

An Introduction to Metamodelling and Graph Transformations

with eMoflon



Part 0: Introduction

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

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

The eMoflon team
Darmstadt, Germany (August 2015)

Part 0:

Introduction

This handbook has been engineered to be *fun*.

If you work through it and, for some reason, do *not* have a resounding “I-Rule” feeling afterwards, please send us an email and tell us how to improve it at contact@emoflon.org.

URL of this document: <http://tiny.cc/emoflon-rel-handbook/part0.pdf>



Figure 0.1: How you should feel when you’re done

To enjoy the experience, you should be fairly comfortable with Java or a comparable object-oriented language, and know how to perform basic tasks in Eclipse. Although we assume this, we give references to help bring you up to speed as necessary. Last but not least, basic knowledge of common UML notation would be helpful.

Our goal is to give a *hands-on* introduction to metamodeling and graph transformations using our tool *eMoflon*. The idea is to *learn by doing* and all concepts are introduced while working on a concrete example. The language and style used throughout is intentionally relaxed and non-academic.

So, what is eMoflon?

eMoflon is a tool for building tools. If you wish, a “meta” tool. This means that if you’re interested in building *domain-specific* tools for end users, then eMoflon could be pretty useful for you.

Why should I be interested?

To build a tool, you typically need a way for users to communicate with it, i.e., you must establish a suitable *language* for specifying input and output to and from the tool. You also need a central data structure to represent the “state” of the tool. This data structure or *model* is usually manipulated and appropriately *transformed* in some useful manner by the tool. Many tools also *generate* something useful from their internal models and keep them synchronized with other models in different tools. To achieve these goals you can use *metamodeling* to define your language (your *metamodel*), *graph transformations* to transform your models, and parsing/code generation techniques to produce something useful from your models. All this and much more is supported by eMoflon. Take a look at Fig. 0.2 to see how all these tasks fit together.

What does this handbook cover?

On the last page, we’ve described each of the 6 parts that make up this handbook. You can work through them sequentially and become an *official*¹ eMoflon master or, depending on your interests, decide what you’d like to read and what to skip. We provide all the necessary materials (i.e., a cheat package) so you can jump right in without having to complete the previous parts. For those of you interested in further details and the mature formalism of graph transformations, we give relevant references throughout the handbook.

¹Certificate not guaranteed

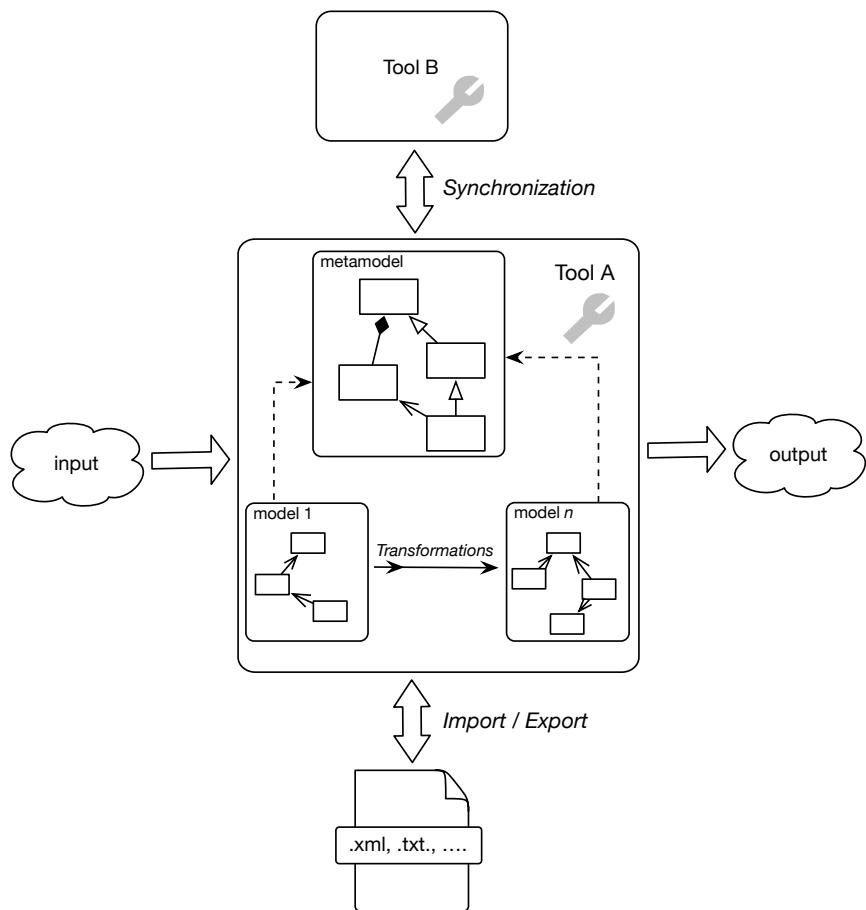


Figure 0.2: Building a tool requires language specification (metamodelling), transformations, synchronization, parsing, and code generation

Part I: Installation and set up provides a very simple example and a few JUnit tests to test the installation and configuration of eMoflon. We also explain the general workflow and the different workspaces involved.

This part can be considered *mandatory* only if you are new to eMoflon, but we recommend working through it anyway.

Approximate time to complete: 1 h

File download: <http://tiny.cc/emoflon-rel-handbook/part1.pdf>

Part II: Ecore takes you step-by-step through a more realistic example that showcases many of the features we currently support. Working through this part should serve as a basic introduction to model-driven engineering, and is especially recommended if you're new to metamodeling (using Ecore and the Eclipse Modeling Framework (EMF)).

Approximate time to complete: 2 h 30 min

File download: <http://tiny.cc/emoflon-rel-handbook/part2.pdf>

Part III: Story Driven Modelling (SDM) introduces *unidirectional* model transformation via programmed graph transformation using story diagrams.

Approximate time to complete: 4 h 45 min

File download: <http://tiny.cc/emoflon-rel-handbook/part3.pdf>

Part IV: TGGs introduces *bidirectional* model transformation with Triple Graph Grammars (TGGs).

Approximate time to complete: 3 h

File download: <http://tiny.cc/emoflon-rel-handbook/part4.pdf>

Part V: Model-to-Text Transformations shows how standard parsing and code generation technology can be combined with story diagrams and TGGs.

Approximate time to complete 3 h 30 min

File download: <http://tiny.cc/emoflon-rel-handbook/part5.pdf>

Part VI: Miscellaneous contains a collection of tips and tricks to keep on hand while using eMoflon. This can be used as a reference to help avoid common mistakes and increase productivity. If you're in a hurry, this part can be skipped and consulted only on demand.

Approximate time to complete: 1 h 20 min

File download: <http://tiny.cc/emoflon-rel-handbook/part6.pdf>

Well, that's it! Download Part I, grab a coffee, and enjoy the ride!

An Introduction to Metamodelling and Graph Transformations

with eMoflon



Part I: Installation and Setup

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

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

The eMoflon team
Darmstadt, Germany (August 2015)

Contents

1	Getting started	1
2	Get a simple demo running	5
3	Validate your installation with JUnit	11
4	Project setup	13
5	Generated code vs. hand-written code	24
6	Conclusion and next steps	26

Part I:

Installation and Setup

This part provides a very simple example and a JUnit test to check the installation and configuration of eMoflon. It can be considered *mandatory* if you are new to eMoflon, but we recommend working through it anyway.

After working through this part, you should have an installed and tested eMoflon working for a trivial example. We also explain the general workflow, the different workspaces involved, and general usage of both our visual and textual syntax.

Approximate time to complete: Just a few minutes...

URL of this document: <http://tiny.cc/emoflon-rel-handbook/part1.pdf>

1 Getting started

Here's how we've organized our handbooks; Black, red, and blue headers are used to separate common, visual, and textual syntax instructions (Fig 1.1).



Figure 1.1: Page headers and links

You'll find a `▷ link` at the bottom of some pages. These will take you to the next appropriate place for your syntax. You are still welcome to go through

the entire handbook page by page. In fact, we encourage it and hope you'll compare the differences and similarities between the two specifications. But be warned! If what you're doing isn't matching what you see, you may be reading the wrong instructions.

If, however, you're finding that the screenshots we've taken aren't matching your screen and you *ARE* in the right place, please send us an email at contact@emoflon.org and let us know. They get outdated so fast! They just grow up, move on, start doing their own thing and ... uh, wait a second. We're talking about pictures here.

Here are some guidelines to help you decide which syntax to use:

- ▶ If you have used a UML tool before and feel comfortable with the standard UML diagrams, then you might prefer our visual syntax.
- ▶ If you do not like switching tools, and want everything integrated completely in Eclipse with zero installation, then you should stick to our textual syntax.
- ▶ If you use a Mac (or some other *Nix system) then you will need virtualisation software to run Windows and install the required UML tool for our visual syntax. Most maconians find this insulting and of course choose our textual syntax.
- ▶ As a final remark, consider that graph transformations obviously have something to do with graphs, which are inherently two dimensional structures. A visual syntax thus has some obvious advantages and is what we (currently) prefer and use internally (eMoflon is built with eMoflon).

1.1 Install our plugin for Eclipse

- ▶ Make sure that you have **Java 1.8** installed.
- ▶ Download and install Eclipse Luna for modelling, which is called “**Eclipse Modeling Tools**” from <http://www.eclipse.org/downloads/index.php>.¹ (Fig. 1.2).

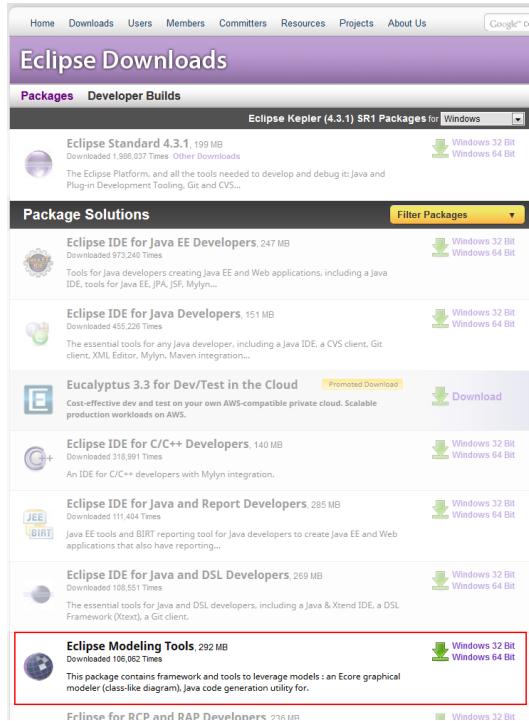


Figure 1.2: Download **Eclipse Modeling Tools**

- ▶ Install our Eclipse Plugin from the following update site²: <http://tiny.cc/emoflon-rel-update-site>

Please note: Calculating requirements and dependencies when installing the plugin might take quite a while depending on your internet connection.

