inspect a rule application, choose one of the children of the precedence structure in the tree view. If you choose, for example, a `CardToEntryRule` node, a diagram similar to that depicted in Figure 5.7 will be shown as a visualisation of the rule application. To understand this diagram, note that the labels of the nodes are of the form `rule variable ==> model element`, showing how the object and link variables in the rule were "matched" to actual elements in the source and target models. The colours indicate which elements would be created by applying this rule (in the simultaneous evolution of all three models). This protocol is not only for documentation, but is also used for model synchronisation, which we'll discuss and try out in a moment.

- To convince yourself that the transformation is actually bidirectional, create a source model, and run the TGG again to perform a *forwards* transformation of a `Box` into a `Dictionary`. The easiest way to do this is to make a copy of `bwd.trg.xmi` and rename it to `fwd.src.xmi`.

- Run `LearningBoxToDictionaryIntegrationTrafo` again and refresh the "instances" folder. Compare the output `fwd.trg.xmi` against the original `bwd.src.xmi` Dictionary model. If everything went right, they should be isomorphic (identical up to perhaps sorting of children).

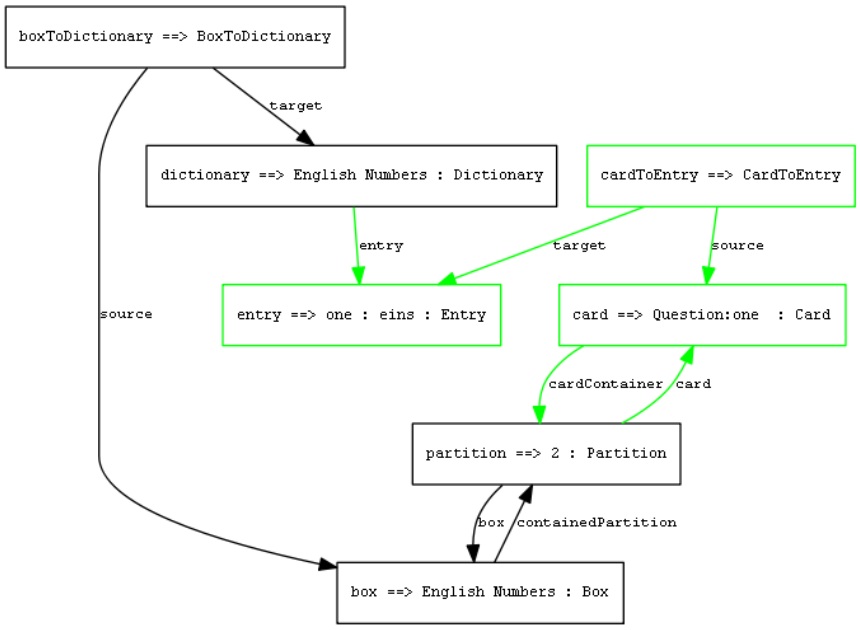

Figure 5.7: Inspecting a rule application

6 Extending your transformation

At this point, we now have a working TGG to transform a `Dictionary` into a `Box` with three `partitions`, and a `Box` with exactly three `Partitions` into a `Dictionary`. The only potential problem is that a learning box with only three partitions may not be the most useful studying tool. After all, the more partitions you have, the more practice you'll have with the cards by being quizzed again and again.

A simple strategy would be to allow additional partitions in the box, but to basically ignore them (treat them all as partitions with index greater than two) when transforming to the dictionary. To accomplish this we need an extra rule that clearly states how such partitions should be ignored, i.e., be translated without affecting the dictionary. We could trivially extend the existing `BoxToDictionaryRule` by connecting a fourth partition, but what if we wanted a fifth one? A sixth? As you can see, this obviously won't work – there will always be the potential for a `n+1`th partition in an `n`-sized box.

With a so-called *ignore rule*, we'll handle some source elements and their connecting link variables without creating any new elements in the target domain. Before specifying this ignore rule, however, let's extend the current

ignore rule

`fwd.src.xmi`[9] by adding a new `Partition` (with `index` = 3) as depicted in Figure 6.1. Connect the new partition `partition3` to `partition0` via a `previous` reference, and connect `partition2` to `partition3` via a `next` reference (this is how to extend a learning box). Create a new `card` in your new `partition3` as well.

