An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part IV: Triple Graph Grammars

For eMoflon Version 2.0.0

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

The eMoflon team
Darmstadt, Germany (August 2015)

Contents

Part IV:

Learning Box to Dictionary *and* back again with TGGs

`Approximate time to complete: 1h 30min`

`URL of this document: http://tiny.cc/emoflon-rel-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! To ensure eMoflon is running correctly however, you should at least work through Part I for the required installation and setup instructions. We try to assume as little as possible from the previous parts in the handbook series, and give appropriate references where necessary.

To briefly review what we have done so far: 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 please work through Parts II and III.

Even though SDMs are crazily cool (don't you forget that!), it is rather unsatisfactory implementing a *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 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

[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?

about deriving forward *and* backward transformations from a common joint specification?

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! 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, but is also sometimes necessary to avoid losing information in one model that simply cannot be recreated with the other model.

Our goal is to investigate how Triple Graph Grammars (TGGs), a *bidirectional* transformation language, can be used to address these problems. To continue with our running example, we plan to 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 (Fig. 0.1).

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

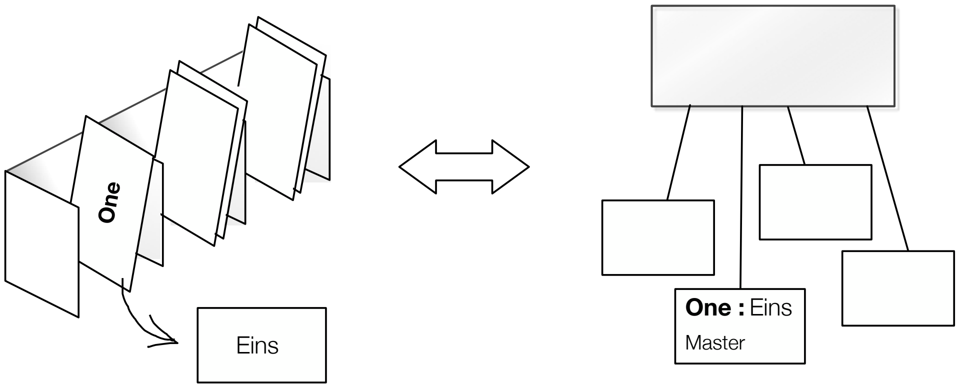

Figure 0.1: Transforming Leitner's learning box into a dictionary

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 will see how to develop your first bidirectional transformation!

1 Triple Graph Grammars in a nutshell

Triple graph grammars [**?**, **?**, **?**] are a declarative, rule-based technique of specifying 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 understanding 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 unidirectional forward (f) and backward (b) transformation) 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 – forward *and* backward rules (i.e., S or T) can be derived automatically from every TGG rule [**?**, **?**]!

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 [**?**],[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 nice to have in any transformation. With these you can, e.g., immediately see which elements are related

[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.

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 what target element was chosen. In this way, eMoflon *bidirectionalize*s 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 may 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 TGGs!

[4]We refer to this type of operational rule as *link creation*. Support for link creation in eMoflon is currently work in progress.

2 Setting up your workspace

To start any TGG transformation, you need to have the 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.

If you haven't worked through the previous parts of this handbook, complete Section 2.1 first to load the source learning box metamodel into your workspace. If you already have it from completing a previous part however, skip ahead to either `Section 2.2 (Visual)` or `Section 2.3 (Textual)` to begin.

2.1 Starting Fresh

► Press the `New` button on the Eclipse toolbar and navigate to "Examples/eMoflon Handbook Examples/" (Fig. 2.1). There are two `Part IV Fresh Start` cheat packages: one for our visual syntax, the other for our textual syntax. They each contain the full `LeitnersLearningBox` metamodel, as well as each method implemented as an SDM and an example instance in "LearningBoxLanguage/instances/". If you need help deciding which syntax to use, refer to Part I, Section 1.

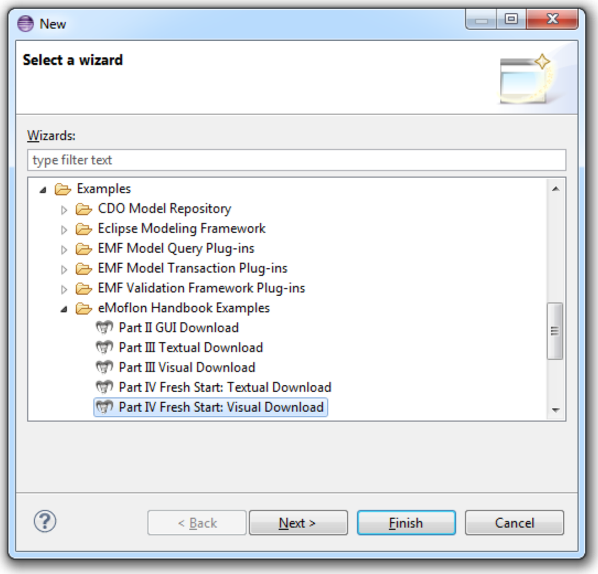

Figure 2.1: Initialize your workspace with your preferred syntax

▶ After loading, if your workspace does not resemble ours in Fig. 2.2, with eMoflon's "Build" icon on the tooolbar and a package explorer with at least two distinct nodes, first switch to the eMoflon perspective by navigating to "Window/Open Perspective/Other..." and choosing "eMoflon" from the list. Then, select the small downward-facing arrow in the upper right corner of the package explorer. "Top Level Elements/Working Sets." To review how these nodes are used to structure our workspace in Eclipse, check out Part I, Section 4.

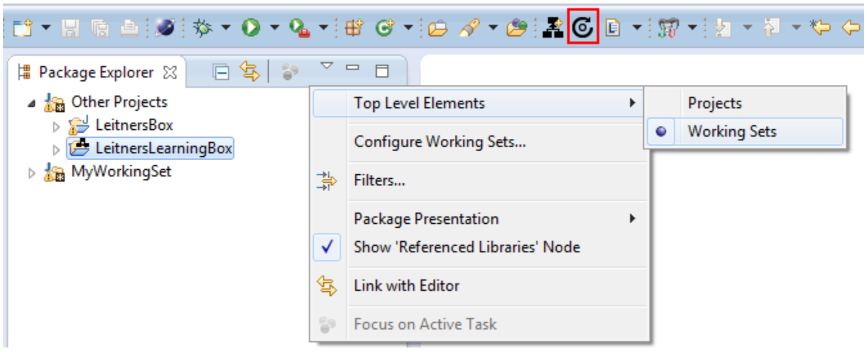

Figure 2.2: Setting your Package Explorer

▶ Fantastic – you now have the source metamodel for your transformation ready to go!

▷ Next [visual]
▷ Next [textual]

2.2 Importing and working with multiple EAPs

Please note that the following instructions on how to properly export and import Enterprise Architect (EA) files are *not* an eMoflon-exclusive feature. We have included them here as part of our handbook as getting this right is crucial for working with eMoflon, especially when working with TGGs. The main problem is that, as far as we know, EA does not (yet) support referencing model elements in one EAP from another, completely different EAP. This means that all required metamodels have to first be merged in the same EAP before such references can be specified (as required for TGGs).

▶ Press the `new` button in the Eclipse toolbar and navigate to "Examples/eMoflon Handbook Examples/" (Fig. 2.10). Find and select `Part IV Visual Dictionary Language` to copy a new `DictionaryLanguage` metamodel project into your workspace.

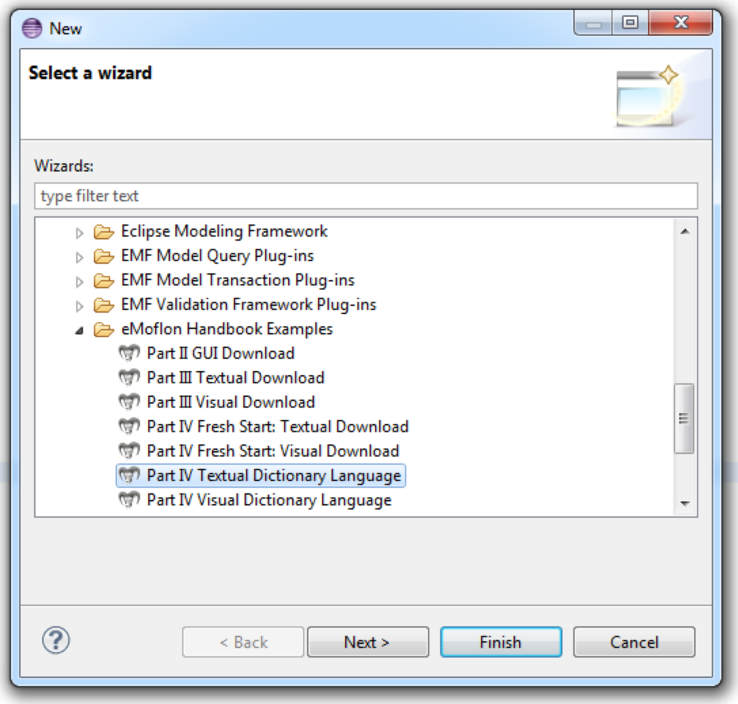

Figure 2.3: Get the visual `DictionaryLanguage` metamodel

▶ If successful, your workspace should resemble Fig. 2.4. Double-click `Dictionary.eap` to open it in EA.

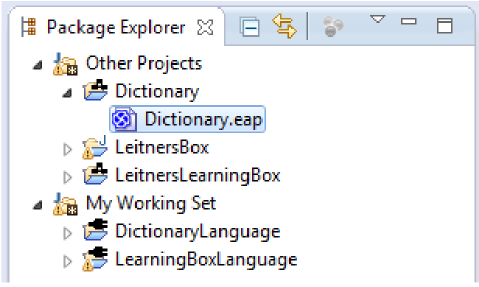

Figure 2.4: `Dictionary` metamodel successfully copied into the workspace

- ▶ The file's project browser should resemble Fig. 2.5. Feel free to inspect the main `DictionaryLanguage` diagram until you're familiar with the metamodel. Our work will be focused on the `Dictionary` and `Entry` classes. You'll be able to see that dictionaries can be assigned unique `EString titles`, and each entry will have some sort of `content` matched with one of three difficulty `levels`.

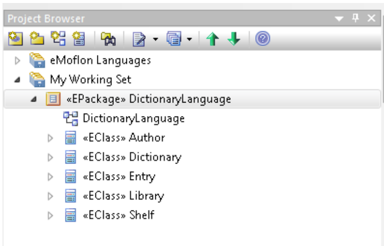

Figure 2.5: The `DictionaryLanguage` metamodel structure

- ▶ It should be said that while you are able to simply copy and paste packages between multiple EAPs (i.e., copy `<<EPackage>>Dictionary-Language` into the `MyWorkingSet` root note `LeitnersLearningBox.eap`), if any of the copied packages have dependencies on other packages, it cannot be done so easily. All links would be destroyed!

- Therefore, to properly migrate the `DictionaryLanguage` package, right-click on the EPackage root, navigate to "Import/Export" and select `Export Model to XMI...` (Fig. 2.6). Alternatively, you can select the root in the project browser and press `Ctrl + Alt + E`.

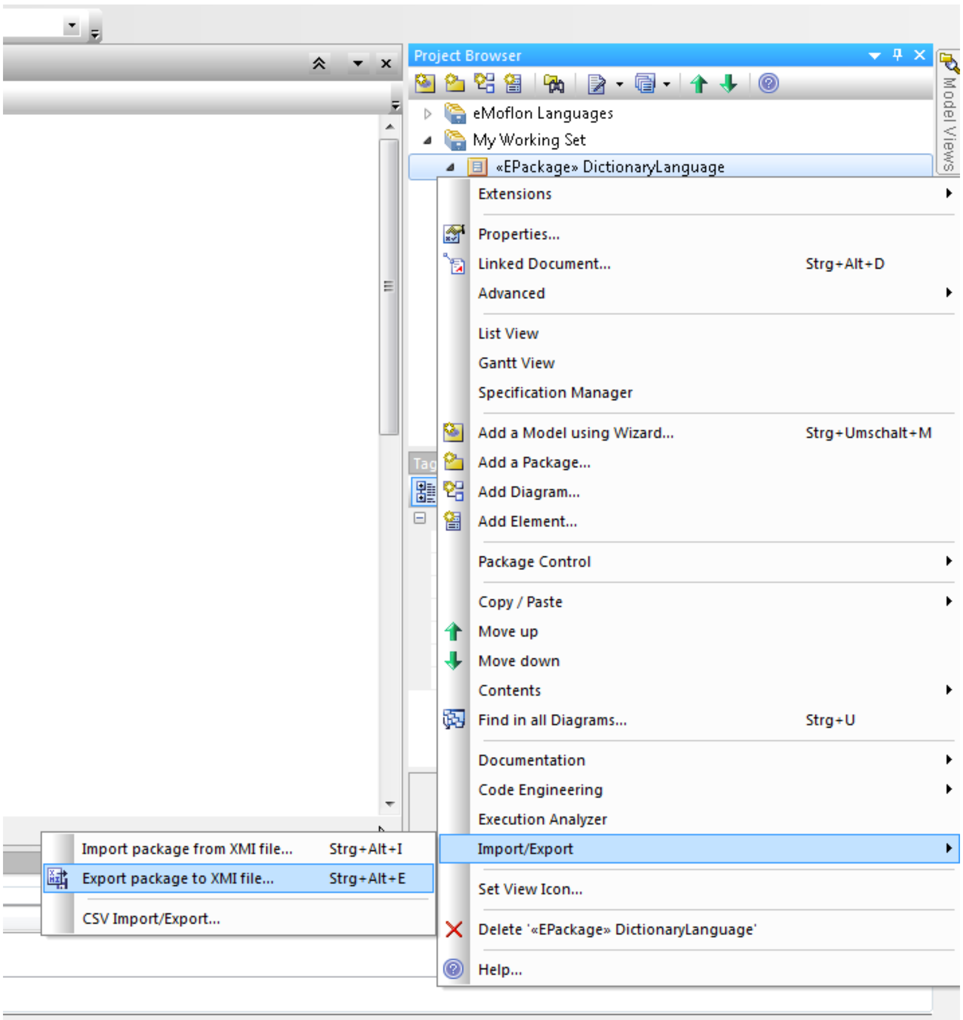

Figure 2.6: Starting the export process in EA

- Switch the export type to `XMI 2.1` in the dialogue and save the file somewhere easily accessible. Press export, and close the window once the small green bar appears (Fig. 2.7).