¹Please note that you *have to* install Eclipse Modelling Tools, or else some features won't work! Although different versions may support eMoflon, our tool is currently tested only for Luna and Java 1.8.

²For a detailed tutorial on how to install Eclipse and Eclipse Plugins please refer to <http://www.vogella.de/articles/Eclipse/article.html>

1.2 Install our extension for Enterprise Architect

Enterprise Architect (EA) is a visual modelling tool that supports UML³ and a host of other modelling languages. EA is not only affordable but also quite flexible, and can be extended via *extensions* to support new modelling tools – such as eMoflon!

- ▶ Download EA for Windows from <http://www.sparxsystems.com/> to get a free 30 day trial and follow installation instructions (Fig. 1.3).



Figure 1.3: Download Enterprise Architect

- ▶ Install our EA extension (Fig. 1.4) to add support for our modelling languages. Download <http://tiny.cc/emoflon-rel-eaaddin>, unpack, and run `eMoflonAddinInstaller.msi`.

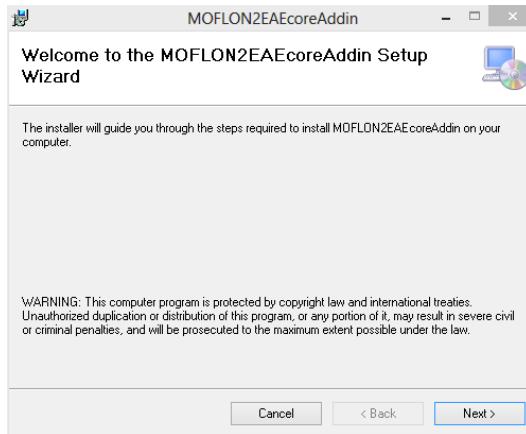


Figure 1.4: Install our extension for EA

³Unified Modelling Language

2 Get a simple demo running

- Open Eclipse to a clean, fresh workspace. Go to “Window/Open Perspective/Other...”⁴ and choose **eMoflon** (Fig. 2.1).

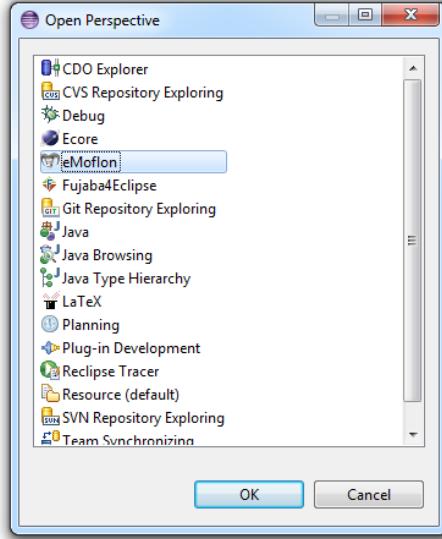


Figure 2.1: Choose the eMoflon perspective

- At either the far right or centre of the toolbar, a new action set should have appeared. Navigate to “New Metamodel” (Fig. 2.2).



Figure 2.2: Invoke the “New Metamodel” wizard

⁴A path given as “foo/bar” indicates how to navigate in a series of menus and toolbars. New definitions or concepts will be *italicized*, and any data you’re required to enter, open, or select will be given as **command**.

- A new dialogue should appear. This is the place where you officially decide which eMoflon syntax you'd like to use (Fig. 2.3).

With the visual syntax, you'll be using Eclipse along with a UML tool, Enterprise Architect (EA). You'll use EA to specify several types of diagrams, then export and refresh your workspace in Eclipse to generate the corresponding Java code.

With the textual syntax (MOSL),⁵ you'll be working entirely within the Eclipse IDE.

No matter which syntax you choose however, give your project a name and make sure the **Add Demo Specification** button is selected. This will create all the files and tests required to ensure you've set up eMoflon correctly.

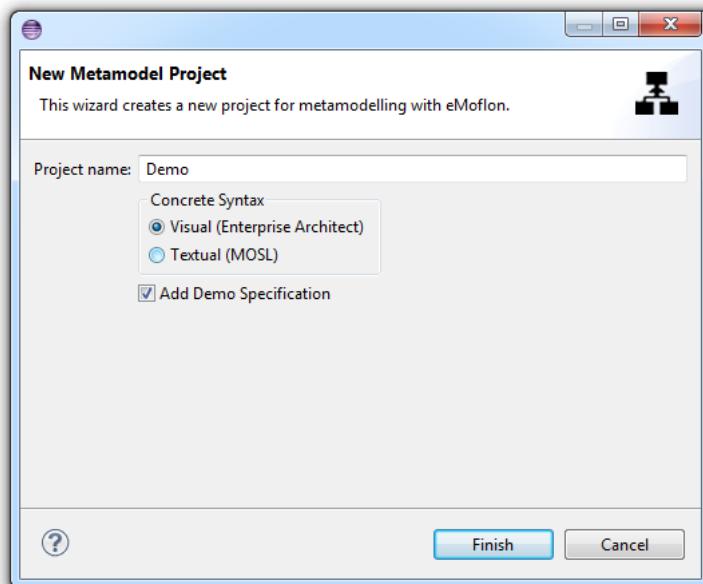


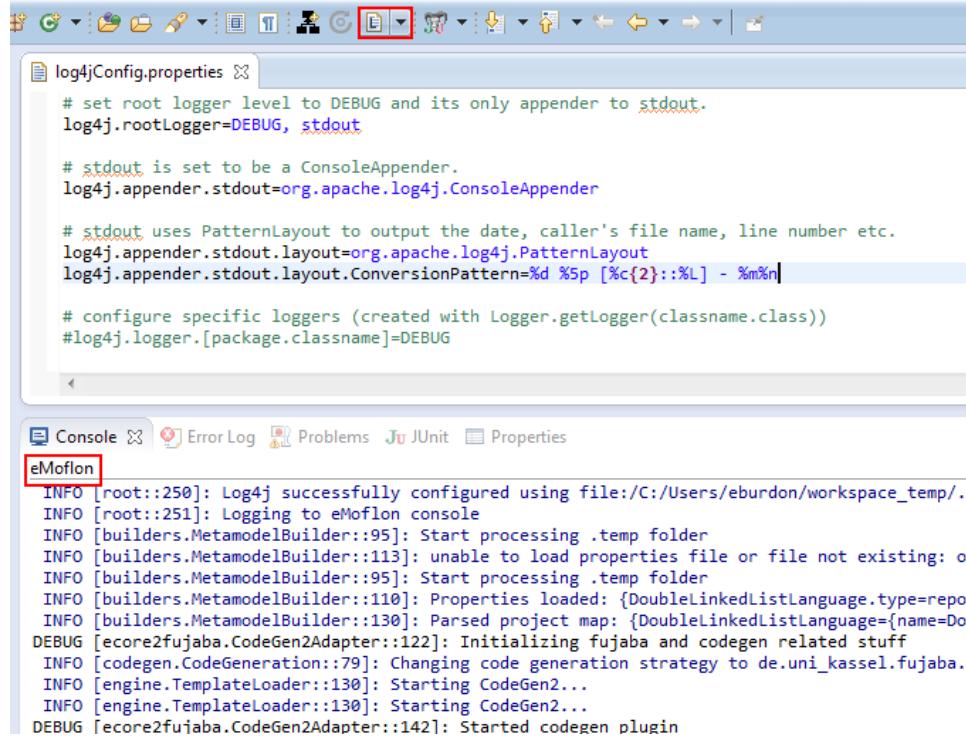
Figure 2.3: Choose your syntax

- Another button in the new action set is “View and configure logging” represented by an L (Fig. 2.4). Clicking this icon will open a `log4jConfig.properties` file where you can silence certain loggers, set the level of loggers, or configure other settings.⁶ All of eMoflon's messages appear in our console window, just below your main editor. This is automatically opened when you selected the `eMoflon`

⁵Moflon Specification Language

⁶If you're not sure how to do this, check out a short Log4j tutorial at <http://logging.apache.org/log4j/1.2/manual.html>

perspective and contains important information for us if something goes wrong!



The screenshot shows the eMoflon IDE interface. At the top, there is a toolbar with various icons. Below the toolbar, a file named "log4jConfig.properties" is open in a code editor window. The code in the file is as follows:

```

# set root logger level to DEBUG and its only appender to stdout.
log4j.rootLogger=DEBUG, stdout

# stdout is set to be a ConsoleAppender.
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

# stdout uses PatternLayout to output the date, caller's file name, line number etc.
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %5p [%c{2}::%L] - %m%n

# configure specific loggers (created with Logger.getLogger(classname.class))
#log4j.logger.[package.classname]=DEBUG

```

Below the code editor, there is a tab bar with several tabs: "Console", "Error Log", "Problems", "JUnit", and "Properties". The "Console" tab is selected and highlighted with a red box. In the console area, there is a list of log messages starting with:

```

INFO [root::250]: Log4j successfully configured using file:/C:/Users/eburdon/workspace_temp/..
INFO [root::251]: Logging to eMoflon console
INFO [builders.MetamodelBuilder::95]: Start processing .temp folder
INFO [builders.MetamodelBuilder::113]: unable to load properties file or file not existing: o
INFO [builders.MetamodelBuilder::95]: Start processing .temp folder
INFO [builders.MetamodelBuilder::110]: Properties loaded: {DoubleLinkedListLanguage.type=repo
INFO [builders.MetamodelBuilder::130]: Parsed project map: {DoubleLinkedListLanguage={name=Do
DEBUG [ecore2fujaba.CodeGen2Adapter::122]: Initializing fujaba and codegen related stuff
INFO [codegen.CodeGeneration::79]: Changing code generation strategy to de.uni_kassel.fujaba.
INFO [engine.TemplateLoader::130]: Starting CodeGen...
INFO [engine.TemplateLoader::130]: Starting CodeGen...
DEBUG [ecore2fujaba.CodeGen2Adapter::142]: Started codegen plugin

```

Figure 2.4: The eMoflon console with log messages

2.1 A first look at EA

- ▶ Can you locate the new `Demo.eap` file in your package explorer? This is the EA project file you'll be modelling in. Don't worry about any other folders at the moment - all problems will be resolved by the end of this section.
- In the meantime, do not rename, move, or delete anything.
- ▶ Double-click `Demo.eap` to start EA, and choose **Ultimate** when starting EA for the first time.
- ▶ In EA, navigate to and select “Extensions/Add-In Windows” (Fig. 2.5). This will activate our tool’s full control panel.