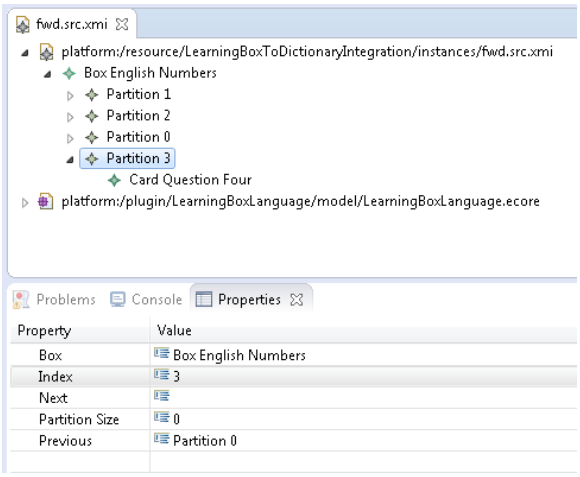

Figure 6.1: Extended `fwd.src.xmi`

If you run your transformation again, you'll get an error message in the console for the forward direction as our `fwd.src.xmi` with four `partitions` cannot be handled with our current TGG. As this might happen quite often when working on a TGG, let's take some time to understand the message and "debug" what went wrong. The first message in the console is:

> *Your TGG LearningBoxLanguageToDictionaryLanguage is not translation locally complete!*

There are basically two things that can go wrong when transforming a source model to a target model with a TGG-based forward transformation: (1) the input model cannot be completely marked (or "parsed", or recognized, etc.) using TGG rules, and (2) some thing went wrong during the translation of the input model, i.e., when trying to extend the output model. The first problem is referred to as *input local completeness*, while the second problem is referred to as *translation local completeness*. The choice of "input" vs. "translation" should be clear from the above explanation. The word "local" indicates that the transformation is always with respect to a certain match, i.e., a restricted fragment of the models. If a TGG is "complete" then these

[9] Remember that you should have created this by copying and renaming `bwd.trg.xmi`.

two problems cannot occur. This is a nice property to have and getting the compiler to check for this statically is ongoing work.

What makes things a bit ugly is that our algorithm does not backtrack while searching for a valid rule application sequence as this would make things extremely (exponentially) slow. The price for this is that we cannot easily differentiate between the case where a TGG rule is simply missing, and the case where a TGG is somehow nasty (requires backtracking) and our algorithm has gotten confused and hit a dead end. This is why the next message in the console is:

> *[...] I was unable to translate [...] without backtracking.*

and not (or slightly more polite isomorphisms thereof):

> *You jerk, I don't have a TGG rule to translate this element!!!*

as you might expect for our concrete example.

After asking you nicely to take a good look at your TGG, the synchroniser tells you exactly where things went wrong:

> *I got stuck while trying to extend the following source matches: [CardToEntryRule]*

That's a bit surprising right? You probably expected everything to work (as it did before), up to the point where the extra partition turns up and no rule can be found for it. Well things aren't that straightforward. To get more information, the synchroniser suggests:

> *Set verbose to true in your synchronization helper and re-run to get the exact list of ignored elements.*

so let's do that and see what happens.

- Open `src/org/moflon/tie/LearningBoxLanguageToDictionaryLanguageTrafo.java` and extend the code invoking the forward transformation as follows:

```
// Forward Transformation
LearningBoxLan... helper = new LearningBoxLan...Trafo();
helper.setVerbose(true); // <--  add this!
helper.performForward("instances/fwd.src.xmi");
```

If you now rerun the transformation, you'll get a printout of exactly which elements could not be translated at all. Curiously, this lists include all partitions *and* the box (`[English Numbers]`). Although we still do not know why the box and the first three partitions could not be translated using `BoxToDicationaryRule`, at least we can now understand why the translation failed at `CardToEntryRule`. The following happened:

1. No match could be found for the box and any partition. Instead of complaining directly, the algorithm assumes that we do not care about these elements (this is very often the case in practice), so it ignores them and tries to continue with the translation.

2. When translating a card, however, although things look ok on the source side, there is no `Dictionary` on the target side and the translation fails. If we would not add the created `Entry` to the `Dictionary`, this rule application would have worked!

OK – let's now understand why the box was already ignored.

- Following the four steps in Figure 6.2, open the generated ecore file in the integration project (Step 1). This file contains not only the correspondence metamodel, but also all operationalized rules. Java code for the transformation is generated directly from this file.