- Go back to Eclipse and open `LeitnersLearningBox.eap`. Right-click on `My Working Set` and navigate to "Import Model from XMI..."

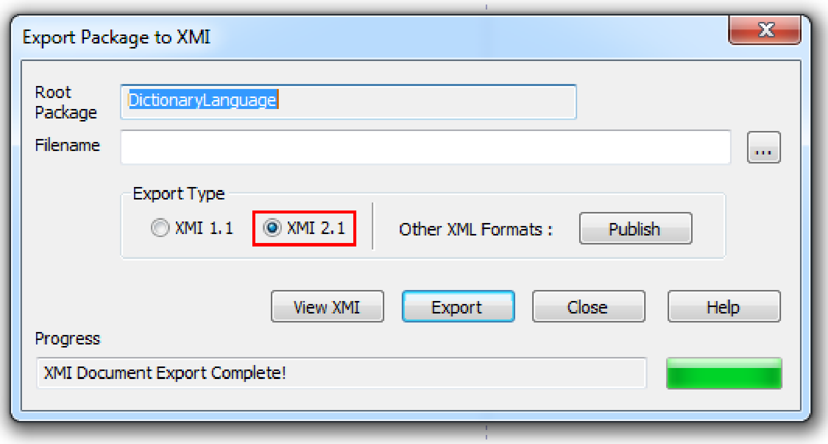

Figure 2.7: Exporting the metamodel to a file

- ▶ Find the `.xmi` file you just saved and press `import`. Press `OK` in the confirmation dialogue. Your project browser should now resemble Fig. 2.8, with both metamodels in the same working set, in the same EAP.

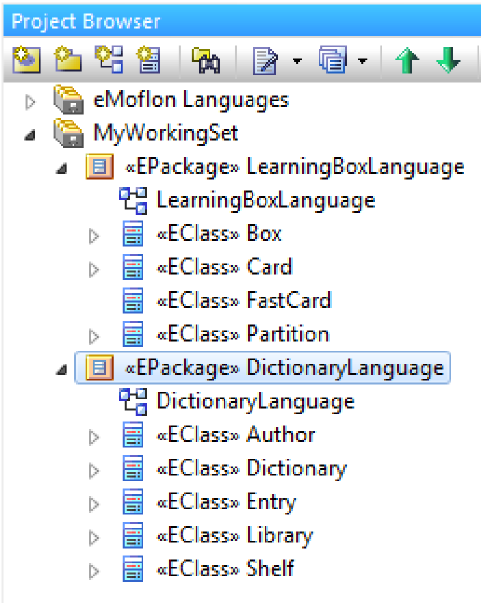

Figure 2.8: The TGG metamodels successfully included in one EAP

- Confirm the import by validating[5] (Fig. 2.9) and exporting the dual-metamodel project to Eclipse, refreshing `LeitnersLearningBox` to rebuild your workspace.

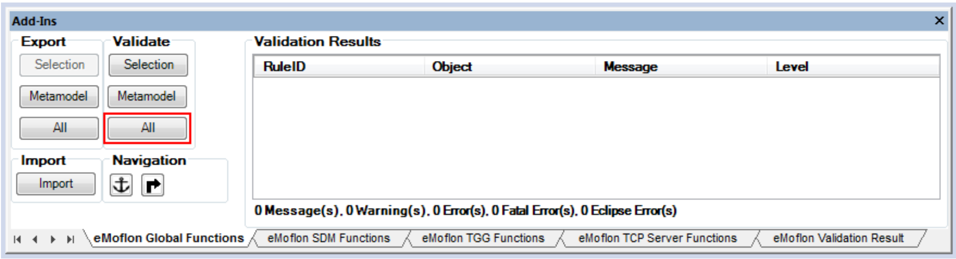

Figure 2.9: No validation errors for `LeitnersLearningBox`

- That's it! You now have the second metamodel for your transformation prepared, and are ready to start specifying your TGG rules.

[5]To review the details of how to use the eMoflon control panel, read Section 2.8 from Part II

▷ Next

2.3 Working with multiple MOSL projects

- ▶ Press the `new` button on the Eclipse toolbar and navigate to "Examples/eMoflon Handbook Examples/" (Fig. 2.10). Find and select `Part IV Textual Dictionary Language` to copy a new `DictionaryLanguage` metamodel project into your workspace.

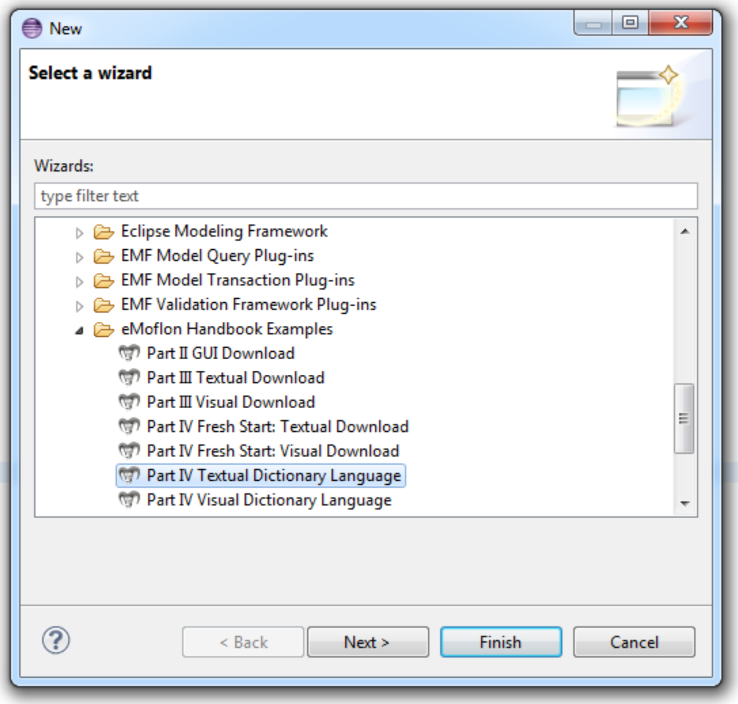

Figure 2.10: Get the textual Dictionary project

- ▶ If successful, your workspace should resemble Fig. 8.2. It would be a good idea to inspect the metamodel until you feel comfortable with what you'll be working with. This transformation will be focusing on the `Dictionary` and `Entry` EClasses.

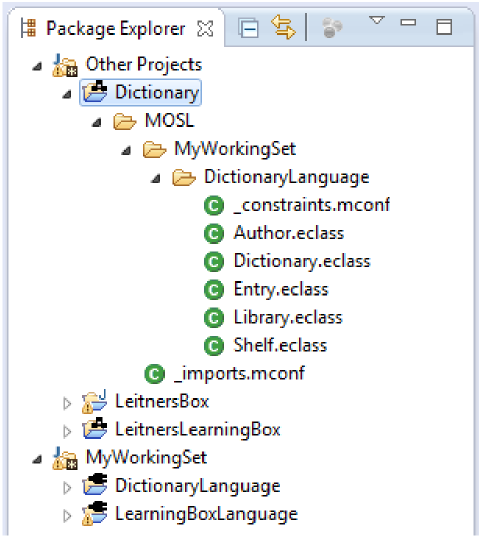

Figure 2.11: `DictionaryLanguage`'s metamodel structure

▶ While you now have your source and target metamodels, `LeitnersLearningBox` is in a different metamodel project (MOSL folder) and still needs to be allowed to access `DictionaryLanguage`. Navigate to "LeitnersLearningBox/MOSL/MyWorkingSet," open `_imports.mconf` and add the statement `import Dictionary` (Fig. 2.12).

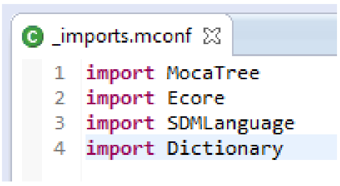

Figure 2.12: Importing `Dictionary` into the learning box

▶ Save and rebuild `LeitnersLearningBox` by pressing "Build (without cleaning)" on the toolbar. You're now ready to start your TGG!

3 Creating your TGG schema

Now that the necessary source and target metamodels are in the same workspace, there are several different ways to begin specifying a TGG. We're going to start with modeling the correspondence component of the 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*.

Link Metamodel
Correspondence Types

While the link metamodel is technically a standard metamodel, eMoflon uses a slightly different naming convention and concrete syntax to represent it. The overall metamodel triple consisting of the relevant parts of the source, link, and target metamodels is called a *TGG schema*.

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.

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

▷ `Next [visual]`
▷ `Next [textual]`

3.1 Visual TGG Schema

- ▶ With `LeitnersLearningBox.eap` open in EA, add a new package to `MyWorkingSet` model root. Name it `LearningBoxToDictionary-Integration` (Fig. 3.1).

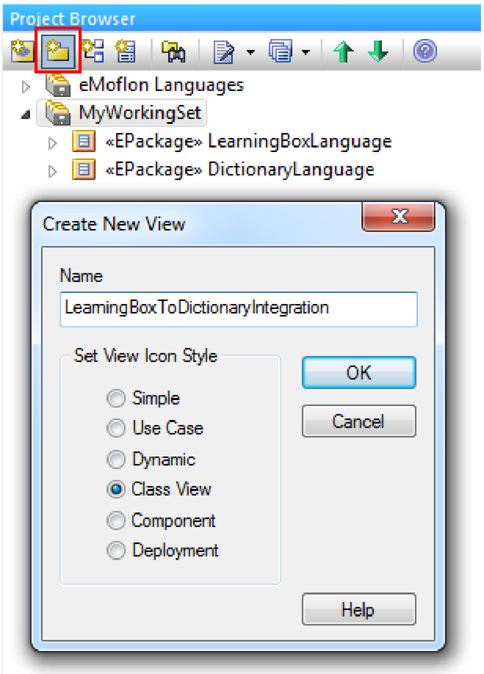

Figure 3.1: Create a new TGG integration package

- ▶ Create a new `TGG Schema` diagram in the new package (Fig. 3.2). The diagram type indicates to EA that the new package is a TGG Project.

- ▶ A dialogue should pop up asking to set the source and target projects of the TGG. Set `LearningBoxLanguage` as the source and `Dictionary-Language` as the target and affirm with `OK` (Fig. 3.3).

- ▶ The structure of your TGG project should now resemble Fig. 3.4. Please note that a subpackage `Rules` and underlying diagram with the same name are also generated.