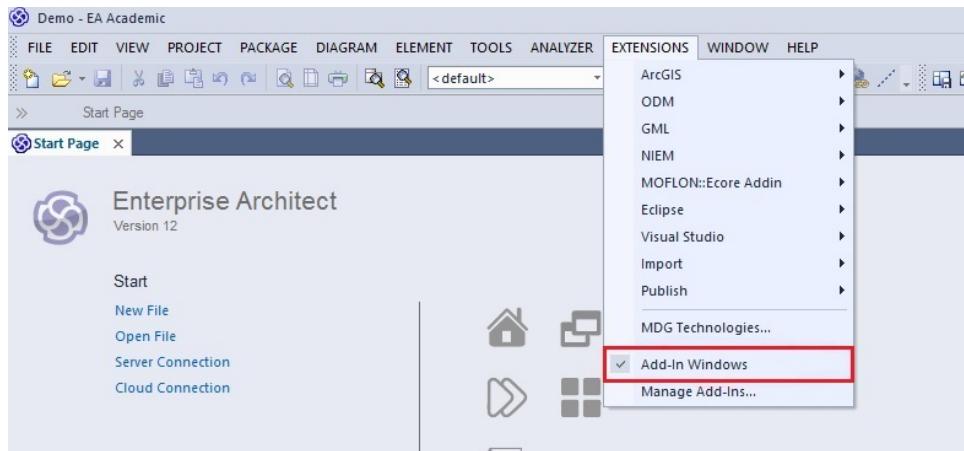


Figure 2.5: Export from EA

- ▶ This tabbed control panel provides access to all of eMoflon’s functionality. This is where you can validate and export your complete project to Eclipse by pressing **All** (Fig. 2.6).

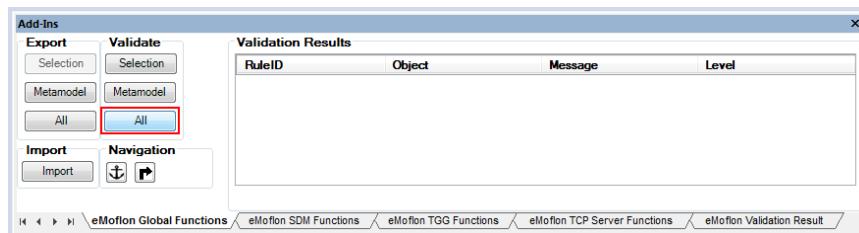


Figure 2.6: eMoflon’s control panel in EA

- ▶ Now try exploring the EA project browser! Try to navigate to the packages, classes, and diagrams. Don't worry if you don't understand that much - we'll get to explaining everything in a moment. Just make sure not to change anything!
- ▶ Switch back to Eclipse, choose your metamodel project, and press F5 to refresh. The export from EA places all required files in a hidden folder (.temp) in the project. A new, third project named *org.moflon.demo.doublelinkedlist* is now being created. Do not worry about the problem markers.
- ▶ The three asterisks (Fig. 2.7) signal that the project still needs to be built.

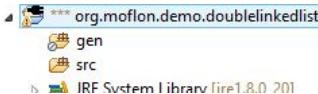


Figure 2.7: Dirty projects are marked with ***

- ▶ Now, right-click *org.moflon.demo.doublelinkedlist* and choose “eMoflon/Build” (or use the shortcut Alt+Shift+E,B⁷).
- eMoflon now generates the Java code in your repository project. You should be able to monitor the progress with the green bar in the lower right corner (Fig. 2.8). Pressing the symbol opens a monitor view that gives more details of the build process. You don't need to worry about any of these details, just remember to (i.) refresh your Eclipse workspace after an export, and (ii.) rebuild projects that bear a “dirty” marker (***)�.
- ▶ If you're ever worried about forgetting to refresh your workspace, or if you just don't want to bother with having to do this, Eclipse does offer an option to do it for you automatically. To activate this, go to “Window/Preferences/General/Workspace” and select **Refresh on access**.

⁷First press Alt+Shift+E, release, and press B. By default, most shortcuts eMoflon start with Alt+Shift+E.

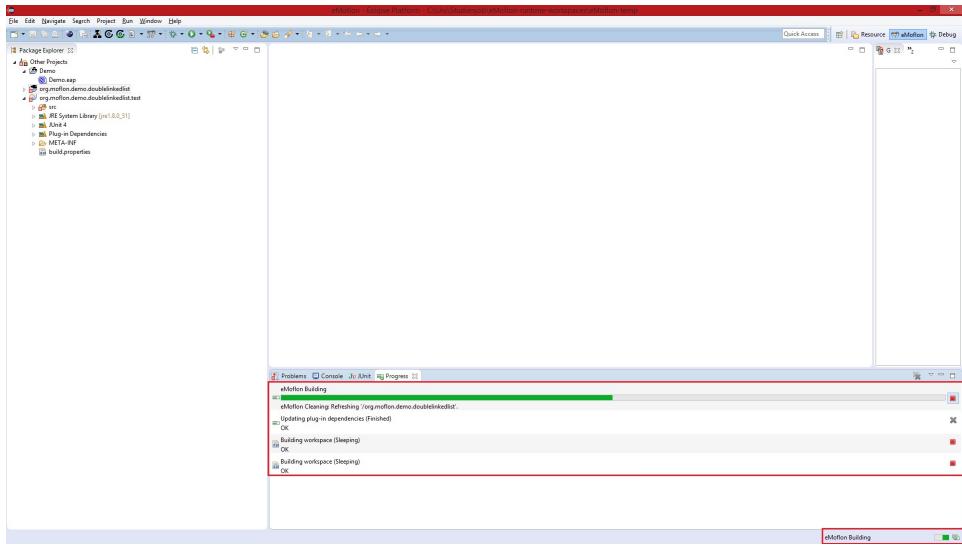


Figure 2.8: Eclipse workspace when using visual syntax

2.2 A first look at MOSL

Please note that the textual syntax is not as thoroughly tested as the visual syntax because most of our projects are built with the visual syntax. This means: Whenever something goes wrong even though you are sure to have followed the instructions precisely, do not hesitate to contact us via contact@emoflon.org.

- ▶ You should immediately have 3 folders available in your project explorer – Your metamodel project, `org.moflon.demo.doublelinkedlist`, and `orgmoflon.demo.doublelinkedlist.test` (Fig. 2.9). Initially, you'll see a red exclamation mark indicating there are errors, but once you give Eclipse a few seconds to refresh, all problems should be solved.
 - ▶ That's it! You're all set up! But, while you're here, feel free to explore the “Demo” folder. It has the basic code that implements this demo, and we recommend you take a brief look to get a feel for the the general syntax.
- In the meantime, please do not rename, move, or delete anything.

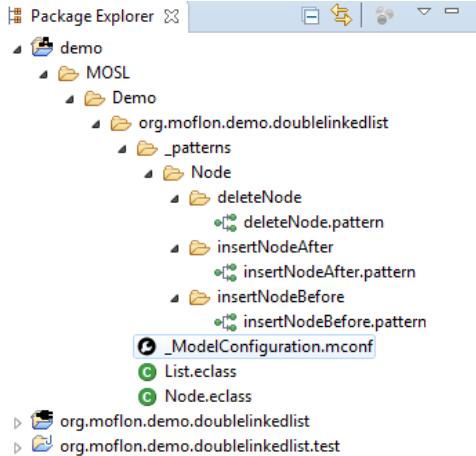


Figure 2.9: Metamodel file structure

3 Validate your installation with JUnit

- In Eclipse, choose **Working Sets** as your top level element in the package explorer (Fig. 3.1), as we use them to structure the workspace.

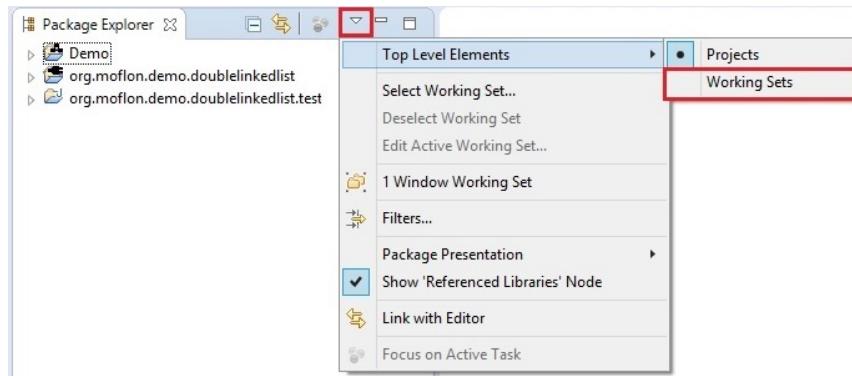


Figure 3.1: Top level elements in Eclipse

- Locate “Other Projects/orgmoflon.demo.doublelinkedlist.test.” This is the testsuite imported with the demo files to make sure everything has been installed and set up correctly. Right click on the project to bring up the context menu and go to “Run As/JUnit Test.” If anything goes wrong, try refreshing by choosing your metamodel project and pressing F5, or right-clicking and selecting Refresh.

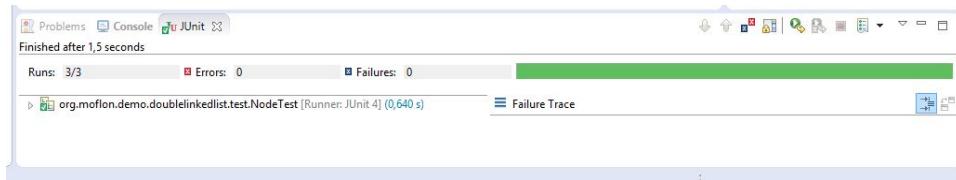


Figure 3.2: All's well that ends well...