- Under `Rules/BoxToDictionaryRule/` locate the so called "isAppriopriate_FWD..." method for the TGG rule. For every TGG rule, three main types of operational rules are derived in each direction: "isAppropriate" methods to check if a match for a rule can be found, "isApplicable" to check if this match can be extended to cover all domain, and "perform" methods to actually apply the rule in the forward or backward direction. These methods are all generated as unidirectional programmed graph transformations (story diagrams). If you want to see how the generated story diagram looks like, go ahead and select the `Activity` element under the operation (you should get a visualisation as a simple activity diagram).

- The most important story node in the story diagram (Step 2) is `test core match and DEC` (all others are more or less bookkeeping and technical stuff). DEC stands for "Dangling Edge Condition" and represents a lookahead for the algorithm. The basic idea is to check for edges that would be impossible to translate if this rule is applied at this location. This can be checked for statically and the result of this DEC analyis is embedded in the transformation as simple Negative

Application Conditions (NACs). Go ahead and select the story pattern (Step 3). Note that it was derived directly from the TGG rule and, in this sense, *is still a* TGG rule (at least according to the metamodel). You should see an object diagram representing the rule. Note that black means context, while blue means negative (it should not be possible to extend a match to cover any of these elements).

- In our case, there are quite a few edges that would be left (in this sense) dangling. An example is shown as Step 4 in Figure 6.2: if the box has any other contained partition apart from the 3 matched in this rule, then it is clear that this extra edge cannot be translated with any other rule in the current TGG. It would thus be dangling and therefore blocks the application of this rule. Another example would be a next edge going out from `partition2`. Can you locate the NAC for this edge?

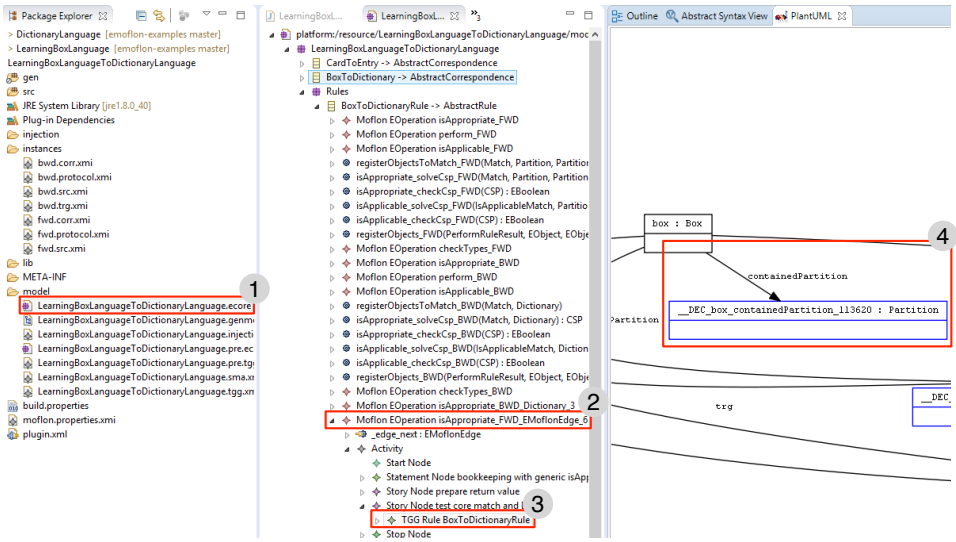

Figure 6.2: Understanding the Dangling Edge Condition

DEC is a great help when it comes to avoiding dead ends without using backtracking, but in our case where the TGG is actually missing a rule, it only postpones the problem and makes it a bit challenging to understand what went wrong. On the bright side we took the chance to dig in a bit right? If you ever have problems understanding why a certain match was not collected, feel free to debug the generated Java code directly if looking at the visualisation does not help (as we did here). Just place breakpoints as usual and run the transformation in debug modus. The generated code is quite readable (at least after a week of practice – haha!).

6.1 AllOtherPartitionsRule

- Create a new rule `AllOtherPartitionsRule`, and complete it according to Figure 6.3.