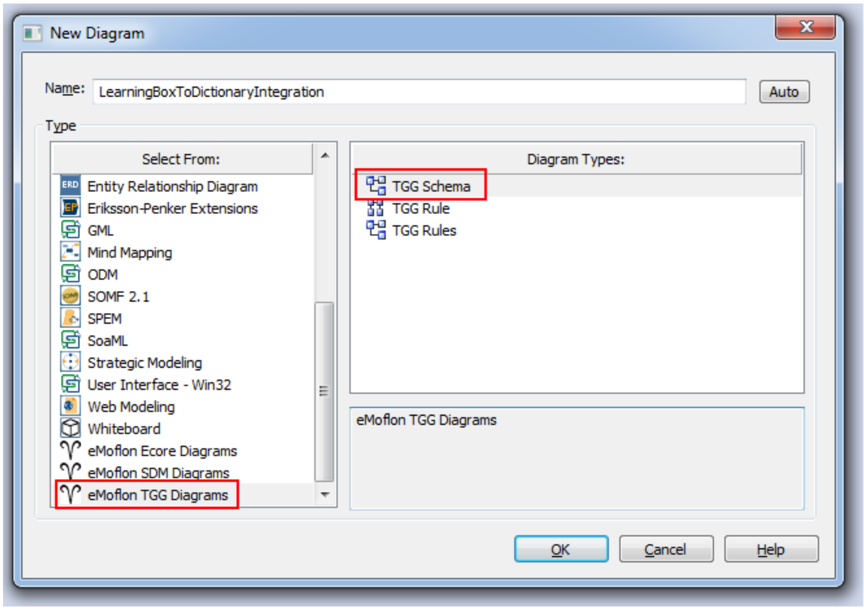

Figure 3.2: Choose TGG Schema as your diagram type

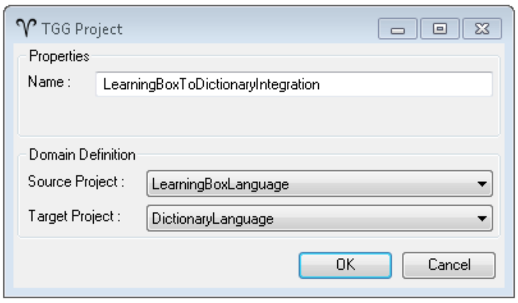

Figure 3.3: Set the source and target projects for the TGG project

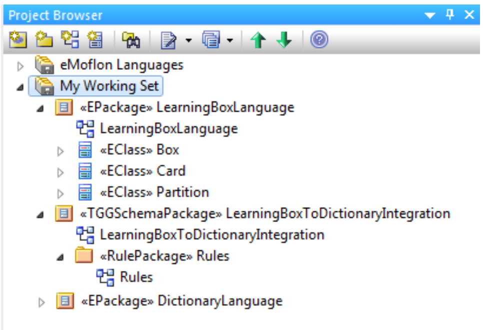

Figure 3.4: Initial structure of a new TGG project

▶ Now it's time to reference classes from the source and target projects in the TGG project to declare the first *correspondence type* between them. Confirm the new TGG schema diagram is open in the editor, then hold `Ctrl` and drag-and-drop the `Box` class from `LearningBox-Language` into the window. Paste the class as a simple `link` into the diagram (Fig. 3.5). For reference, each attribute and operation is included in the diagram.[6]

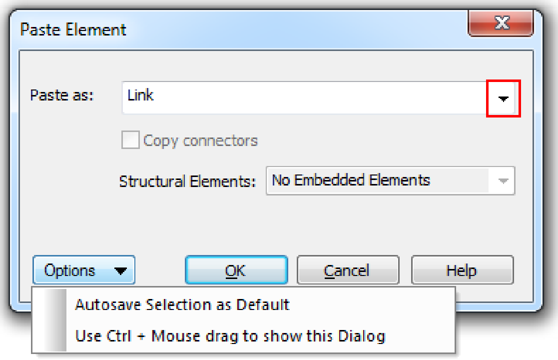

Figure 3.5: Copying an element as a simple link

▶ Note that you are able to set `Autosave Selection as default`. We'll need to switch drag types several times during this part, so it's best to leave this unchecked if you do not want to hold `Ctrl` each time you use the drag-and-drop gesture.

▶ Repeat the action for the `Dictionary` class from `DictionaryLanguage` so that you have a class from each metamodel in the schema.

▶ We can now create a correspondence type! Quick-link from `Box` to `Dictionary`, selecting `Create TGG Correspondence Type` in the context menu (Fig. 3.6).

[6] Take caution: If you press `Ctrl + Delete` to delete the element in this diagram, you will also delete it from its original metamodel package!

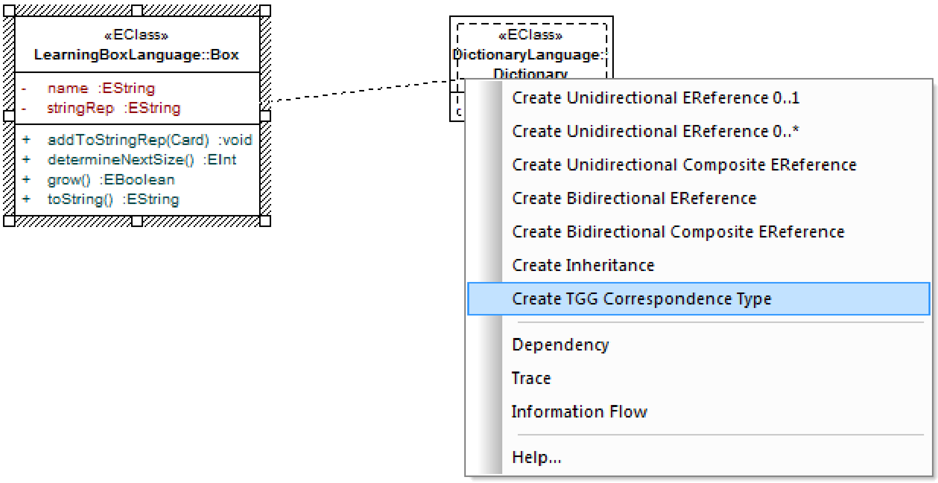

Figure 3.6: Quick-link to create a correspondence type

▶ As you can see, a correspondence type has been created, visualized as a hexagon (Fig. 3.7). It is automatically named `BoxToDictionary` and the references are appropriately named.

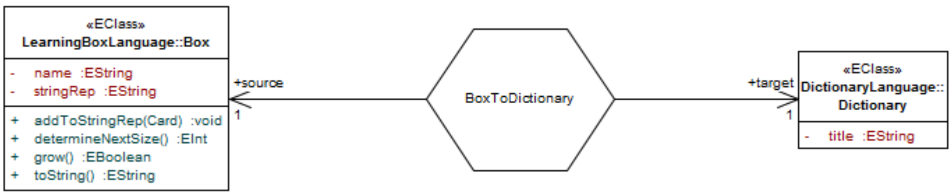

Figure 3.7: An established correspondence type

▶ You've just finished initalizing your TGG schema! To see how this is done with the textual syntax, check out Fig. 3.10 in the next section.

▷ Next

3.2 Textual TGG Schema

▶ Within the learning box metamodel folder, right-click on `MyWorkingSet`, and navigate to "New / TGG" (Fig. 3.8).

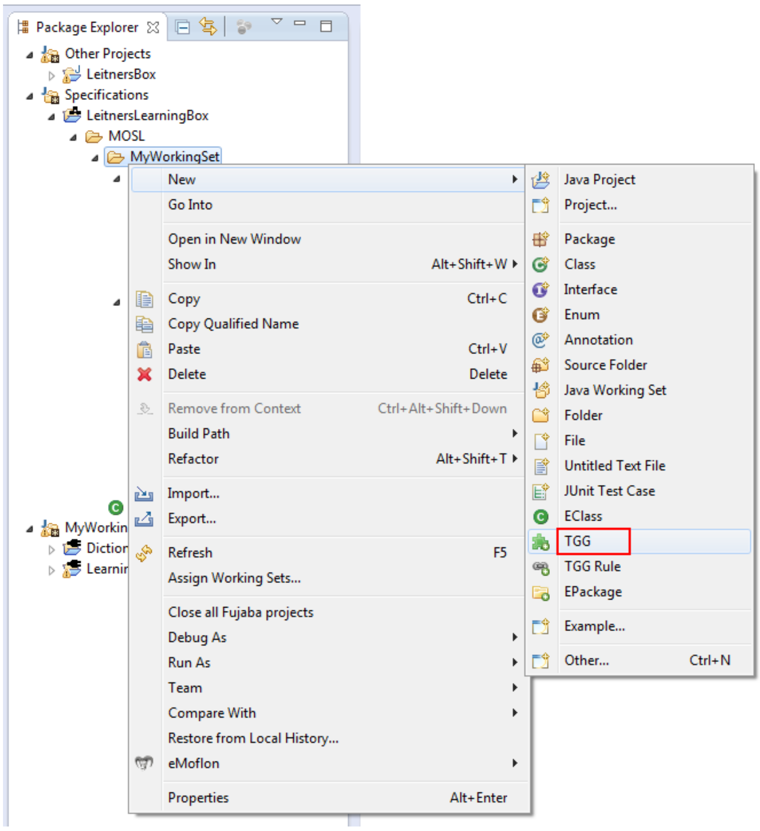

Figure 3.8: Creating a new TGG schema

▶ Name the TGG `LearningBoxToDictionaryIntegration`, setting the source as `LearningBoxLanguage`, and target as `DictionaryLanguage` (Fig. 3.9).

▶ A new TGG `schema` file should now be active in the editor! This is the *TGG Schema* which declares each *correspondence type* as an `integration class`. Press `Ctrl + spacebar` and use the auto completion to generate a new integration class.

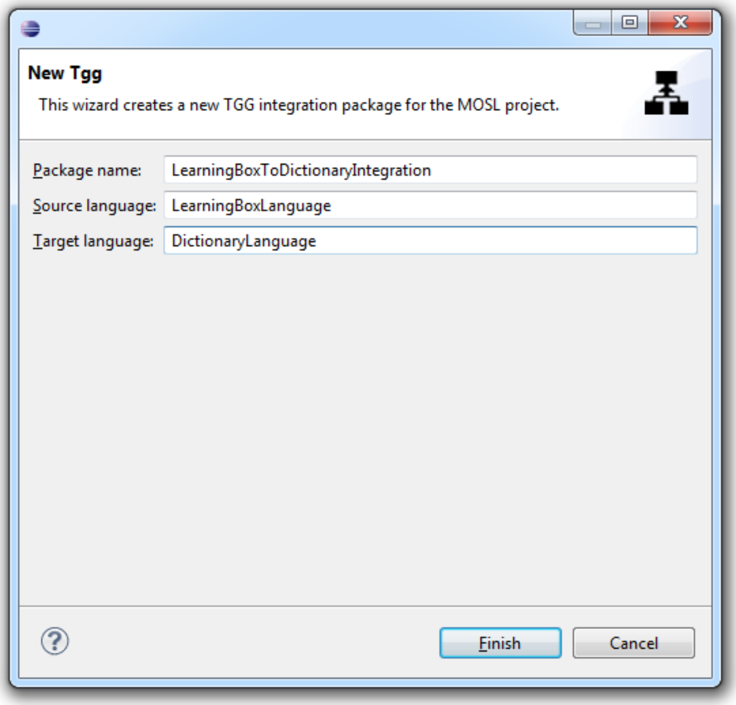

Figure 3.9: Setting your `source` and `target` metamodels

▶ Note that when using a template, you can press `tab` to cycle through each element. Name the class `BoxToDictionary`, and list the source as `Box` and target as `Dictionary` (Fig. 3.10).

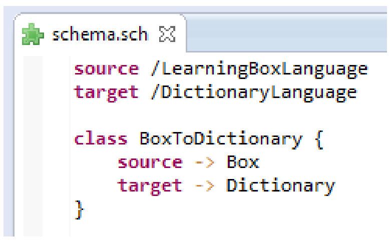

Figure 3.10: Creating a correspondence type

▶ Believe it or not, that's all you need for your first correspondence type! Your schema is now complete with connections to your `source` and `target` metamodels via a *link* metamodel. To see the equivalent structure in the visual syntax, check out Fig. 3.7 from the previous section.

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 SDM 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 SDMs, they do not delete or modify any existing elements. In other words, TGG rules are *monotonic*. *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.

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 `cards`. 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 `cards` 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`.

▷ Next [visual]
▷ Next [textual]

4.1 Visual TGG Rules

EA distinguishes the different elements of a TGG rule with distinct visual and spatial clues. Correspondence elements, for example, are depicted as hexagonal boxes, while attribute constraints are depicted as notes, referencing the relevant object variables.

4.2 BoxToDictionaryRule

▶ In EA, open the `Rules` diagram of your TGG project we pointed out earlier. Create your first rule by either hitting the `spacebar` and selecting `Rule` from the context menu, or performing a drag-and-drop of the `Rule` item from the TGG toolbox to the left of the diagram window (Fig. 4.1). Press `Alt + Enter` to raise its `Properties` dialogue, and update its name to `BoxToDictionaryRule`.

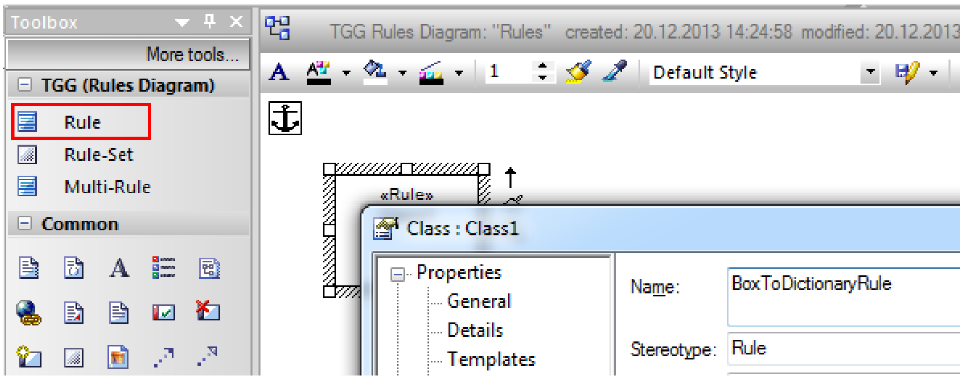

Figure 4.1: Creating a TGG rule

▶ Double-click the element to open its rule diagram. Drag-and-drop `Box` from the project browser into the diagram once again, choosing to paste the element as an `instance`.[7] The `name` and `binding operator` should already be set to `box` and `create`. Repeat the action to create an instance of `Dictionary`.

[7] If the 'Paste Element' dialogue doesn't appear, hold `Ctrl` while dragging and dropping and confirm you haven't selected the autosave option under `options`.