Congratulations! If you see a green bar (Fig. 3.2), then everything has been set up correctly and you are now ready to start metamodelling!

- ▷ [Next \[visual\]](#)
- ▷ [Next \[textual\]](#)

4 Project setup

4.1 Your Enterprise Architect Workspace

Now that everything is installed and setup properly, let's take a closer look at the different workspaces and our workflow. Before we continue, please make a few slight adjustments to Enterprise Architect (EA) so you can easily compare your current workspace to our screenshots. These settings are advisable but you are, of course, free to choose your own colour schema.

- ▶ Select “Tools/Options/Themes” in EA, and set Diagram Theme to **Enterprise Architect 10**.
- ▶ Next, proceed to “Gradients and Background” and set “Gradient” and “Fill” to White (Fig. 4.1).

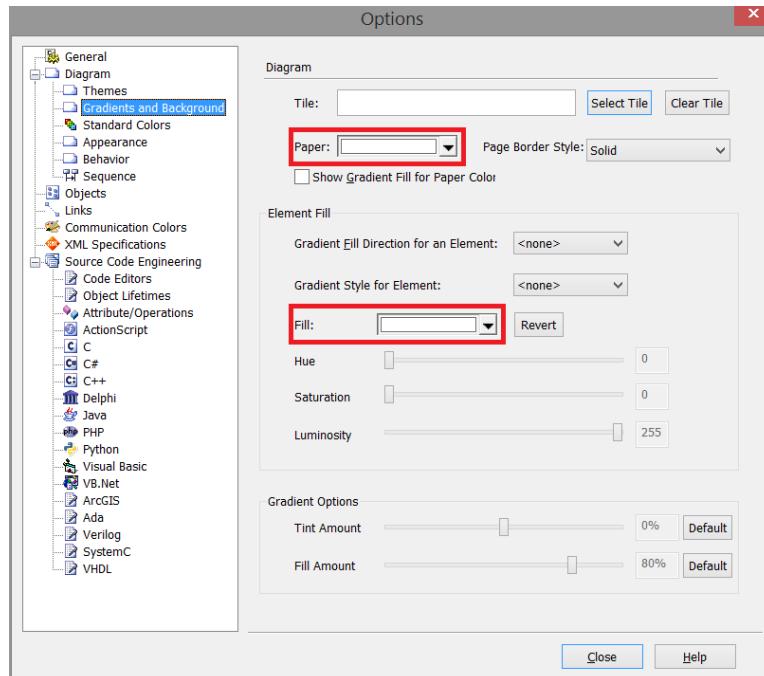


Figure 4.1: Suggested paper background and element fill

- ▶ In the “Standard Colors” tab, and set your colours to reflect Fig. 4.2.

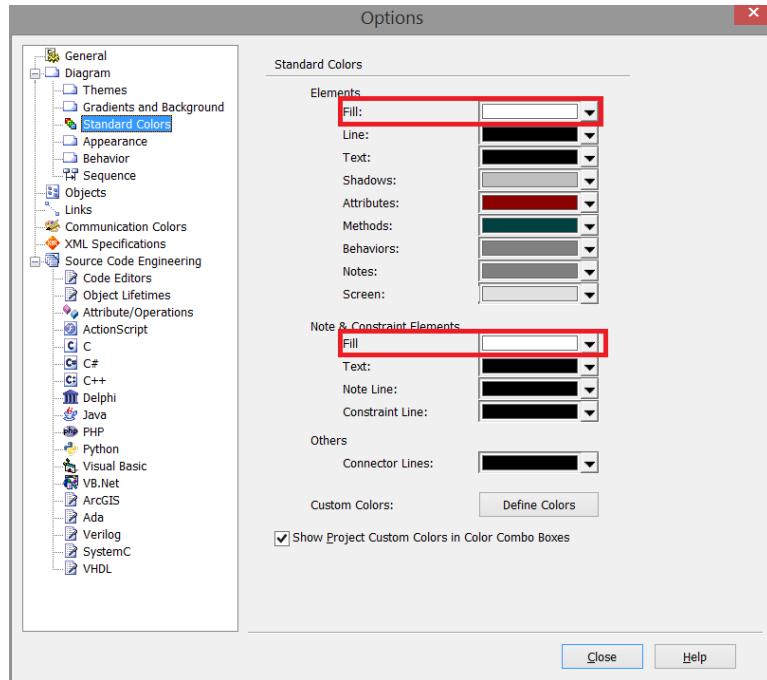


Figure 4.2: Our choice of standard colours for diagrams in EA

- In the same dialogue, go to “Diagram/Appearance” and reflect the settings in Fig. 4.3. Again, this is just a suggestion and not mandatory.
- Last but not least open the ”Code Engineering” toolbar (Fig. 4.4) and choose **Ecore** as the default language (Fig. 4.5). This setting *is* mandatory, and very important.

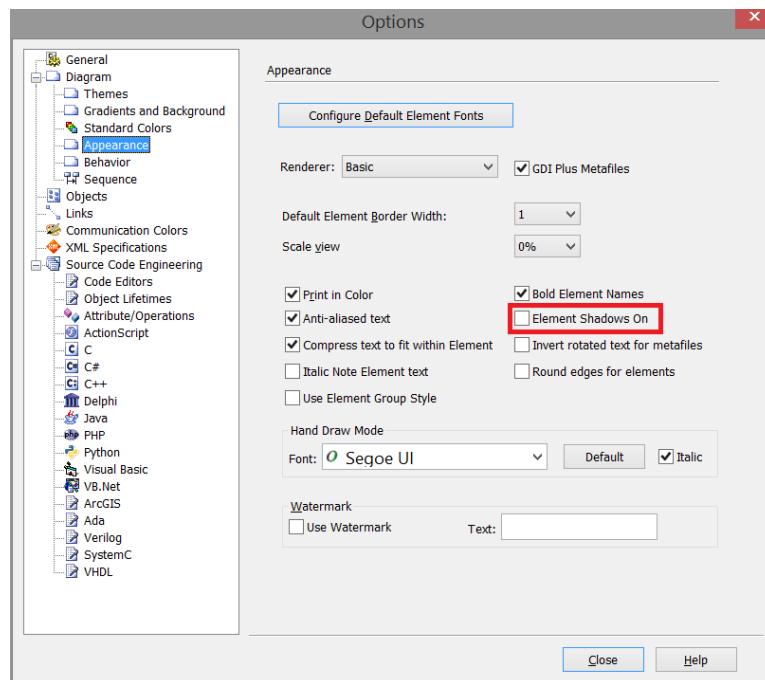


Figure 4.3: Our choice of the standard appearance for model elements

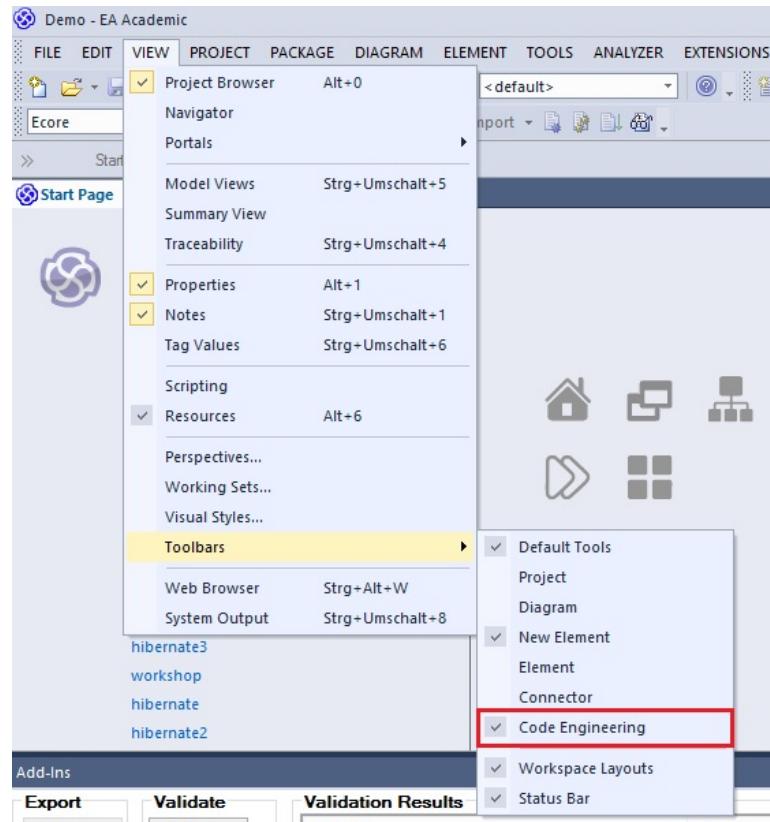


Figure 4.4: Open the "Code Engineering" toolbar

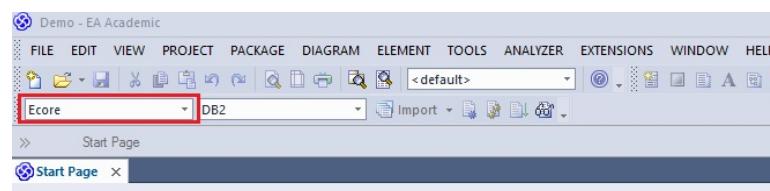


Figure 4.5: Make sure you set the standard language to Ecore

In your EA ‘workspace’ (actually referred to as an *EA project*), take a careful look at the project browser: The root node **Demo** is called a *model* in EA lingo, and is used as a container to group a set of related *packages*. In our case, **Demo** consists of a single package **DoubleLinkedListLanguage**. An EA project however, can consist of numerous models that in turn, group numerous packages.

Now switch back to your Eclipse workspace and note the two nodes named **Specifications** and **org.moflon.demo** (Fig. 4.6).