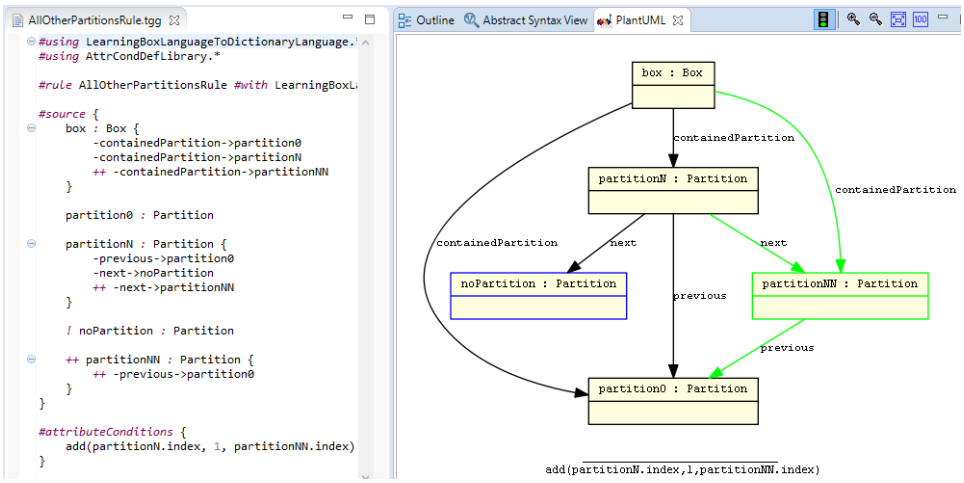

Figure 6.3: The completed `AllOtherPartitionsRule`

- As you can see, this rule doesn't assume to know the final `partition` in the transformation. It matches the `n`th partition as the partition without any next partition, then connects a new `n+1`th partition to `n` and `partition0` (clear as every partitions previous is `partition0`). Note that TGG transformations assume that the models are valid, i.e., have the expected structure (in our case meaning that the learning box is correctly "wired").[10] Remember that "blue" means "negative".

- Generate code for your improved TGG and re-run the transformation. It should work now without any error message. Inspect the protocol to understand what happened.

- Go ahead and add as many `partition`s and `card`s as you like to your model instance. Your TGG is now also able to handle a `box` with any number of `partition`s beautifully. For five partitions all with cards, the protocol gets quite interesting and is no longer a flat tree. Try it out!

[10] This should actually be formalised with a set of metamodel constraints that must be checked before a transformation is run, but we've omitted this here to simplify things.

7 Model Synchronization

At this stage, you have successfully created a trio of rules that can transform a `Box` with any number of `Partitions` and `Cards` into a `Dictionary` with an unlimited number of `Entrys` (or vice versa).

Now suppose you wanted to make a minor change to one of your current instances, such as adding a single new card or entry into one of your instances. Could you modify the instance models and simply run the transformation again to keep the target and sources consistent?

The current `fwd.src.xmi` file (Figure 6.1) has a partition with an index of three which, when transformed, correctly produces a target dictionary with all four entries. What would happen if we attempted to transform this dictionary back into the same learning box, with all four partitions?

- Copy and paste `fwd.trg.xmi`, renaming it as `bwd.src.xmi`.[11]
- Run `LearningBoxToDictionaryIntegrationTrafo.java` and inspect the resulting `bwd.trg.xmi` (Figure 7.1). Unfortunately, the newest `partition3` is missing!

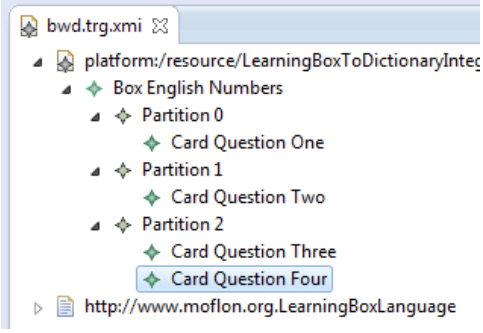

Figure 7.1: We lose information if the original box has more than 3 partitions

If you think about this for a moment it should be clear that our TGG-based backward transformation can only *create* a box with exactly three partitions. Why doesn't it try to apply `AllOtherPartitionsRule` you might be thinking? Well, given only the target model it is simply unclear how often this rule should be applied: Once? Twice? A hundred times? We could of course integrate user interaction to decide, but for a TGG of realistic size this would get pretty messy pretty soon.

[11] Feel free to either delete or rename the original `bwd.src.xmi` for later reference.

When going from the source to the target model, this information (how many partitions exist) is simply discarded and lost. So how can we prevent this information loss when we need to update our models in the future? Luckily, eMoflon can take care of this for you as it provides a *synchronization* mode to support incremental updates.

Let's change our source model by adding a new `Card` to `Partition3` and see if the partition still exists after synchronizing to and from the resulting `Dictionary` model.