▶ Quick-link from `box` to `dictionary` this time to create a TGG correspondence *link*. To keep things simple and self-explanatory, keep the default name `boxToDictionary` and select the `BoxToDictionary` correspondence type from the drop-down list (which you declared in the schema).

Believe it or not, with just this link, our rule *already* creates a `Box`, `Dictionary`, and correspondence link between them at the same time! Unfortunately, this only creates the objects, and doesn't relate any of their attributes. Why don't we try to connect the `name` of the `box` to the `title` of the dictionary so that they always match?

For this, we use *attribute constraints*.[8] When used in TGG rules, attribute constraints provide a bidirectional and high-level solution for attribute manipulation by solving a *constraint satisfaction problem* (short CSP). In this case, our CSP instance will ensure that `box.name` and `dictionary.title` remain consistent. *CSP*

▶ Following a similar process as creating a new `Rule`, either hit the `spacebar` or use the toolbox to create a `CSP instance` (Fig. 4.2). A `Define CSP` dialog will pop-up. You can open this dialog anytime by double-clicking the CSP note.

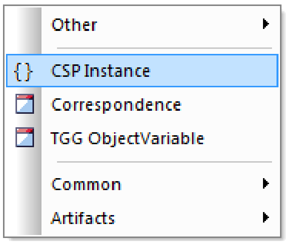

Figure 4.2: CSP instance from the toolbox

▶ You'll notice a pre-populated list of available constraints. Choose `eq` (representing 'equals') and double-click each of the `Value` fields to specify the `a` and `b` values as depicted in Fig. 4.3. Press `Add` to save the constraint, then `OK` to affirm and close the window.

▶ Your rule should now resemble Fig. 4.4, where the new arrows indicate the constraint dependencies.

[8] First defined in Part III, Section 4

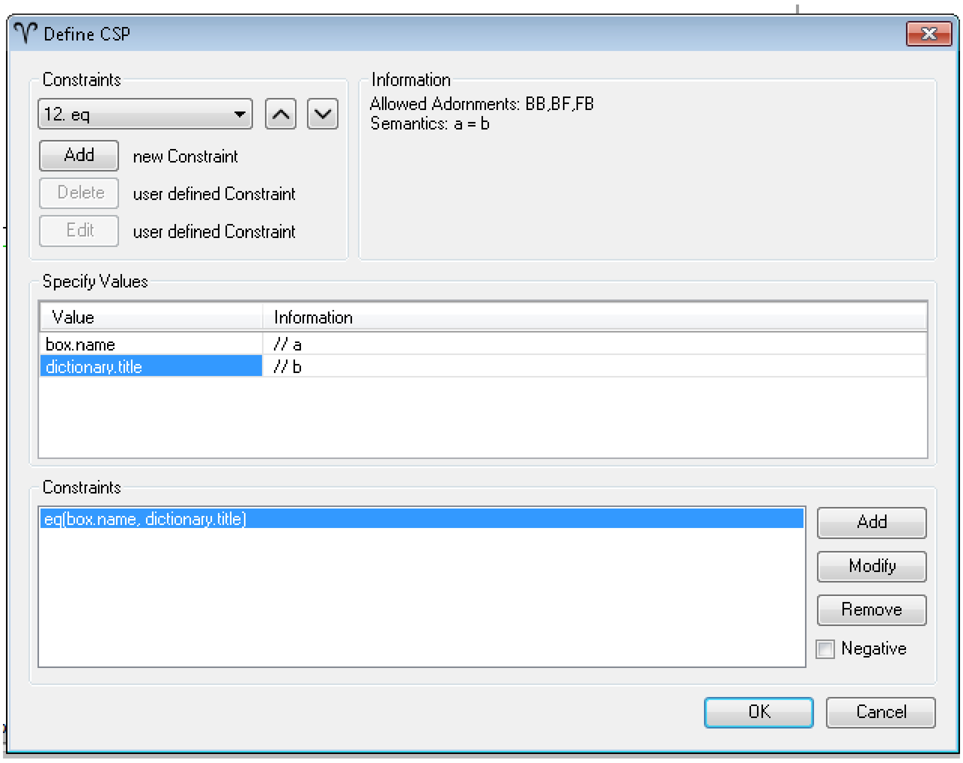

Figure 4.3: Completing the constraint

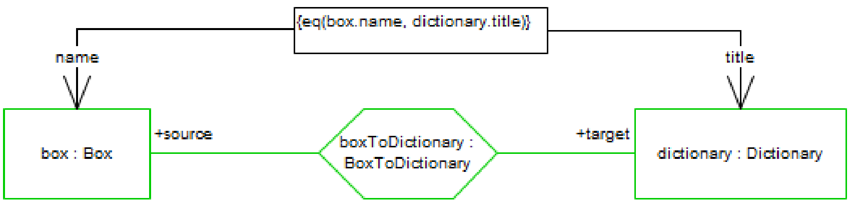

Figure 4.4: A TGG rule with an attribute constraint

Our first TGG rule is not yet complete. Our goal is to transform a `Box` into a `Dictionary`, so we still need to create the initial structure of the learning box. In contrast to the rather simple dictionary, where `Dictionary` is a direct container for every `Entry` object, we have to create a number of connected `Partitions` to hold the `Cards`.

▶ Given that there are three valid difficulty levels for every `Entry` we create three `Partition` object variables, complete with appropriate link variables that satisfy the Leitner's Box rules (the `next`, `previous`, and `box` references). Your TGG rule should come to resemble Fig. 4.5.[9]

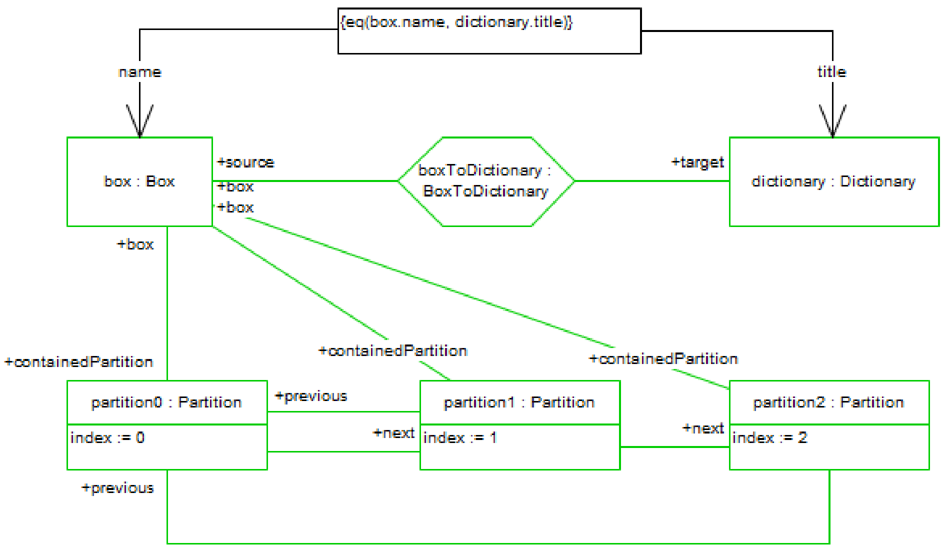

Figure 4.5: Complete TGG rule diagram for `BoxToDictionaryRule`

Fantastic work! The rule of our transformation is complete! If you are in hurry, you can jump ahead and proceed 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.

[9]To review how to set *inline* object attribute constraints (e.g., `index := 0`), review Part III, Section 4

4.3 CardToEntryRule

The next goal is to be able to handle `card` and `entry` elements. The challenge is that it will require a strict pre-condition – you should not be able to transform these child elements unless certain structural conditions 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! Luckily, eMoflon has a cool feature in its visual syntax to help with this. We can go to any existing rule and *derive* a new one from it. The benefits of this may not be so obvious with this small example, but this could potentially be a real time-saver in a large project.

- ▶ First confirm that your eMoflon control panel window is open in the `BoxToDictionaryRule` diagram. Then hold `Ctrl` and select `box`, `boxToDictionary`, and `dictionary` simultaneously.
- ▶ Switch to the `eMoflon TGG Functions` tab on the control panel and press `Derive` (Fig. 4.6). In the dialogue that appears, enter `CardToEntryRule` as the name of the rule, and press `OK`. The new rule will automatically open in a new window.

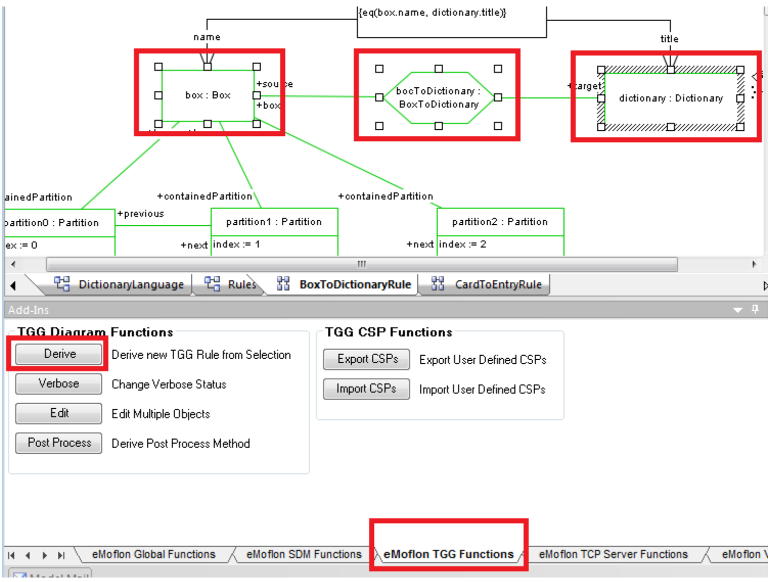

Figure 4.6: Derive a new rule from an existing one

- ▶ Add a context (black) instance of `Partition` to the new rule, and link it to `box`.

- Add green instances of `Card` and `Entry` to the new rule, and link them to their respective `partition` and `dictionary` elements.

- Quick-link from `card` to `entry` and create another TGG correspondence link. You'll notice that the `select correspondence link` drop-down menu is empty – we haven't defined one between these types yet in the schema. Luckily, we're able to create one here on-the-fly. Select `Create New Correspondence Type` and name it `CardToEntry` (Fig 4.7).

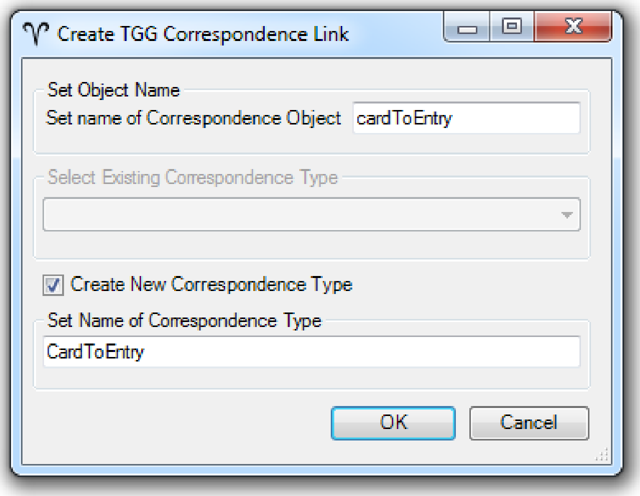

Figure 4.7: Create a new correspondence type on-the-fly

- Your diagram should now resemble Fig. 4.8. We're not done yet though – we still need to handle attributes!

We must create a series of constraints in order to specify how relevant attributes should be handled. Let's first define a construct for every `entry-.content`, `card.back`, and `card.face` EString values so that it's easy to (temporarily) persist the values during the transformation. This will help us figure out how we should combine the front and back of each `card` as a single `content` attribute and, in the opposite direction, help to separate the contents so that they may be split into `card.back` and `card.face`.

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

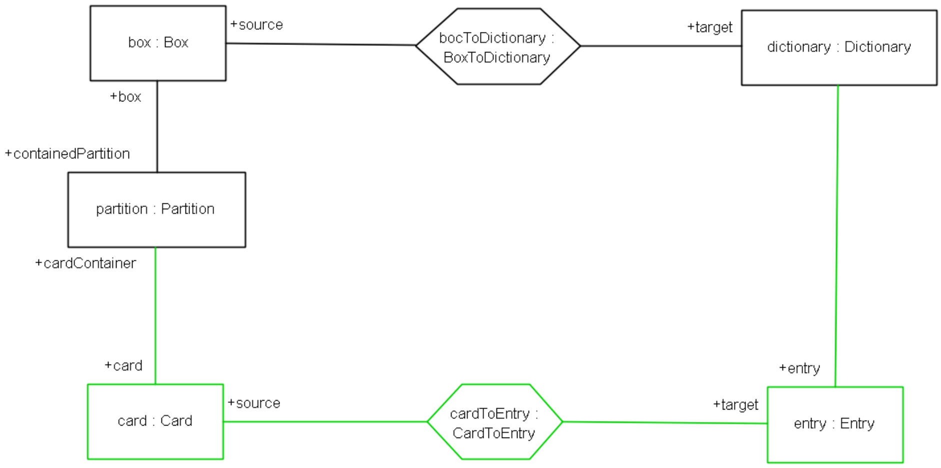

Figure 4.8: `CardToEntryRule` without attribute manipulation

▶ We can now define three *attribute constraints* to implement this. Luckily, we have two predefined constraints, `addPrefix` and `concat` to help us. Use the toolbox again to create a new TGG constraint, and add the following to your diagram:

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

Your rule should now resemble Fig. 4.9, where "Question" and "Answer" are EString literals, `word` and `meaning` are temporary variables, and `card.face`, `card.back`, and `entry.content` are attribute expressions (this should be familiar from SDM story patterns).