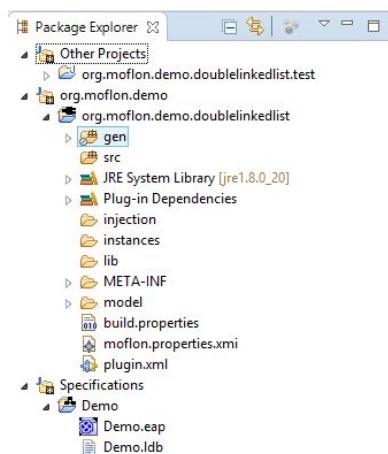


Figure 4.6: Project structure

These nodes, used to group related *Eclipse projects* in an Eclipse workspace, are called *working sets*. The working set **Specifications** contains all *metamodel projects* in a workspace. Your metamodel project contains a single EAP (EA project) file and is used to communicate with EA and initiate code generation by simply pressing F5 or choosing Refresh from the context menu. In our case, **Specifications** should contain a single metamodel project **Demo** containing our EA project file **Demo.eap**.

Figure 4.7 depicts how the Eclipse working set **Demo** and its contents were generated from the EA model **Demo**. Every model in EA is mapped to a working set in Eclipse with the same name. From every package in the EA model, an Eclipse project is generated, also with the same name.

These projects, however, are of a different *nature* than, for example, metamodel projects or normal Java projects. These are called *repository projects*. A nature is Eclipse lingo for “project type” and is visually indicated by a corresponding nature icon on the project folder. Our metamodel projects sport a neat little class diagram symbol. Repository projects are generated automatically with a certain project structure according to our conventions.

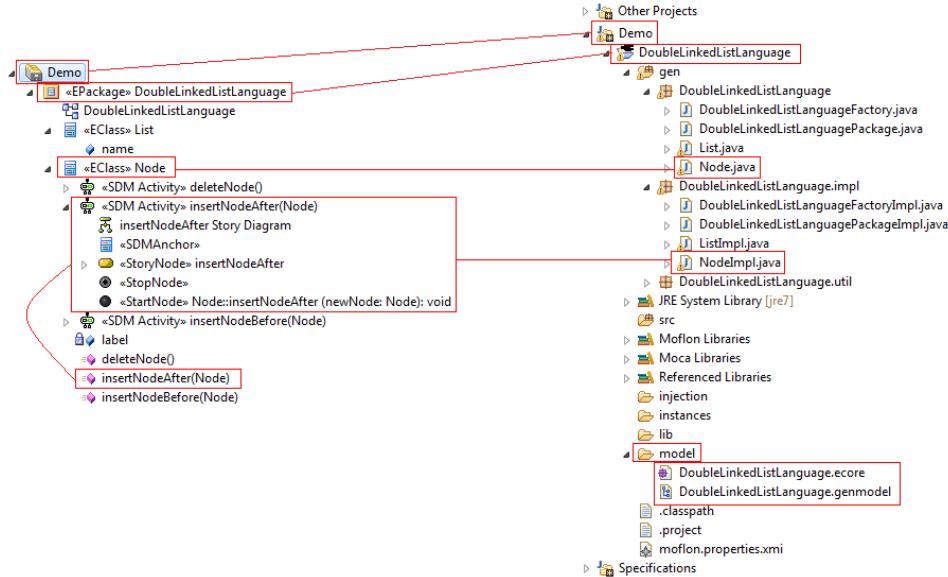


Figure 4.7: From EA to Eclipse

The `model` subfolder in the Eclipse package explorer is probably the most important as it contains the *Ecore model* for the project. Ecore is a metamodeling language that provides building blocks such as *classes* and *references* for defining the static structure (concepts and relations between concepts) of a system. This folder also contains a *Genmodel*, the second model required by the Eclipse Modeling Framework (EMF) to generate Java code.

Looking back to Fig. 4.7, realize that it also depicts how the class `Node` in the EA model is mapped to the Java interface `Node`. Double-click `Node.java` and take a look at the methods declared in the interface. These correspond directly to the methods declared in the modelled `Node` class.

As indicated by the source folders `src`, `injection`, and `gen`, we advocate a clean separation of hand-written (should be placed in `src` and `injection`) and generated code (automatically in `gen`). As we shall see later in the handbook, hand-written code can be integrated in generated classes via *injections*. This is sometimes necessary for small helper functions.

Have you noticed the methods of the `Node` class in our EA model? Now hold on tight – each method can be *modelled* completely in EA and the corresponding implementation in Java is generated automatically and placed in `NodeImpl.java`. Just in case you didn't get it: The behavioural or dynamic aspects of a system can be completely modelled in an abstract, platform (programming language) independent fashion using a blend of activity diagrams and a “graph pattern” language called *Story Driven Mod-*

elling (SDM). In our EA project, these *Story Diagrams* or simply *SDMs*, are placed in SDM Containers named according to the method they implement. E.g. `<<SDM Activity>> insertNodeAfter SDM` for the method `insertNodeAfter(Node)` as depicted in Fig. 4.7. We'll dedicate Part III of the handbook to understanding why SDMs are so **Crazily cool!**

To recap all we've discussed, let's consider the complete workflow as depicted in Fig. 4.8. We started with a concise model in EA, simple and independent of any platform specific details (1). Our EA model consists not only of static aspects modelled as a class diagram (2), but also of dynamic aspects modelled using SDM (3). After exporting the model and code generation (4), we basically switch from *modelling* to *programming* in a specific general purpose programming language (Java). On this lower *level of abstraction*, we can flesh out the generated repository (5) if necessary, and mix as appropriate with hand-written code and libraries. Our abstract specification of behaviour (methods) in SDM is translated to a series of method calls that form the body of the corresponding Java method (6).

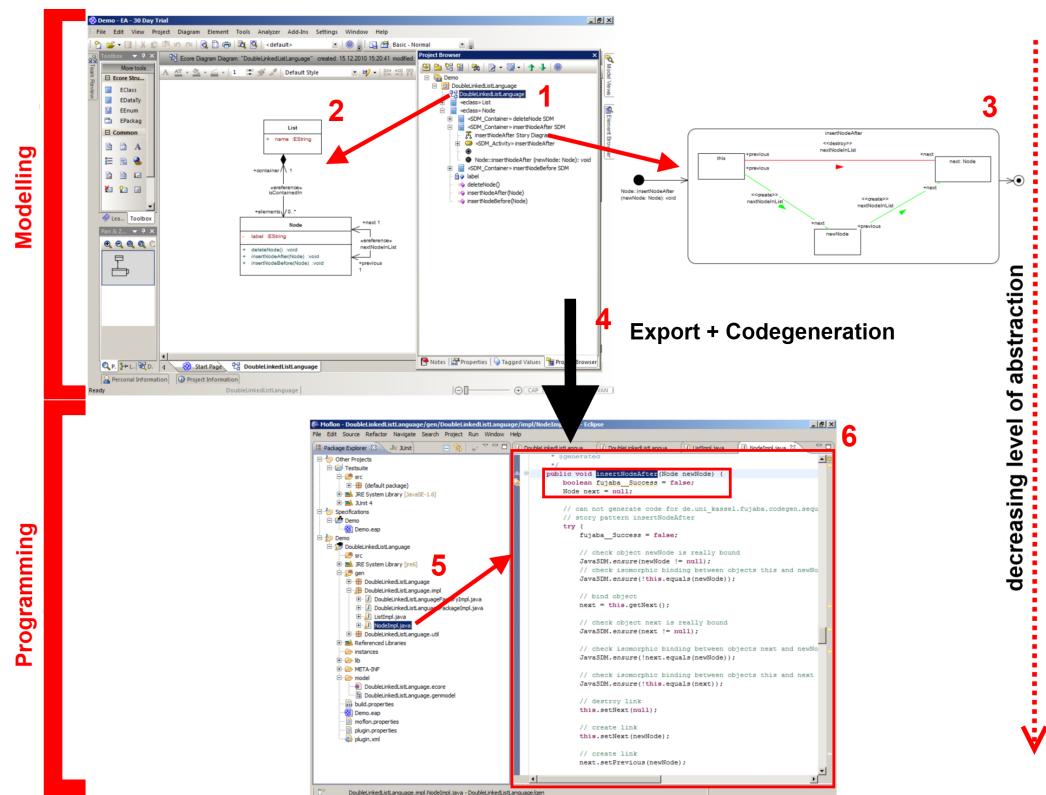


Figure 4.8: Overview

▷ Next

4.2 Your MOSL workspace

As you now know, eMoflon is a plug-in for Eclipse. More precisely, eMoflon requires the Eclipse Modeling Framework (EMF) in order to work. EMF uses two separate models, a Genmodel and an Ecore model, for code generation. The Genmodel contains boring information about code generation such as path, file prefixes, and other information. We are more interested in the Ecore model, which we specify with MOSL.

When you switched the “Top Level Elements” from **Projects** to **Working Sets**, you noticed that a few extra nodes were displayed in the project browser. Each node you see has different criteria for grouping related Eclipse Projects together, which makes them your project *working sets*.

The **Specifications** working set contains all *metamodel projects* in a workspace (Fig. 4.9). This means that for every new metamodel you create in your current workspace, all of the relevant files will be placed here.

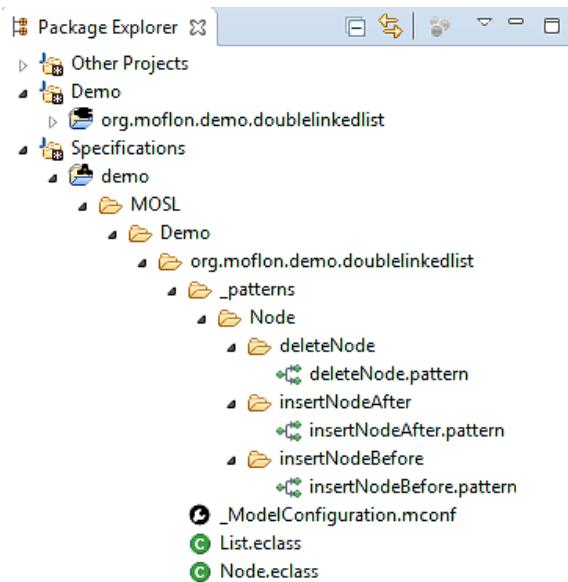


Figure 4.9: Specification working set

Let's have a look at the two *eclasse* files. While you can combine several short class declarations in a single file in some languages (such as Java), each class is kept separated in its own file when using MOSL.

Inspect the file **List.eclass** (Listing 4.1), and you'll see it has just one *EAttribute*. EAttributes are defined by their name, followed by a colon and type (line 3).