- Make sure your `fwd.src.xmi` model is in the "extended" version depicted in Figure 6.1, and that you have a fresh triple of corresponding `fwd.corr.xmi` and `fwd.trg.xmi` created via the forward batch transformation.

- Our synchronisation algorithm requires an explicit "delta" representing the exact changes that are to be propagated. Although this can be extracted from different versions of a model this is not as easy as it sounds. To make matters worse, "delta recognition" is also often not unique, i.e., there are many possible deltas for the same pair of old and new models.

  As we are more interested in the actual synchronisation than in change recognition or "diffing", we require the exact delta which can come from anywhere in actual applications, including being programmed as a hard coded change operation in some kind of synchroniser GUI.

  To play around with a TGG, however, we provide a simple *delta editor*, which behaves very much like the standard EMF model editor, but records all the changes you make and persists them as a delta model, which we understand.

  To open the delta editor, right-click on `fwd.trg.xmi` and choose `Open With/Delta Editor`.

- Let's try a simple attribute change: As `Entry four:vier` was created from a card in the third partition, its default level is beginner (remember we assume easy cards have wondered further into the box). We'll we think this is wrong so let's change the level to `master`. Perform this change directly with the delta editor as if it were the standard EMF model editor (compare with Figure 7.2). Save the editor and close it. Take a look at the created `fwd.trg.delta.xmi` and inspect it. Although we do not (yet) have a fancy visualisation for delta models, it should be pretty easy to see how your change has been represented as a data structure. Finally, note that `fwd.trg.xmi` has *not* been changed yet, i.e., the synchroniser will do this when trying to propagate your

changes. This is important to understand as handling conflicts (work in progress, planned for a release in the far, far away future) might imply a compromise, i.e., not all your changes might be accepted or possible.

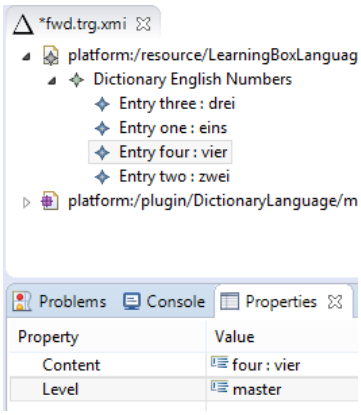

Figure 7.2: Change the level of an entry

- To backward propagate your changes, open `LearningBoxLanguage-ToDictionaryLanguageSync.java`, our generated stub for model synchronisation. Our stubs are optimized for this handbook so you do not have to change a thing. It's all plain and simple Java though, so you should be able to understand how our API is being used. The section where you can adjust and make changes is:

  ```
  // Adjust values as required
  String delta = "instances/fwd.trg.delta.xmi";
  String corr  = "instances/fwd.corr.xmi";
  BiConsumer<...> synchronizer = helper::syncBackward;
  ```

  These values all fit our current change (the created delta file, the relevant correspondence model, and we want to synchronise backwards), so we're ready to go. Hit `Run` and see what happens! If everything went well, `Card question:four` should have been relocated to `partition0` to fit its corrected difficulty level. The great thing is `partition3` is now empty, *and has been retained*. Not that the protocol has been updated, and that `fwd.trg.xmi` not contains these changes.

Remember this is all academic software and especially the delta editor is rather new and very buggy. Feel free to try out all kinds of deltas and if you think you've found a bug, please send us an email at `contact@moflon.org` and we'll be happy to fix it (or reply with a "back in your face!" if it's not one and *you* just got confused).

8 Model Generation with TGGs

In addition to model transformation and model synchronization, TGG specifications can be used directly to generate models. Often there is a need for large and randomly generated models for testing purposes and it's surprisingly hard and awful work to whip up such a generator that *only* creates valid models with respect to the TGG. Once you add or change a rule – puff! Your generator produces rubbish. It makes sense to automate this generation process and eMoflon provides some basic support.

- We assume for this section, that you've completed the previous section and have the source, target, and integration projects in their final versions in your workspace.