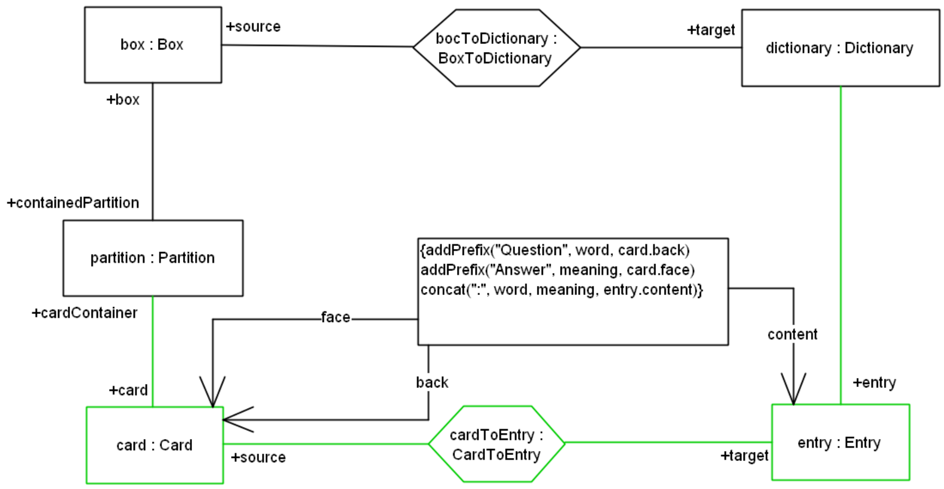

Figure 4.9: Attribute manipulation for `card` and `entry`

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

- Add one more constraint to your diagram but, instead of choosing a predefined constraint, click “Add” just below the drop-down menu to create a custom one. Name it `IndexToLevel`, and enter the values given in Fig. 4.10.

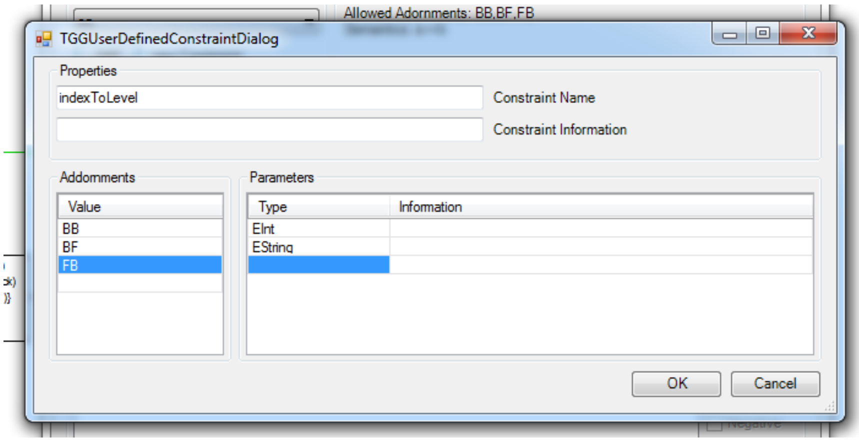

Figure 4.10: Creating an unique constraint

- Please note that this is just a specification of a custom constraint – we still need to implement in Java! Since we're so close to finishing this TGG rule however, let's finish and export what we've made to Eclipse before doing so. We'll explain the exact meaning of the mysterious adornments and parameters of the constraint in a moment. For now, just make sure you enter the exact values in Fig. 4.10.
- Save the new constraint, then select it from the drop-down menu in the TGG Constraint Dialog. Enter `partition.index` as an `EInt` value, and `entry.level` as an `EString`.
- Your completed TGG rule should resemble Fig. 4.11. Great work! All that's left to do is implement the `IndexToLevel` constraint, and give your transformation a test run.
- Check out Fig. 4.14 in Section 4.3 to see how `BoxToDictionaryRule` is specified in the textual syntax, or Fig. 4.19 in Section 4.4 for the `CardToEntryRule`.

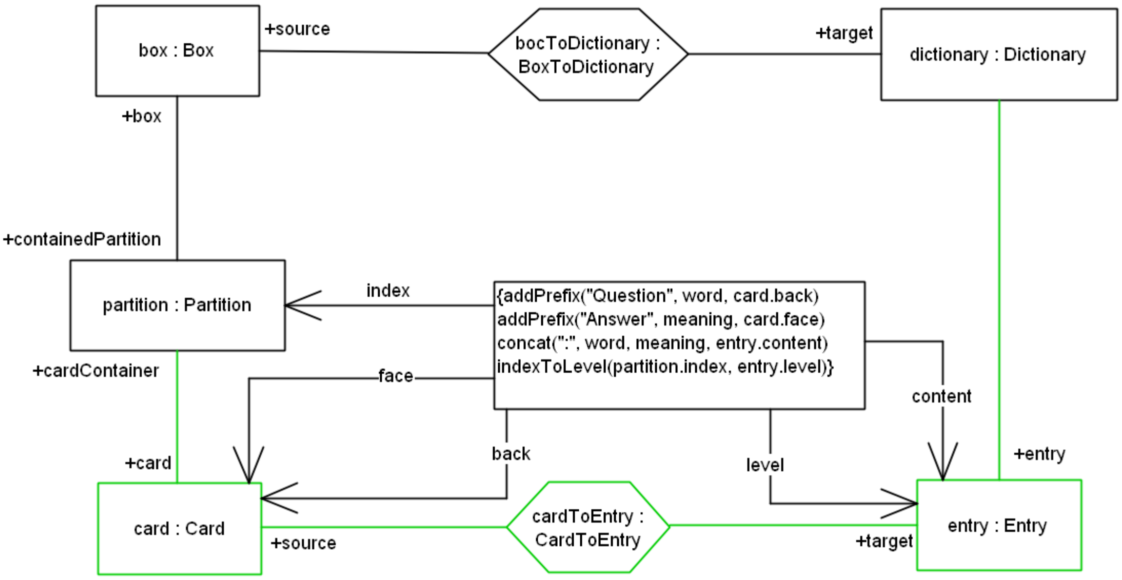

Figure 4.11: `CardToEntryRule` with complete attribute manipulation

4.4 Textual TGG Rules

Rules in the texual syntax are clearly separated into three primary scopes – source, correspondence, and target – along with a final scope for constraints, which can manipulate attributes based on the transformation direction.

4.5 BoxToDictionaryRule

- You may have noticed that a `Rules` folder was created and included in the TGG package when you first created it. Create your first TGG rule by right-clicking on this folder and navigating to "New/TGG Rule." Name it `BoxToDictionaryRule`, and confirm the opened file in the editor window.

- Let's first establish the `source` and `target` scopes. Given that this is the first rule to be applied in a transformation, we can assume there is no context to work with, so each of our objects will need to be set to 'green' (create). In the `source` scope, create a `box` of type `Box`. Similarly, in the `target` scope, create a `dictionary` of type `Dictionary`. Your rule should now resemble Fig. 4.12.

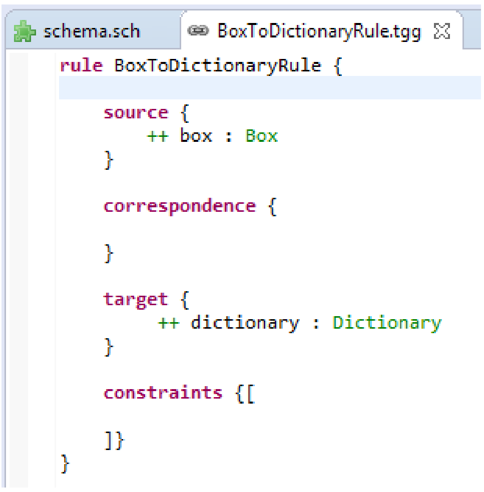

Figure 4.12: Creating source and target objects

- Now we can create our first TGG correspondence link! In the `corr-espondence` scope, enter

```
++ box <- boxToDictionary :  BoxToDictionary -> dictionary
```

- ▶ Please note that this statement creates *one* link, named `boxToDictionary`, of type `BoxToDictionary` which was declared in the schema.

If this rule were to be run at this point, as-is, it would successfully create a single `Box` and `Dictionary`! Besides the correspondence link however, these items have nothing in common. Let's try connecting the `name` of `box` to the `title` of `dictionary` with an *attribute constraint.* In TGG rules, attribute constraints provide a bidirectional and high level solution for attribute manipulation. In addition to the basic math constraints such as addition (add), subtraction (sub), divide, max, multiply, and smallerOrEqual, we have some pre-existing string constraints we can use. These include stringToNumber, concatenate (concat), addPrefix, addSuffix, and equals (eq).

- ▶ In the `constraints` scope, write:

```
eq(box.name, dictionary.title)
```

Your rule should now resemble Fig. 4.13.

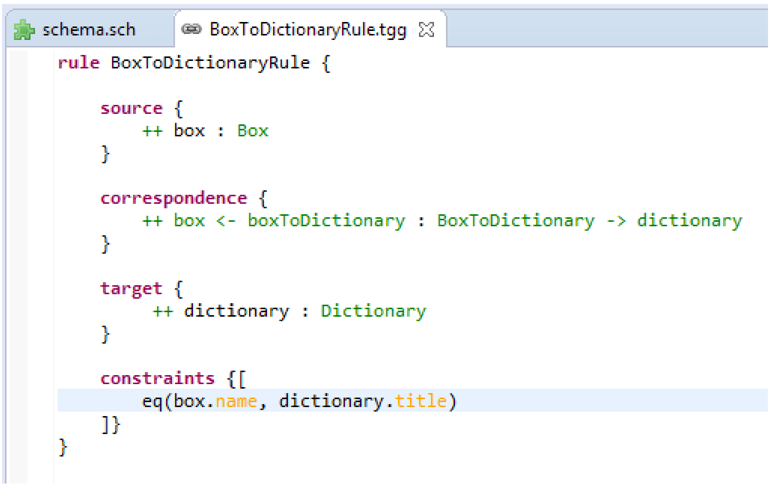

Figure 4.13: Creating a correspondence link and adding attribute constraints

We're nearly done, but what's missing from this first rule? We've created the primary container structures for the `target` and `source`, and knowing that entires can be stored directly in `dictionary`, we know the target scope can remain empty. `cards` however must be contained within `partitions`, so our source scope is till incomplete!

- Given that there are three difficulty `levels` for each dictionary `entry`, create three `partitions` in the box that will correspond to the levels: `partition0`, `partition1`, and `partition2`.

- Complete the rule by setting both the individual `index` values and appropriate `containedPartition`, `next` and `previous` link variables so that your rule matches Fig. 4.14.

BoxToDictionaryRule.tgg

```
rule BoxToDictionaryRule {
    source {
        ++ box : Box {
            ++ -containedPartition-> partition0
            ++ -containedPartition-> partition1
            ++ -containedPartition-> partition2
        }

        ++ partition0 : Partition {
            partition0.index := 0
            ++ -next-> partition1
        }

        ++ partition1 : Partition {
            partition1.index := 1
            ++ -next-> partition2
            ++ -previous-> partition0
        }

        ++ partition2 : Partition {
            partition2.index := 2
            ++ -previous-> partition0
        }
    }

    correspondence {
        ++ box <- boxToDictionary : BoxToDictionary -> dictionary
    }

    target {
     ++ dictionary : Dictionary

    }

    constraints {[
        eq(box.name,dictionary.title)
    ]}
}
```

Figure 4.14: The completed `BoxToDictionaryRule`

Great work! This rule is now able to transform a `box` into a `dictionary` and vice versa. Unfortunately, it will only be able to handle completely empty boxes and dictionaries – you can see we haven't provided any additional handling for `Card` or `Entry` items. If you're in a hurry, feel free to jump ahead to Section 4: TGGs in Action, to try executing this rule anyway. Otherwise, the next rule we create will integrate itself with `BoxToDictionaryRule` to take care of this.

4.6 CardToEntryRule

- ▶ Analogously to how you began the previous rule, return to the TGG schema and create a second correspondence type called `CardToEntry`, with a `Card` source and `Entry` target. Your updated file should now resemble Fig. 4.15.

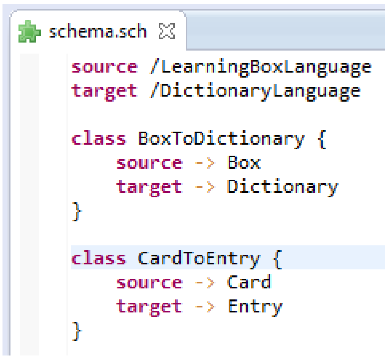

Figure 4.15: Updating the schema

- ▶ Right-click on the `Rules` folder again, and create the `CardToEntryRule`.