This class also has one *EReference*, a *container reference*. This is represented by the diamond operator in front of a long arrow (two minuses and a greater than) which points at a type. In the middle of the arrow the EReference name is represented, followed by their multiplicity (line 6).

```

1  class List
2  {
3      name : EString
4
5      // Sets a container reference to hold an unknown number of 'node'
6      <> - elements(0..*) -> Node
7 }
```

Listing 4.1: The `List` eclass

Switch to the `Node` eclass (Listing 4.2) and you can observe the second reference type, a *simple reference*. This reference type will be created by writing a container reference without a diamond (line 6).

In the `Node` eclass , a few methods have been declared. You can see that each function is remarkably small. In fact, the only thing the functions are doing are invoking *patterns*. These patterns represent structural changes, and their container functions are used exclusively for control flow (i.e., sequences, branches, and loops).

```

1  class Node
2  {
3      label : EString
4
5      // Set up simple references
6      - container(0..1) -> List
7      - next(1..1) -> Node
8      - previous(1..1) -> Node
9
10     // Destroys all references to this node and repairs those that
11     // remain
12     deleteNode() : void
13     {
14         [deleteNode]
15         return
16     }
17
18     // Sets 'next' reference of current node to 'newNode,' updating
19     // other nodes where required
20     insertNodeAfter(newNode : Node) : void
21     {
22         [insertNodeAfter]
23         return
24     }
25 }
```

```

24  // Sets 'previous' reference of current node to 'newNode' and
25  // updates any others
26  insertNodeBefore(newNode: Node) : void
27  {
28      [insertNodeBefore]
29      return
30  }

```

Listing 4.2: The Node eclass

After many long discussions, it was decided that patterns should always be implemented in separate files. Inspect Fig. 4.9 again, and observe the locations where the patterns are placed. You'll notice that there is a folder for the `Node` EClass, and a subfolder for each method. There's no folder for `List` because it never calls a pattern.

Check out the `deleteNode` pattern (Listing 4.3). You can see that there is a destroy command on `@ this`, denoted by `--`. It gets rid of the node, but it doesn't do anything about the previous and next references defined on it (remember that we're dealing with a double linked list here!). That's because when the node is removed, everything attached to it will be automatically cleaned and removed. The final command reconnects the next and previous nodes of the deleted node to close the 'hole' in the list. This is accomplished by setting `(++)` the previous reference of the next node to the previous node.

```

1  pattern deleteNode
2  {
3
4      // repairs 'next' link
5      @ this : Node
6      {
7          ++ - next -> newNext
8      }
9
10     // delete next Node
11     -- next : Node
12     {
13         -- - next -> newNext
14     }
15
16     newNext : Node
17 }

```

Listing 4.3: The deleteNode pattern

So that's a quick overview of the MOSL language but how do we generate code from all of this?

First, eMoflon does not generate code with every change. To improve performance, only the parser is invoked when you save files. This means that to generate code for the project, you need to either invoke the `Build` command from the eMoflon context menu after right clicking your metamodel project, or by navigating to the “Build (dirty projects)” button next to the “New Metamodel” button. Cleaning first deletes all generated files while building without cleaning tries to merge the newly generated code into existing files. We’ll discuss this in more detail later in the handbook.

5 Generated code vs. hand-written code

Now that you've worked through the specifics of your syntax, lets have a brief discussion on code generation.

The Ecore model is used to drive a code generator that maps the model to Java interfaces and classes. The generated Java code that represents the model is often referred to as a repository. This is the reason why we refer to such projects as repository projects. A repository can be viewed as an adapter that enables building and manipulating concrete instances of a specific model via a programming language such as Java. This is why we indicate repository projects using a cute adapter/plug symbol on the project folder.

If you take a careful look at the code structure in `gen` (Fig. 5.1), you'll find a `FooImpl.java` for every `Foo.java`. Indeed, the subpackage `.impl` contains Java classes that implement the interfaces in the parent package. Although this might strike you as unnecessary (why not merge interface and implementation for simple classes?), this consequent separation in interfaces and implementation allows for a clean and relatively simple mapping of Ecore to Java, even in tricky cases such as multiple inheritance (allowed and very common in Ecore models). A further package `.util` contains some auxiliary classes such as a factory for creating instances of the model.

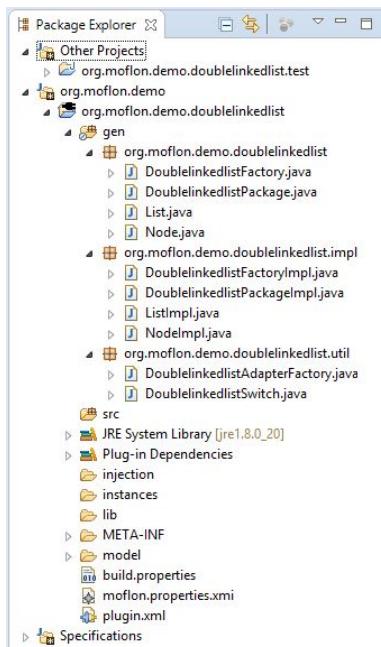


Figure 5.1: Package structure of generated code (`gen`)

If this is your first time of seeing generated code, you might be shocked at the sheer amount of classes and code generated from our relatively simple model. You might be thinking: “Hey – if I did this by hand, I wouldn’t need half of all this stuff!” Well, you’re right and you’re wrong. The point is that an automatic mapping to Java via a code generator scales quite well.

This means for simple, trivial examples (like our double linked list), it might be possible to come up with a leaner and simpler Java representation. For complex, large models with lots of mean pitfalls however, this becomes a daunting task. The code generator provides you with years and *years* of experience of professional programmers who have thought up clever ways of handling multiple inheritance, an efficient event mechanism, reflection, consistency between bidirectionally linked objects, and much more.

A point to note here is that the mapping to Java is obviously not unique. Indeed there exist different standards of how to map a modelling language to a general purpose programming language such as Java. As previously mentioned, we use a mapping defined and implemented by the Eclipse Modelling Framework (EMF), which tends to favour efficiency and simplicity over expressiveness and advanced features.

6 Conclusion and next steps

Congratulations – you’ve finished Part I! If you feel a bit lost at the moment, please be patient. This first part of the handbook has been a lot about installation and tool support, and only aims to give a very brief glimpse at the big picture of what is actually going on.

If you enjoyed this section and wish to get started on the key features of eMoflon, Check out Part II⁸! There we will work through a hands-on, step-by-step example and cover the core features of eMoflon.

We shall also introduce clear and simple definitions for the most important metamodeling and graph transformation concepts, always referring to the concrete example and providing references for further reading.

If you’re already familiar with the tool, feel free to pick and choose individual parts that are most interesting to you. Check out Story Driven Modeling (SDMs) in Part III⁹, or Triple Graph Grammars (TGGs) in Part IV¹⁰. We’ll provide instructions on how to easily download all the required resources so you can jump right in. For further details on each part, refer to Part 0¹¹.

Cheers!

⁸Download: <http://tiny.cc/emoflon-rel-handbook/part2.pdf>

⁹Download: <http://tiny.cc/emoflon-rel-handbook/part3.pdf>

¹⁰Download: <http://tiny.cc/emoflon-rel-handbook/part4.pdf>

¹¹Download: <http://tiny.cc/emoflon-rel-handbook/part0.pdf>

An Introduction to Metamodelling and Graph Transformations

with eMoflon



Part II: Ecore

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

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

The eMoflon team
Darmstadt, Germany (August 2015)

Contents

1	A language definition problem?	2
2	Abstract syntax and static semantics	5
3	Creating instances	49
4	eMoflon's graph viewer	53
5	Introduction to injections	55
6	Leitner's Box GUI	60
7	Conclusion and next steps	62
	Glossary	63

Part II:

Leitner's Learning Box

Approximate time to complete: 1 hour

URL of this document: <http://tiny.cc/emoflon-rel-handbook/part2.pdf>

The toughest part of learning a new language is often building up a sufficient vocabulary. This is usually accomplished by repeating a long list of words again and again until they stick. A *Leitner's learning box*¹ is a simple but ingenious little contraption to support this tedious process of memorization.

As depicted in Fig. 0.1, it consists of a series of compartments or partitions usually of increasing size. The content to be memorized is written on a series of cards which are initially placed in the first partition. All cards in the first partition should be repeated everyday and cards that have been successfully memorized are placed in the next partition. Cards in all other partitions are only repeated when the corresponding partition is full and cards that are answered correctly are moved one partition forward in the box. Challenging cards that have been forgotten are treated as brand new cards and are always returned to the first partition, regardless of how far in the box they have progressed.

These “rules” are depicted by the green and red arrows in Fig. 0.1. The basic idea is to repeat difficult cards as often as necessary and not waste time on easy cards which are only repeated now and then to keep them in memory. The increasing size of the partitions represent how words are easily placed in our limited short term memory and slowly move in our theoretically unlimited long term memory if practised often enough.

¹http://en.wikipedia.org/wiki/Leitner_system

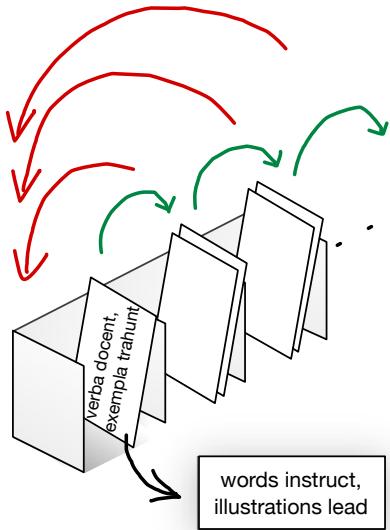


Figure 0.1: Possible *concrete syntax* of a Leitner's learning box

A learning box is an interesting system, because it consists clearly of a static structure (the box, partitions with varying sizes, and cards with two sides of content) and a set of rules that describe the dynamic aspects (behaviour) of the system. In this Part II of the eMoflon handbook, we shall build a complete learning box from scratch in a model-driven fashion and use it to introduce fundamental concepts in metamodeling and *Model-Driven Software Development* (MDSD) in general. We'll begin with a short discussion, then develop the box's abstract syntax, learn about eMoflon's validation, create some dynamic instances of the box, use injections to integrate a handwritten implementation of a method with generated code from the model, and get to know a small GUI that will allow us to take the implemented learning box for a spin.