  In your project `src` folder locate and open `LearningBoxToDictionaryIntegrationModelGen.java` (Figure 8.1), our default stub for model integration.

```
30  public static void main(String[] args) throws IOException
31  {
32     // Set up logging
33     BasicConfigurator.configure();
34
35     AbstractModelGenerationController controller = new DefaultModelGenController();
36     controller.addContinuationController(new MaxRulePerformCounterController(20));
37     controller.addContinuationController(new TimeoutController(5000));
38     controller.setRuleSelector(new LimitedRandomRuleSelector().addRuleLimit("<enter rule name>", 1));
39
40     ModelGenerator gen = new ModelGenerator(LearningBoxToDictionaryIntegrationPackage.eINSTANCE, controller);
41     gen.generate();
42  }
```

Figure 8.1: Stub for the model generator

The `ModelGenerator` class uses an `AbstractModelGenerationController` to control the generation process (Line 35). In this default template the generation process will be terminated after 20 rules have been applied (`MaxRulePerformCounterController` (Line 36)). Additionally, the `TimeoutController` will terminate the process after 5000ms (Line 37). You can use the `MaxModelSizeController` class to terminate the generation process if a specific model size has been reached. The `RuleSelector` controls which rules are selected as the next to be executed. The built-in `LimitedRandomRuleSelector` always selects a random rule and has the additional feature to limit the number of performs for specific rules (Line 38). As with everything we generate, this is standard Java code, doesn't bite, and can be extended as you wish with your own controller classes and generation strategies.

- To ensure that we only get a model with a single root, change `<enter`

`rule name`> to `BoxToDictionaryRule` so that this island rule (which is always applicable) will only be applied once.

- Save the file and hit `Run`!

  You'll probably get some logging information in the console (Figure 8.2) containing information gathered during the generation process such as model size for each domain, number of performs for each rule, duration of generation process for each rule etc.

```
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - --- Model Generation Log ---
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - performs: 26
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - duration: 105ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - failures: 1
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - 29/11 nodes/edges created for source
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - 20/0 nodes/edges created for target
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - 20/40 nodes/edges created for correspondence
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - BoxToDictionaryRule
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    performs: 1
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    duration: 56ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    failures: 0
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - CardToEntryRule
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    performs: 19
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    duration: 35ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    failures: 0
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - AllOtherCardsRule
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    performs: 6
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    duration: 14ms
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  -    failures: 1
70 [main] INFO org.moflon.tgg.algorithm.modelgenerator.ModelGenerator  - --- Model Generation Log ---
```

Figure 8.2: Logging output after model generation

In your `instances` folder there should now be a new folder named `generated-Models` with a timestamp suffix. It contains your newly generated source and target models. Have fun viewing. Why not try creating like seriously gigantic models?

Note that to support model generation for custom attribute condition in general, you might have to specify additional `#gen` adornments. No additional adornments were required for our new attribute condition, and all library conditions suppo

9 Conclusion and next steps

Absolutely amazing work – you've mastered Part IV of the eMoflon handbook! You've learnt the key points of Triple Graph Grammars and *bidirectional* transformations, and now know how to set up a TGG consisting of a schema and a set of rules. With these basic skills, you should be able to tackle quite a few bidirectional transformations using TGGs.

For detailed descriptions on the upcoming and previous parts of this handbook, please refer to Part 0, which can be found at `https://emoflon.github.io/eclipse-plugin/beta/handbook/part0.pdf`.

Cheers!

Glossary

Correspondence Types Connect classes of the source and target metamodels.

Graph Triples Consist of connected source, correspondence, and target components.

Link or correspondence Metamodel Comprised of all correspondence types.

Monotonic In this context, non-deleting.

Operationalization The process of deriving step-by-step executable instructions from a declarative specification that just states what the outcome should be but not how to achieve it.

Triple Graph Grammars (TGG) Declarative, rule-based technique of specifying the simultaneous evolution of three connected graphs.

TGG Schema The metamodel triple consisting of the source, correspondence (link), and target metamodels.

References

[1] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars. In *2010 Workshop on Model-Driven Engineering, Verification, and Validation*, pages 19–24. IEEE, October 2010.

[2] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In Thomas Whittle, Jon and Clark, Tony and Kühne, editor, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 668–682, Berlin / Heidelberg, 2011. Springer.

[3] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Andy Schürr, C. Lewerentz, G. Engels, W. Schäfer, and B. Westfechtel, editors, *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *Lecture Notes in Computer Science*, pages 141–174. Springer, Heidelberg, November 2010.

[4] M Lauder, A Anjorin, G Varró, and A Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In *Proceedings of the 6th International Conference on Graph Transformation*, Lecture Notes in Computer Science (LNCS), Heidelberg, 2012. Springer Verlag.

[5] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G Tinhofer, editor, *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[6] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *4th International Conference on Graph Transformation*, volume

5214 of *Lecture Notes in Computer Science (LNCS)*, pages 411–425, Heidelberg, 2008. Springer Verlag.