One of the key differences between this rule and the last is that `CardToEntryRule` should only be invoked with a certain context i.e., this will only be used if a pre-existing `partition` has `card` elements that need to be transformed into entries in an established `dictionary`. In terms of MOSL, this means there will be both 'black' and 'green' elements.

- ▶ To begin, create three object variables in the `source` scope: `box`, `partition0`, and `card`. Which ones are already known from the context? Which element still needs to be made? Your rule should come to resemble Fig. 4.16.

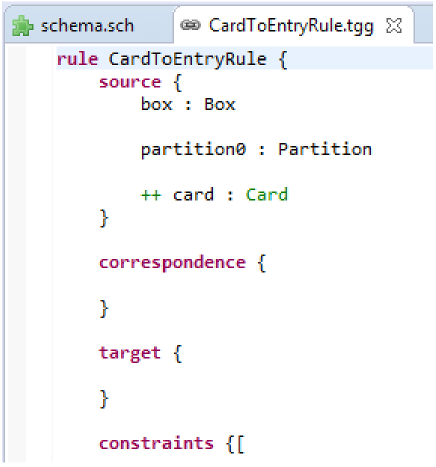

Figure 4.16: The source language with both 'black' and 'green' elements

- ▶ Similarly, in the `target` scope, you can demand a 'black' `dictionary:Dictionary` element from the context, but will need to create a new entry object via `++ entry:Entry`.

- ▶ With all of our objects now created, we can complete the `correspondence`. Our contextual `box` and `dictionary` objects must be connected via the same `boxToDictionary` link as declared in `BoxToDictionaryRule`, but a second link needs to be created between `card` and `entry`. Use the correspondence type from the updated schema and write:

```
++ card <- cardToEntry :  CardToEntry -> entry
```

- ▶ Finally, let's make sure the transformation handles the `card` and `entry` attributes correctly. Complete each of your `box`, `partition0`, and `dictionary` object variable scopes with the relevant references until your rule matches Fig. 4.17.[10]

[10] Don't forget that eMoflon's type completion can help you establish references here; Press `Ctrl + spacebar` after writing `->` for a list of available link variables from the relevant `EClass`.

CardToEntryRule.tgg

```
rule CardToEntryRule {
    source {
        box : Box

        partition0 : Partition {
            -box-> box
            ++ -card-> card
        }

        ++ card : Card
    }

    correspondence {
        box <- boxToDictionary : BoxToDictionary -> dictionary

        ++ card <- cardToEntry : CardToEntry -> entry
    }

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

        ++ entry : Entry

    }

    constraints {[

    ]}
}
```

Figure 4.17: Rule with all object variables

Now let's establish the necessary constraints to handle the relevant content attributes of card and entry. We'll need to first decide on some common variables and syntax between card.face, card.back, and entry.content so that we can combine each side of a card into one content value, or split each entry into a question and answer.

- ▶ Let's define the syntax for entry.content as <word>:<meaning>, card-.back as Question:<word>, and card.face as Answer:<meaning>.

- ▶ Using the pre-existing String attribute constraints addPrefix and contact to edit your constraint scope until it resembles Fig. 4.18.

CardToEntryRule.tgg

```
rule CardToEntryRule {
    source {
        box : Box

        partition0 : Partition {
            -box-> box
            ++ -card-> card
        }

        ++ card : Card
    }

    correspondence {
        box <- boxToDictionary : BoxToDictionary -> dictionary

        ++ card <- cardToEntry : CardToEntry -> entry
    }

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

        ++ entry : Entry

    }

    constraints {[
        addPrefix("Question ", word, card.back)
        addPrefix("Answer", meaning, card.face)
        concat(":", word, meaning, entry.content)
    ]}
}
```

Figure 4.18: `CardToEntryRule` with its required attribute manipulation

We're not quite done – we need to add *one* more constraint. Given that we have three partitions, and three difficulty levels for each `Entry`, why don't we have the transformation assign a level based on whatever partition a `card` is found in? Hard cards, for example, are more likely to be found in the first partition (due to being shifted backwards from wrong guesses), while easy cards will be near the end. As you can imagine, there is no constraint type currently existing in eMoflon to manage this. We must define our own!

▶ Add the following declaration to the `constraint` scope:

```
indexToLevel[BB,BF,FB](EInt, EString)
```

We will discuss what each of the options mean in a moment.

▶ You can now invoke your rule with an `indexToLevel(partition0.-index, entry.level)` statement immediately below the declaration. Your completed `CardToEntryRule` should now resemble Fig. 4.19, where every new `card` will have an equivalent (and consistent) `entry` element.

▶ Awesome work! If you haven't already, save the file and confirm the MOSL parser hasn't raised any errors. Press `Build (Without Cleaning)` and admire your two TGG transformation rules.

▶ To see how `BoxToDictionaryRule` is implemented in the visual syntax, check out Fig. 4.5 from Section 4.1. Similarly, `CardToEntryRule` is depicted in Fig. 4.11 in Section 4.2.

CardToEntryRule.tgg

```
rule CardToEntryRule {
    source {
        box : Box

        partition0 : Partition {
            -box-> box
            ++ -card-> card
        }

        ++ card : Card
    }

    correspondence {
        box <- boxToDictionary : BoxToDictionary -> dictionary

        ++ card <- cardToEntry : CardToEntry -> entry
    }

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

        ++ entry : Entry

    }

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

        indexToLevel[BB,BF,FB](EInt,EString)
        indexToLevel(partition0.index,entry.level)
    ]}
}
```

Figure 4.19: Completed `CardToEntryRule` scopes

4.7 Implementing IndexToLevel

If everything has been done correctly up to this point, your project should save and build without errors in Eclipse. In fact, there should now be three generated repository projects included in `MyWorkingSet`. We're most concerned with `LearningBoxToDictionaryIntegration`, which implements our TGG and its rules. Expand your folder so it resembles Fig. 4.20.

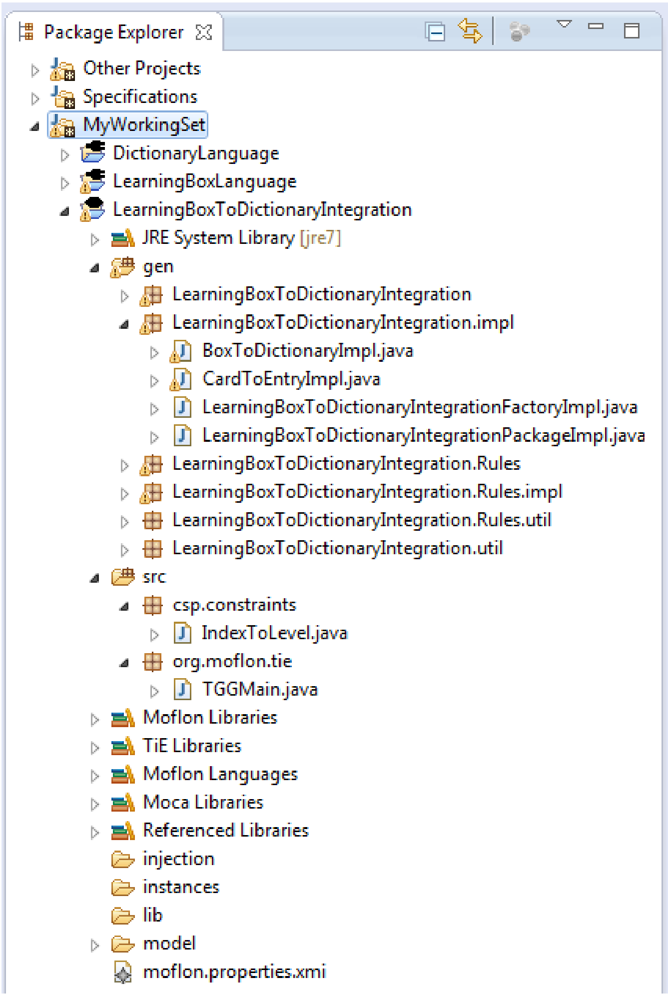

Figure 4.20: Files generated from your TGG

Despite all these files, the TGG isn't yet complete. While we've declared and used our custom `IndexTolevel` attribute constraint, we haven't actually implemented it yet. Let's quickly review the purpose of constraints before we do.

Just like patterns describing *structural* correspondence, *attribute constraints* can be automatically *operationalized* as required for the forward concrete transformations. Even more interesting, a set of constraints might have to be ordered in a specific way depending on the direction of the transformation, or they might have to be checked for pre-existing attributes. Others still might have to set values appropriately in order to fulfill the constraint.

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

In many cases however, a required constraint might be extremely narrow and problem-specific, such as our *IndexToLevel*. There might not be any fitting combination of library constraints to express the consistency condition, so a new attribute constraint type must be declared before its use.

There is a list of *adornments* in the declaration which specify the cases for which the constraint can be operationalized. Each adornment consists of a `B` (bound) or `F` (free) variable setting for each argument of the constraint. It sounds complex, but is 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 indicates 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`.

Note that we decide not to support **FF** as we would have to generate a consistent pair of `index` and `level`. Although this is possible and might even make sense for some applications, it does not in the context of partitions and entries (the pairs are not unique, so which pair should we take? `partition2` set to `beginner`?).

At compile time, the set of constraints (also called *Constraint Satisfaction Problem* (CSP)) 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 constraint. If the CSP cannot be solved, an exception is thrown at compile time.

Now that we have a better understanding behind the construction of attribute constraints, 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 Eclipse's built-in auto-complete feature to help implement the code in Fig. 4.21 to replace the default code.[11]

To briefly explain, the `levels` list contains each `level` at position 0, 1, or 2 in the list, which correspond to our three `Partition.index` attributes. You'll notice that instead of setting 'master' to 2, it has been set to match the first 0 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 guess. 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 the starting position.

In the `solve` method, there is a switch statement based on whichever adornment is currently active. For all cases, `setSatisfied` informs the TGG whether or not the constraint (and by consequence, the precondition of the rule) can be satisfied. For `BF`, it suggests that if a negative partition were to exist, to simply set its index value to 0. Similarly, if there was ever a partition more than 2 (i.e., `partition4`), it would set its index to the highest difficulty level, 2. Otherwise, `BF` simply gets the index of the partition, assigns it so it becomes bound, and terminates. In the final case, where `level` is already known (i.e., transforming an `entry` into a `card`), if the String `level` cannot be matched to any of those in the list, the constraint cannot be fulfilled, and the rule cannot be completed.

[11] Although tempting, we recommend not to copy and paste the contents from your pdf viewer into Eclipse. Invisible characters are likely to be added, and your code might not work.

```java
package csp.constraints;

import java.util.Arrays;
import java.util.List;

import TGGLanguage.csp.Variable;
import TGGLanguage.csp.impl.TGGConstraintImpl;

public class IndexToLevel extends TGGConstraintImpl {

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

        public void solve(Variable var_0, Variable var_1) {
                int index = ((Integer) var_0.getValue()).intValue();
                String level = (String) var_1.getValue();
                String bindingStates = getBindingStates(var_0, var_1);
                switch (bindingStates) {
                case "BB":
                        if (index < 0) {
                                index = 0;
                        } else if (index > 2) {
                                index = 2;
                        }
                        setSatisfied(levels.get(index).equals(level));
                        break;
                case "BF":
                        if (index < 0)
                                var_1.setValue(levels.get(0));
                        else if (index > 2)
                                var_1.setValue(levels.get(2));
                        else
                                var_1.setValue(levels.get(index));
                        var_1.setBound(true);
                        setSatisfied(true);
                        break;
                case "FB":
                        index = levels.indexOf(level);
                        if (index == -1) {
                                setSatisfied(false);
                        } else {
                                var_0.setValue(index);
                                var_0.setBound(true);
                                setSatisfied(true);
                        }
                        break;
                }

        }
}
```

Figure 4.21: Implementation of our custom `IndexToLevel` constraint

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[12] 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`. Make sure you persist the instance in `LearningBoxToDictionaryIntegration/instances/` (Fig. 5.1).

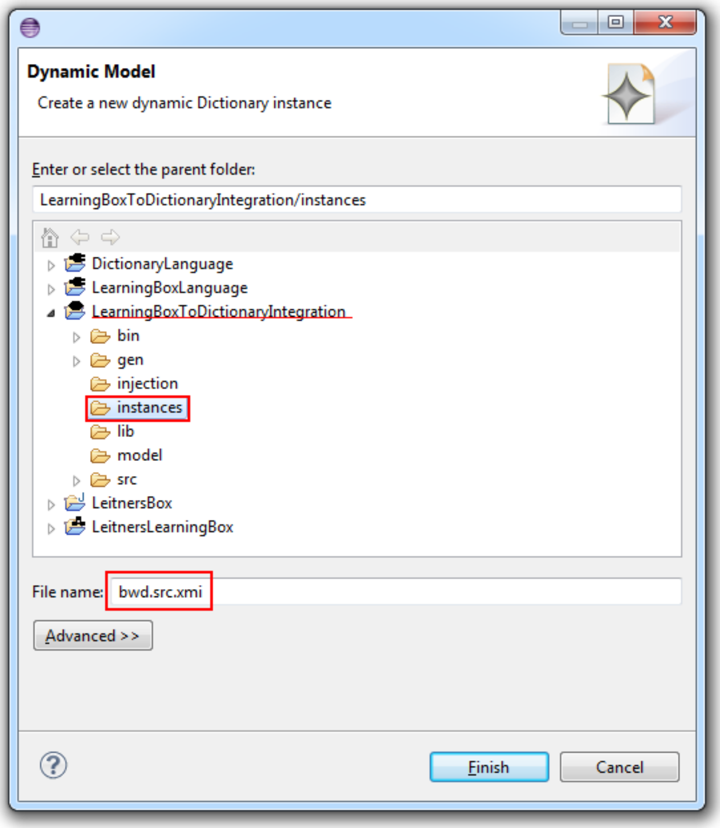

Figure 5.1: Create a dynamic instance of `Dictionary`

[12] 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 created for each `entry.content` in the `CardToEntryRule` when setting up the constraints! Be sure to set this property as `<word>:<meaning>`. Give each `entry` a diffirent difficulty `level`, e.g., `beginner` for `One:Eins`, `advanced` for `Two:Zwei`, and `master` for `Three:Drei`. Your instance should resemble Fig. 5.2.