1 A language definition problem?

As in any area of study, metamodeling has its fair share of buzz words used by experts to communicate concisely. Although some concepts might seem quite abstract for a beginner, a well defined vocabulary is important so we know exactly what we are talking about.

The first step is understanding that metamodeling equates to language definition. This means that the task of building a system like our learning box can be viewed as defining a suitable language that can be used to describe

such systems. This language oriented approach has a lot of advantages including a natural support for product lines (individual products are valid members of the language) and a clear separation between platform independent and platform specific details.

So what constitutes a language? The first question is obviously what the building blocks of your language “look” like. Will your language be textual?

Visual? This representation is referred to as the *concrete syntax* of a language and is basically an interface to end users who use the language. In the case of our learning box, Fig. 0.1 can be viewed as a possible concrete syntax. As we are building a learning box as a software system however, our actual concrete syntax will be composed of GUI elements like buttons, drop-down menus and text fields.

Concrete Syntax

Irrespective of what a language looks like, members of the language must adhere to the same set of “rules”. For a natural language like English, this set of rules is usually called a *grammar*. In metamodeling, however, everything is represented as a graph of some kind and, although the concept of a *graph grammar* is also quite well-spread and understood, metamodelers often use a *type graph* that declaratively defines what types and relations constitute a language.

Grammar

Graph

Grammar

Type Graph

A graph that is a member of your language must *conform* to the corresponding type graph for the language. To be more precise, it must be possible to type the graph according to the type graph - the types and relations used in the graph must exist in the type graph and not contradict the structure defined there. This way of defining membership to a language has many parallels to the class-object relationship in the object-oriented (OO) paradigm and should seem very familiar for any programmer used to OO. This type graph is referred to as the *abstract syntax* of a language.

Abstract Syntax

Often, one might want to further constrain a language, beyond simple typing rules. This can be accomplished with a further set of rules or constraints that members of the language must fulfil in addition to being conform to the type graph. These further constraints are referred to as the *static semantics* of a language.

Static Semantics

With these few basic concepts, we can now introduce a further and central concept in metamodeling, the *metamodel*, which is basically a simple class diagram. A metamodel defines not only the abstract syntax of a language but also some basic constraints (a part of the static semantics).

Metamodel

In analogy to the “everything is an object” principle in the OO paradigm, in metamodeling, everything is a model! This principle is called *unification*, and has many advantages. If everything is a model, a metamodel that defines (at least a part of) a language must be a model itself. This means that it con-

Unification

Modelling Language

forms to some *meta-metamodel* which in turn defines a *(meta)modelling language* or *meta-language*. For metamodeling with eMoflon, we support *Ecore* as a modelling language and it defines types like **EClass** and **EReference**, which we will be using to specify our metamodels. Alternate modelling languages include MOF, UML and Kermeta.

*Meta-metamodel
Meta-Language*

Thinking back to our learning box, we can define the types and relations we want to allow. We want an entire box of flashcards where each card is contained within a partition, and each partition is contained within the box. Multiplicities are an example of static semantics that do not belong to the abstract syntax, but can nonetheless be expressed in a metamodel. An example could be that a card can only ever exist in one partition, or that a partition can have either one **next** partition, or none at all.

More complex constraints that cannot be expressed in a metamodel are usually specified using an extra *constraint language* such as the Object Constraint Language (OCL). This idea, however, is beyond the scope of this handbook. We'll stick to metamodels without an extra constraint language.

Constraint Language

In addition to its static structure, every system has certain dynamic aspects that describe the system's behaviour and how it reacts to external stimulus or evolves over time. In a language, these rules that govern the dynamic behaviour of a system are referred to collectively as the *dynamic semantics* of the language. Although these rules can be defined as a set of separate *model transformations*, we take a holistic approach and advocate integrating the transformations directly in the metamodel as operations. This naturally fits quite nicely into the OO paradigm.

Dynamic Semantics

A short recap: We have learned that metamodeling starts with defining a suitable language. For the moment, we know that a language comprises a concrete syntax (how the language “looks”), an abstract syntax (types and relations of the underlying graph structure), and static semantics (further constraints that members of the language must fulfil). Metamodels are used to define the abstract syntax, and a part of the static semantics of a language, while *models* are graphs that conform to some metamodel (i.e., can be typed according to the abstract syntax and must adhere to the static semantics).

Model

This handbook is meant to be hands-on, so enough theory! Lets define, step-by-step, a metamodel for a learning box using our tool, eMoflon.

2 Abstract syntax and static semantics

The first step in creating any metamodel is defining the abstract syntax, also known as the type graph. This involves defining each class, its attributes, references, and method signatures.

If you completed the demo in Part I, your Eclipse workspace will look slightly different than ours depicted in the screenshots. In an effort to keep things as clear as possible, we have removed those files from our package explorer, but still recommend keeping them for future reference.

Additionally, if you're continuing from the visual demo, you can begin modelling this project in two different ways. You can either develop your metamodel in the same workspace as the demo, or create a new one. Either way, please note that the steps are exactly the same, but our project browsers in EA may not exactly match. This handbook has assumed you prefer the latter.

2.1 Getting started in EA

- To begin, navigate to “New Metamodel Project,” and start a new visual project (this time without the demo specifications) named **LeitnersLearningBox** (Fig. 2.1). Open the empty .eap file in EA.

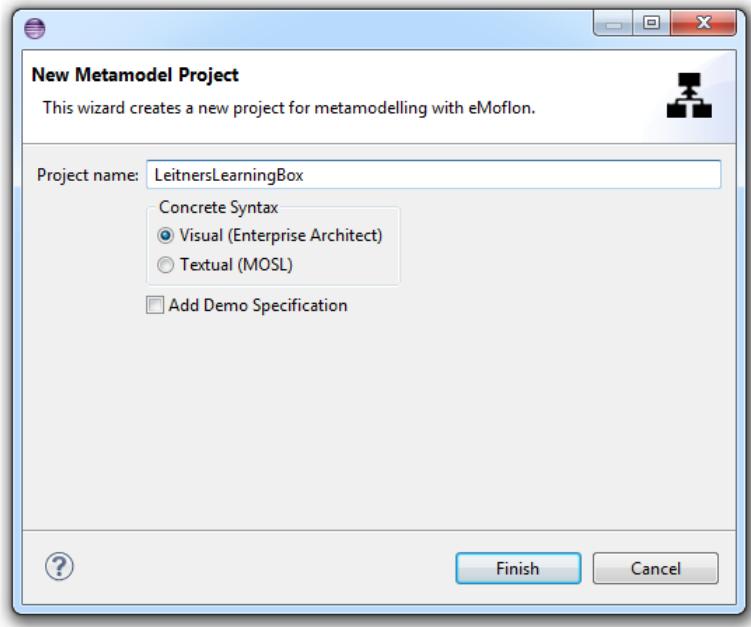


Figure 2.1: Starting a new visual project

- In EA, select your working set and press the “Add a Package” button (Fig. 2.2).



Figure 2.2: Add a new package to MyWorkingSet

- In the dialogue that pops up (Fig. 2.3), enter **LearningBoxLanguage** as the name of the new package. In this case select **Package Only** and click **OK**. Later you can select **Create Diagram** to skip the next step.

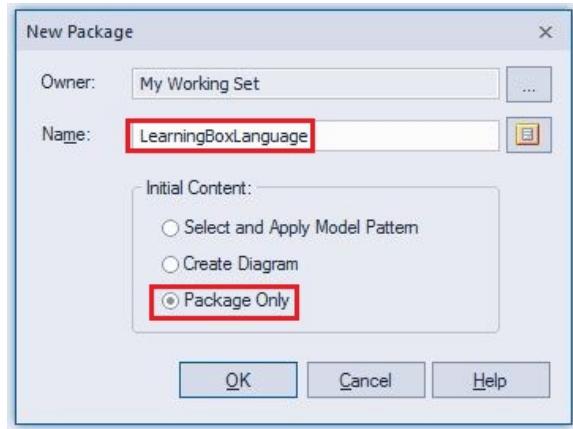


Figure 2.3: Enter the name of the new package

- Your Project Browser should now resemble Fig. 2.4.

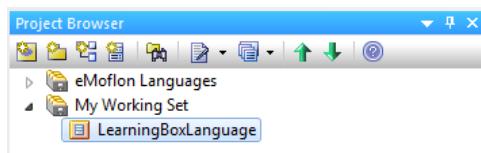


Figure 2.4: State after creating the new package

- Now select your new package and create a “New Diagram” (Fig. 2.5).

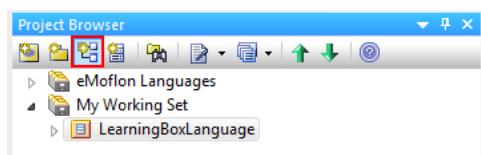


Figure 2.5: Add a diagram

- In the dialogue that appears (Fig. 2.6), choose **eMoflon Ecore Diagrams** and press **OK**.

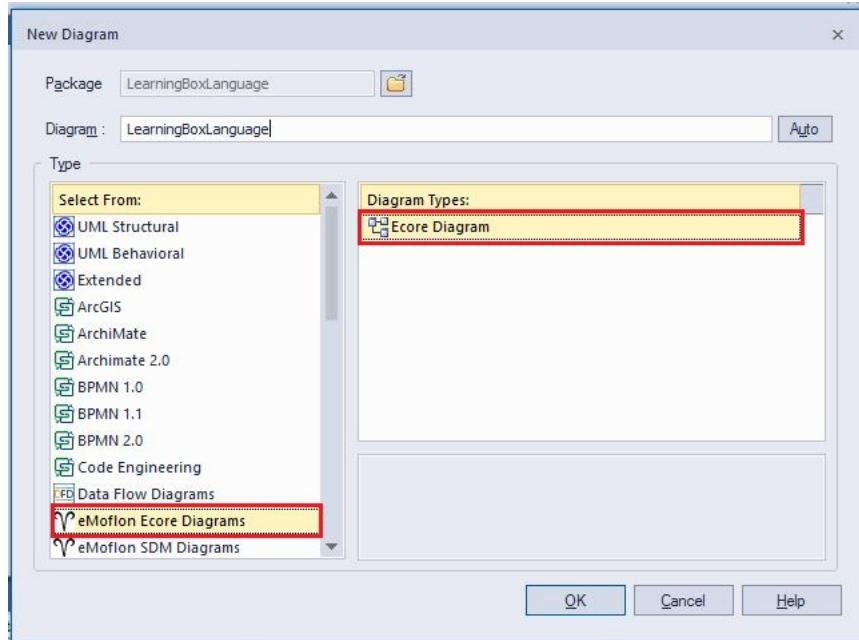


Figure 2.6: Select the ecore diagram type

- After creating the new diagram, your **Project Browser** should now resemble Fig. 2.7. You'll notice that your **LearningBoxLanguage** package has been transformed into an EPackage.