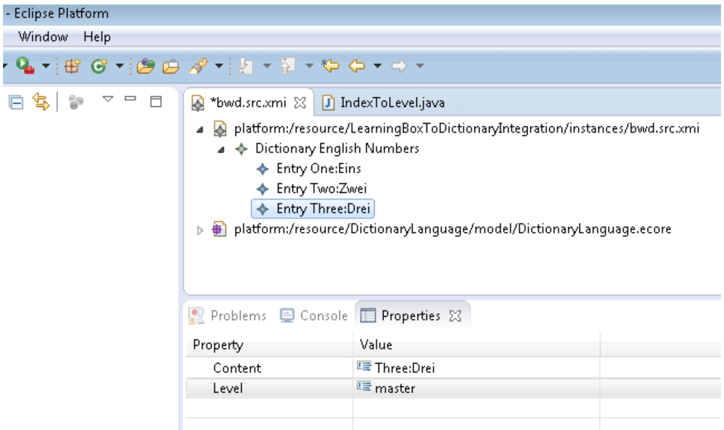

Figure 5.2: Fill a `Dictionary` for the transformation

- Let's check out the file that will actually execute our transformation. Navigate to "LearningBoxToDictionaryIntegration/src/org.moflon.tie" and click to open `LearningBoxToDictionaryIntegrationTrafo.java`.

- As you can see, this file is the driver which runs the complete transformation, first transforming forward from a source `box` to a target `dictionary`, then backward from `dictionary` to `box`. As this is plain Java, you can adjust everything freely as you wish.

- Right-click the file in the Package Explorer and got 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 (Fig. 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.

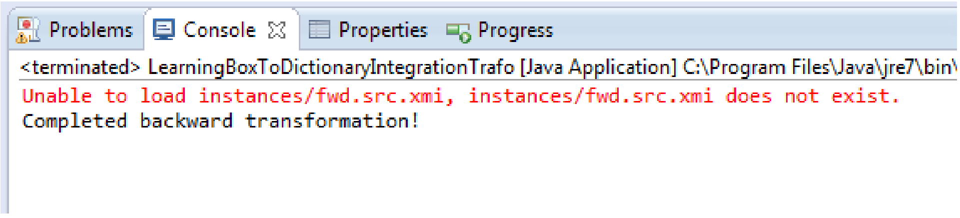

Figure 5.3: Running the backward transformation

- ▶ Refresh the integration'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 listing of the attempted steps taken (as well as their results), and `bwd.trg.xmi`, the output of the transformation. Open this last file in the editor.

- ▶ It's a `Box` of `English Numbers`! Expand the tree and you'll see our `Dictionary` in its equivalent `Box` format containing three `Partitions` (Fig. 5.4). Double click each `card` and observe how each `entry.content` was successfully split into two sides.

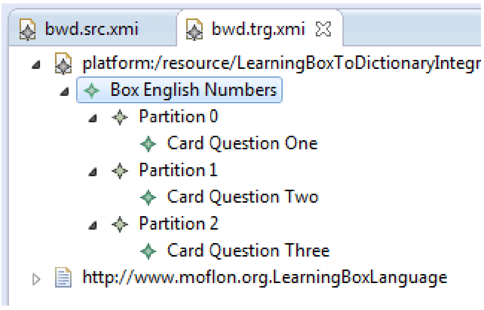

Figure 5.4: Result of the *backwards* transformation

- ▶ Congratulations! You have successfully performed your first *backward* transformation using TGGs!

► Don't forget about one of eMoflon's coolest model visualizing features – the graph viewer.[13] This is an especially useful tool for TGGs when you need to quickly confirm your transformation was successful. Drag-and-drop `Box English Numbers` into the graph view (Fig. 5.5). You should be able to see each `Card`'s container `partition` and the edges via which they'll move between partitions.

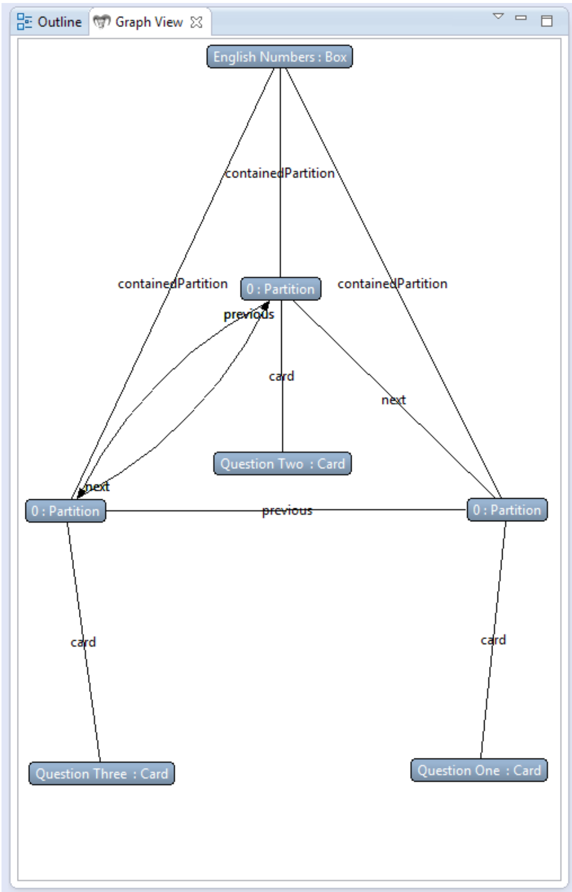

Figure 5.5: Confirm the transformation with the Graph Viewer

► To show that the transformation is actually bidirectional, let's create a source model (thus resolving the error), and run the TGG again to perform a *forwards* transformation of a `Box` into a `Dictionary`. Make a copy of `bwd.trg.xmi` and rename it to `fwd.src.xmi`.

[13]Refer to Part 2, Section 4 to review how to open and use this tool

- ► Run `LearningBoxToDictionaryIntegrationTrafo` again by pressing the green "Run As…" icon on the toolbar. You should now have two success messages in the console window! Finally, refresh the "instances" folder and compare the output `fwd.trg.xmi` against the original `bwd.src.xmi` Dictionary model. If everything executed properly, they should look exactly the same.

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.

Our goal was never to be able to put an `Entry` into partitions with indices greater than two,[14] but simply to be able to put any `card` into a `Dictionary`. This means that such additional partitions are irrelevant for the dictionary and should be ignored. In this particular case, you should specify an extra rule that clearly states how such partitions should be ignored, i.e., be translated without affecting the dictionary. In this spirit, let's add a new rule to handle additional partitions. We could keep things simple by extending 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.

ignore rule

While building this so-called *ignore rule*, keep in mind that the goal is to handle any additional elements and their connecting link variables in `Box`. This means we don't need to create any new elements in the `Dictionary`.

Before specifying the ignore rule, extend your model `fwd.src.xmi` by a new `partition3` (with `index` = 3) as depicted in Fig.6.1. Connect your `partition3` to `partition0` via a `previous` reference, and connect also `partition2` to `partition3` via a `next` reference. Create a new `card` in your new `partition3` as well. If you run your transformation again, you will just get some errors for the forward direction as our `fwd.src.xmi` with four `partitions` simply cannot be handled with our TGG.

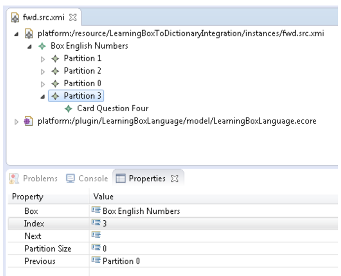

Figure 6.1: Extended `fwd.src.xmi`

[14] As resolved in the `IndexTolevel` implementation

▷ Next [visual]
▷ Next [textual]

6.1 AllOtherPartitionsRule

Remember that you can start the majority of new rules in two different ways! You can either return to the TGG's `Rules` diagram and use the toolbox there, or knowing that you need `box` and `partition0` for the context of the transformation, you can *derive* this rule from `BoxToDictionaryRule`.

- ▶ Once you've initialized the `AllOtherPartitionsRule` diagram, build the rule until it matches Fig. 6.2.

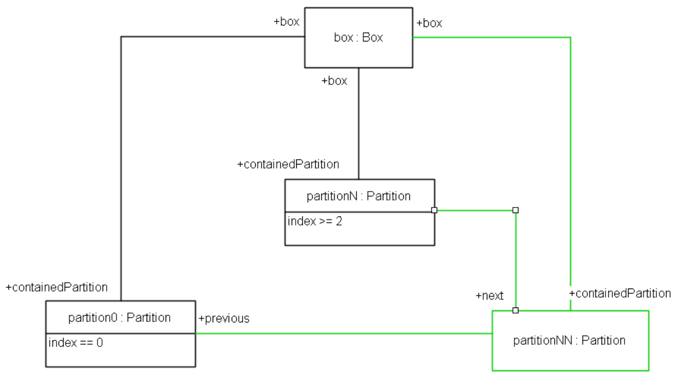

Figure 6.2: The completed `AllOtherPartitionsRule`

- ▶ As you can see, this rule doesn't assume to know the final `partition` in the transformation. It matches some `n`th partition with an `index` 2 or more, then connects a new `n+1`th partition to `n` and `partition0`.
- ▶ Save, validate, export, and refresh your Eclipse package explorer to generate code for this rule. Run the TGG again – it works! The transformation is now able to handle the troublesome `next` dangling edge from the third partition.
- ▶ Feel free to go ahead and add as many `partitions` and `cards` as you like to your model instance. Your TGG is now also able to handle a `box` with any number of `partitions` beautifully.
- ▶ To see how this rule is specified in the textual syntax, check out Fig. 6.3 in the next section.

6.2 AllOtherCardsRule

- ▶ Right click on the `Rules` folder again and create `AllOtherCardsRule`. Complete each scope until your file resembles Fig. ??.

AllOtherCardsRule.tgg

```
rule AllOtherCardsRule {
    source {
        box : Box {
            -containedPartition-> partition0
            -containedPartition-> partitionN
            ++ -containedPartition-> partitionNN
        }

        partition0 : Partition {
            partition0.index == 0
        }

        partitionN : Partition {
            partition0.index >= 2
            ++ -next-> partitionNN
        }

        ++ partitionNN : Partition {
            ++ -previous-> partition0
        }
    }

    correspondence {

    }

    target {


    }

    constraints {[
        add(partitionN.index, 1, partitionNN.index)
    ]}
}
```

Figure 6.3: A complete `AllOtherCardsRule`

- ▶ You'll notice that `box` and `partition0` have been established as 'black' objects. This is so the rule may only be evaluated when these objects are already translated, so we can use their values from the context of the transformation.

- ▶ A second partition, `partitionN`, has also been established as part of the context. It represents the `nth`, or last translated partition in a `box` (with an index of 2 or higher), whose `next` reference will also be translated in order to provide an access link to the new `partitionNN` element.

▶ Given that the syntax of `add(a,b,c)` is `a+b=c`, the sole constraint of this rule sets the `index` of the `n+1`th partition so that the `partitions` are still listed in order. Note that the correspondence and target scopes are empty, which is typical for such ignore rules.

▶ That's it! Save and build, then run the TGG again with the 'extra' `partition` to confirm it worked! If so, you are now free to add as many `partitions` and `cards` to `source.xmi` – the transformation is now able to elegantly ignore them all.

▶ Be sure to check out how this rule is implemented in eMoflon's visual syntax in Fig. 6.2 from the previous section.

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). Your source and target metamodels are complete, and given that you probably won't make any further changes to your rules, your correspondence metamodel is also complete.

Now suppose you wanted to make a minor change to one of your current instances, such as adding a single a 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 (Fig. 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`.[15]
- ▶ Run `LearningBoxToDictionaryIntegrationTrafo.java` and inspect the resulting `bwd.trg.xmi` (Fig. 7.1). Unforunately, the newest `partition3` is missing!

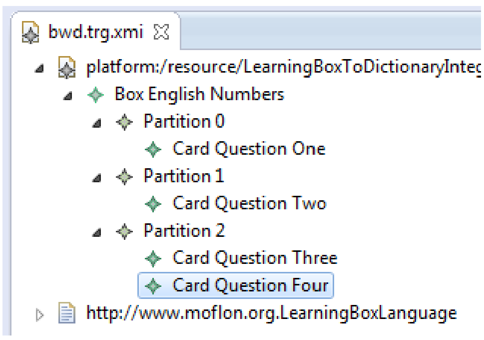

Figure 7.1: The transformation loses data for any index greater than 2

As expected, this extra partition was lost because our TGG rules are only able to create exactly three partitions, not additional ones with unique indexing. How can we prevent data loss when we need to update our models in the future? Luckily, eMoflon can take care of this for you as it provides

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

synchronization support to update your files incrementally. 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.

- ▶ Open `LearningBoxToDictionaryIntegrationSynch.java`, locate the empty `syncForward` method, and edit it as shown in Fig. 7.2.

```
public void syncForward(String corr) {
        setChangeSrc(root -> {
                Box box = (Box) root;
                Partition partition3 = box.getContainedPartition()
                        .stream().filter(p -> p.getIndex() > 2).findAny().get();
                Card newCard = LearningBoxLanguageFactory.eINSTANCE.createCard();
                newCard.setBack("Question Five");
                newCard.setFace("Answer Fuenf");
                partition3.getCard().add(newCard);
        });
        loadTriple(corr);
        loadSynchronizationProtocol("instances/fwd.protocol.xmi");
        integrateForward();
        saveResult("fwd");

        System.out.println("Completed forward synchronization");
}
```

Figure 7.2: Implementation of our custom `IndexToLevel` constraint

- ▶ Save and run the file. You'll notice that even though no changes were specified for the backward synchronization, both directions completed without error.

- ▶ View the results of your changes by first opening your learning box in `sync.fwd.src.xmi`. Expand `Partition 3` – is there now a fifth `Card` inside? As you can see, the synchronization saved your changes to this new file, rather than overwriting the original instance model.

- ▶ Open the synchronization's output file, `sync.bwd.src.xmi`. If it was successful there should be a fifth `Entry` in the Dictionary. Together with `sync.fwd.corr.xmi` and `sync.fwd.protocol.xmi`, this files form a new triple and will remain consistent with one another!

- ▶ What if we wanted to make a change in the other direction? Let's try deleting an entry while keeping all four partitions. Open the synchronization's Java file again and locate `syncBackward`. Replace the code as depicted in Fig. 7.3.

```java
public void syncBackward(String corr) {
        setChangeTrg(root -> {
                Dictionary dictionary = (Dictionary) root;
                Entry deleted = dictionary.getEntry().remove(0);
        });
        loadTriple(corr);
        loadSynchronizationProtocol("instances/fwd.protocol.xmi");
        integrateBackward();
        saveResult("bwd");

        System.out.println("Completed backward synchronization");
}
```

Figure 7.3: Implementation of our custom `IndexToLevel` constraint

- ▶ Run the synchronization a final time, and refresh the "instances" folder. Open and inspect both `sync.bwd.src.xmi` and `sync.bwd.-trg.xmi`. If everything has executed correctly, a `Card` should be missing in `Box`, and `partition3` still exists! Equivalently, there should only be four `Entry` elements in the output `Dictionary`.

- ▶ You may have noticed that there is no `Entry Five`, which we added to `partition3` in the forward synchronization. Recall that the process loads and makes changes to the *original* triple, not the most recent copies. The files are simple Java code, so you are invited to modify them as you wish for your own projects.

8 Model Generation

In addition to model transformation and model synchronization, TGG specifications can be used to generate models. Often there is a need for large and randomly generated models for testing purposes. As creating large and valid models manually is quite difficult, eMoflon offers a model generation framework to automate this generation process.

- ▶ Press the `new` button on the Eclipse toolbar and navigate to "Examples/eMoflon Handbook Examples/" (Fig. 8.1). Find and select `Final Solution : Visual` to copy the necessary projects into your workspace.

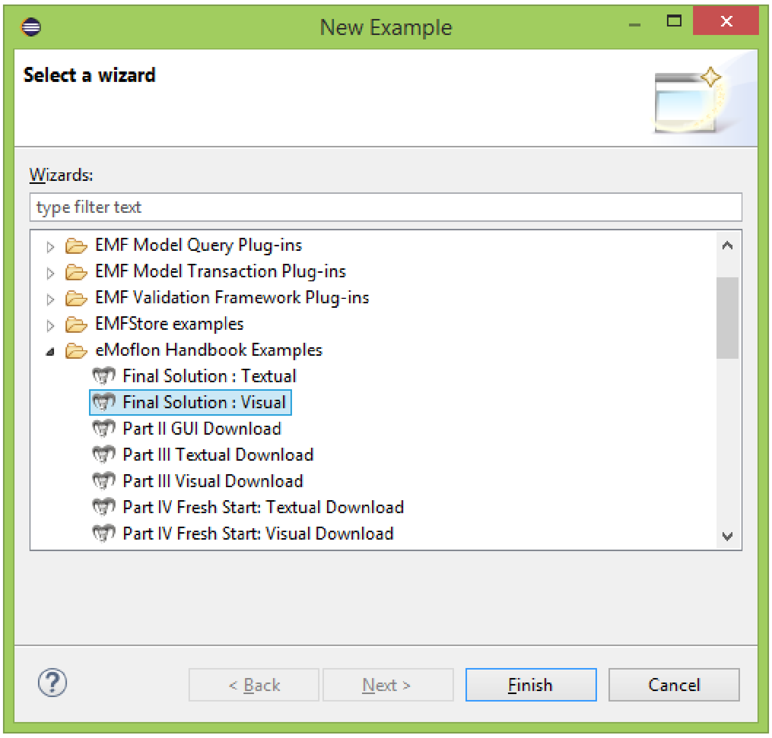

Figure 8.1: Get the final visual solution

- ▶ Now delete the projects `Dictionary` and `DictionaryCodeAdapter` as they are not needed for the model generator. If successful, your workspace should resemble Fig. 8.2.
- ▶ In order to generate code for the model generator you should adjust your **moflon.properties.xmi** file in the **LearningBoxToDic-**

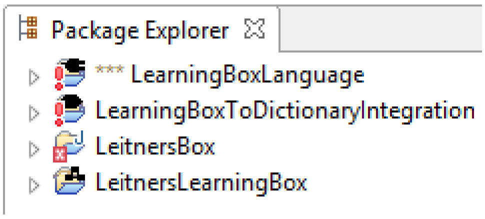

Figure 8.2: Example projects from the wizard

tionaryIntegration project (Fig. 8.3). Make sure that the value of the **TGG Build mode** property is set to **SIMULTANEOUS** or **ALL** and save the file.

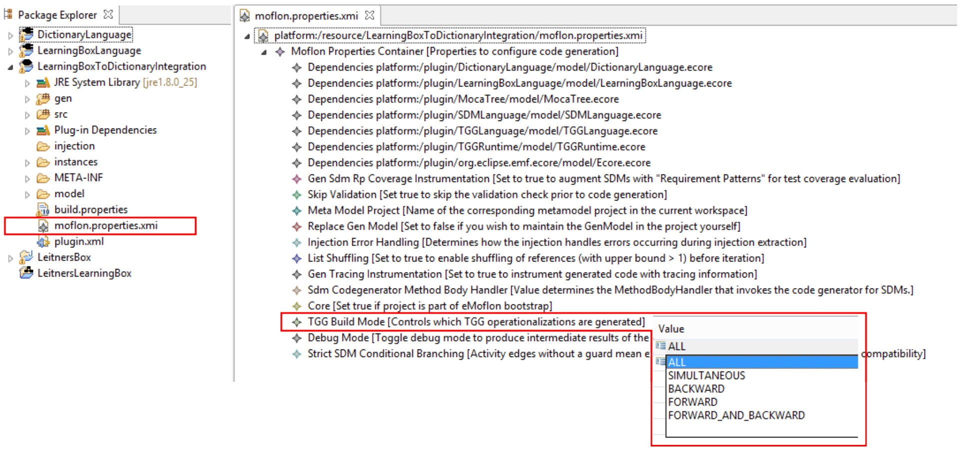

Figure 8.3: Screenshot of the **moflon.properties.xmi**

- ▶ Run a new `Export and build` on the `LeitnersLearningBox.eap` and build your projects in Eclipse in order to generate code.

Now your generated code contains the necessary methods for model generation. In your project **src** folder you can see the file **LearningBoxToDictionaryIntegrationModelGen.java** (Fig. 8.4). This class contains a stub and can be used to execute the model generation process.

The **ModelGenerator** class uses an **AbstractModelGenerationController** to control the generation process (line 35). In this template the generation process will be terminated after 20 rules has been performed (**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

```java
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 8.4: Stub for the model generator

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). For instance, if you want axiom rules to be performed exactly once to only generate models with a single root. You can create your own controller classes if the delivered ones are not sufficient. To create your first models do the following:

- ▶ Open your **LearningBoxToDictionaryIntegrationModelGen.java** file.
- ▶ Change <**enter rule name**> to **BoxToDictionaryRule** to create single root models. This rule will only be performed once.
- ▶ Save the file and run it.

You will get some logging information in the console (Fig. 8.5) which contains details 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.

In your **instances** folder should now be a new folder named **generatedModels** with a timestamp suffix. It contains your newly generated source and target models.

To support model generation for custom attribute constraints you may have to specify additional adornments. Adjusting the adornments is needed if the existing adornments does not cover the cases which arise using the TGG for model generation (e.g. the attribute constraints refers to only green object variables where all variables are free). Note that no additional adornments are required for our example.

- ▶ Open up the dialog for your custom constraint (figure 8.6).

```
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 8.5: Logging output after model generation

- ► Enter the required adornments for the model generator. Implement the new case in your constraint java file. On figure 8.7 you can see the built-in implementation for the **FF**-case of the **Eq**-constraint. The Generator class is able to return random string values with respect to the type parameter. For a number type the string will only contain numbers.

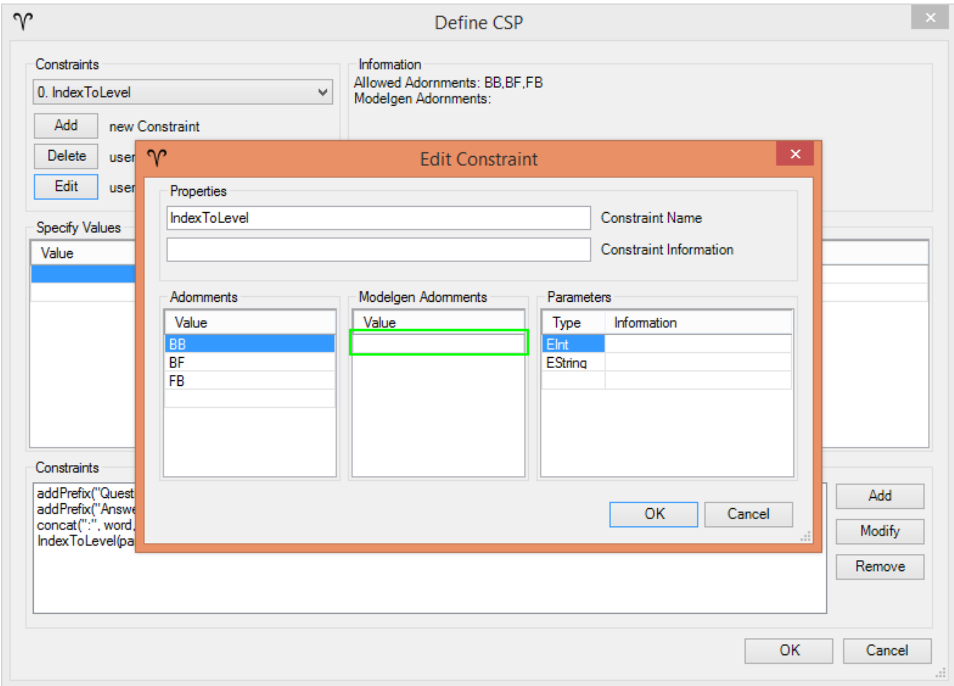

Figure 8.6: Custom constraint dialog

```
// modelgen implementation
else if (bindingStates.equals("FF"))
{
  String value = Generator.getNewRandomString(a.getType());
  a.bindToValue(value);
  b.bindToValue(value);
  setSatisfied(true);
}
```

Figure 8.7: Eq-constraint implementation for the FF case

9 Conclusion and next steps

Fantastic work – you've mastered Part IV of the eMoflon handbook! You've learnt the key points of Triple Graph Grammars and *bidirectional* transformations, how to set up a TGG via a schema and a set of rules. The transformation was visually inspected using eMoflon's integrator. With these basic skills, you should be able to tackle most bidirectional transformations using TGGs.

If you enjoyed working through this part, try completing the next part of this handbook, Part V: Model-to-Text Transformations. There, we shall implement a larger transformation as a case study using TGGs. Alternatively, if you don't have much time left, skip ahead to Part VI: Miscellaneous for information about some additional eMoflon features, some tips and tricks on using eMoflon efficiently, as well as an expanded glossary and list of all eMoflon hotkeys.

For detailed descriptions on the upcoming and previous parts of this handbook, please refer to Part 0, which can be found at `http://tiny.cc/emoflon-rel-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.