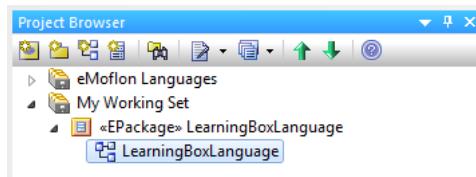


Figure 2.7: State after creating diagram

- You can now already export your project to Eclipse,² then refresh your **Package Explorer**. A new node, **My Working Set**³ should have appeared containing your EPackage (Fig. 2.8). You can see that a **LearningBoxLanguage.ecore** file has been generated, and placed in "model." This is your metamodel that will contain all future types you create in your diagrams.

²If unsure how to perform this step, please refer to Part I, Section 2.1

³If you do not have the two distinct nodes, ensure your "Top Level Elements" are set to **Working Sets**

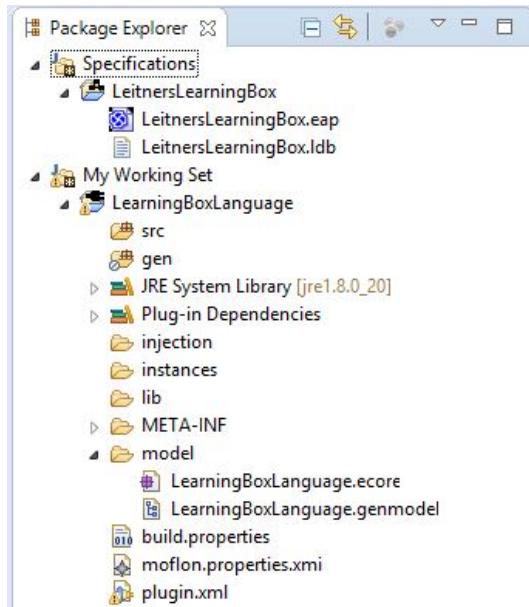


Figure 2.8: Initial export to Eclipse

- If you're interested in reviewing the overall project structure and the purposes of certain files and folders, read Section 4.1 from Part I. Otherwise, continue to the next section to learn how to declare classes and attributes.

▷ [Next](#)

2.2 Getting started with MOSL

Please note that the textual syntax is not as thoroughly tested as the visual syntax because most of our projects are built with the visual syntax. This means: Whenever something goes wrong even though you are sure to have followed the instructions precisely, do not hesitate to contact us via contact@emoflon.org.

- ▶ Create a new metamodel project in Eclipse by navigating to the **New Metamodel** button in the toolbar. In the dialogue that appears, enter **LeitnersLearningBox** as the project name, and select **Textual (MOSL)** (Fig. 2.9).

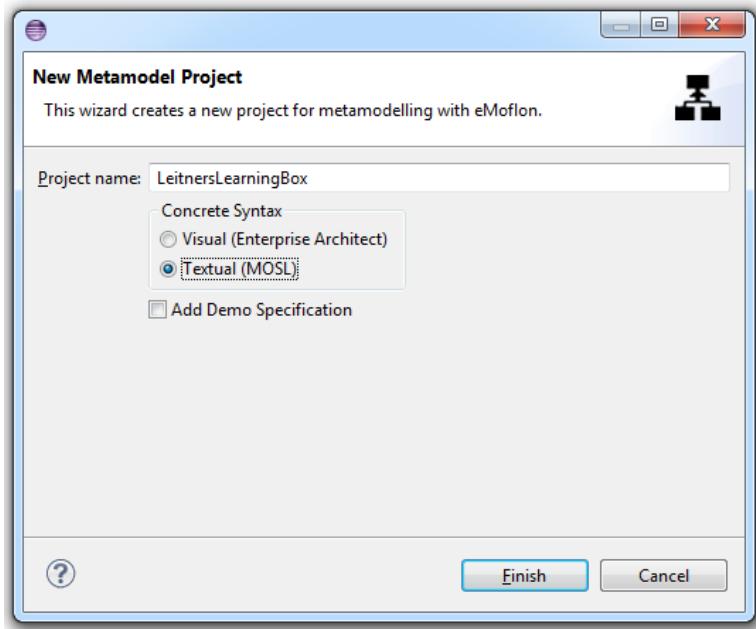


Figure 2.9: Creating a new metamodel project

- ▶ You'll see your new project appear under the “Specifications” node.⁴ If you're interested in the details of eMoflon's project structure, review Section 4.2 from Part I. Otherwise, expand the project as deep as it goes (Fig. 2.10).

⁴If no nodes appear in your package explorer, ensure your “Top Level Elements” are set to “WorkingSets”

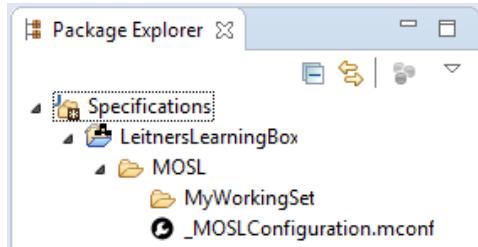


Figure 2.10: Expanded project files

- We're most interested in MOSL/MyWorkingSet, which represents the project scope. Right click this folder, and create a new EPackage (Fig. 2.11), naming it LearningBoxLanguage. Using this wizard to generate your EPackage will automatically create the necessary _patterns folder and _ModelConfiguration file.

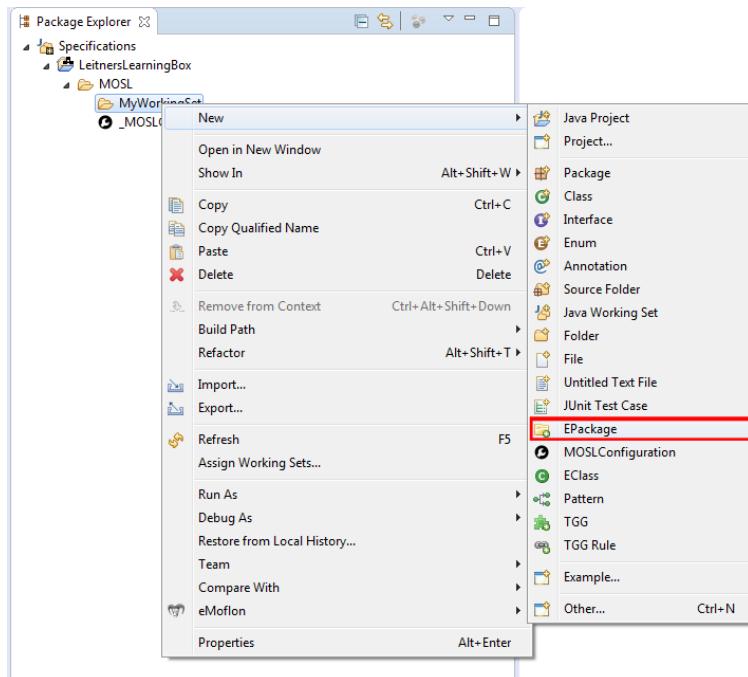


Figure 2.11: Create an EPackage in your working set

- ▶ Your package explorer should now resemble Fig. 2.12.
- ▶ To finish initialising your metamodel, navigate to “Build (Without cleaning),” found beside “New Metamodel” in the toolbar (Fig. 2.12).

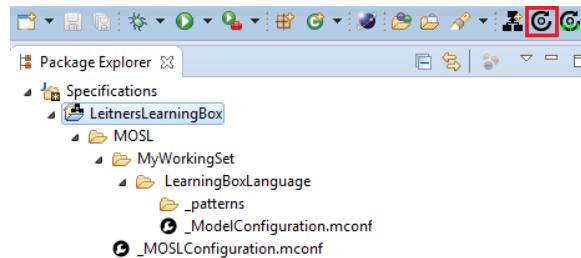


Figure 2.12: Initial structure of Leitner’s Learning Box

- ▶ A new, “MyWorkingSet” node, named after your project container, should have been created (Fig. 2.13). The folder “gen” is where all Java files generated from your metamodel will be placed.

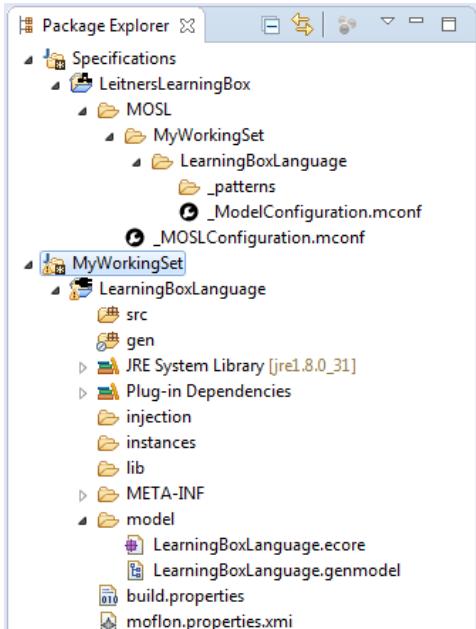


Figure 2.13: The project fully initialised

- ▶ Navigate to “LearningBoxLanguage/model.” This folder contains `LearningBoxLanguage.ecore`, your metamodel. This will contain all

types you define in “LeitnersLearningBox/MOSL/MyWorkingSet/LearningBoxLanguage.”

- Your project structure is now complete! In the next section, we’ll start creating classes and attributes.

2.3 Declaring classes and attributes

- ▶ Return to EA, and double-click your `LearningBoxLanguage` diagram to ensure it's open.
- ▶ There are two ways for you to create your first `EClass`. First, to the left of the workbench, a *Toolbox* containing the Ecore types available for metamodeling should have appeared (Fig. 2.14).⁵ Click on the `EClass` icon then somewhere in the diagram to create a new object. Alternatively, you can click in the diagram and press `space` to invoke the toolbox context menu, then select `EClass`.

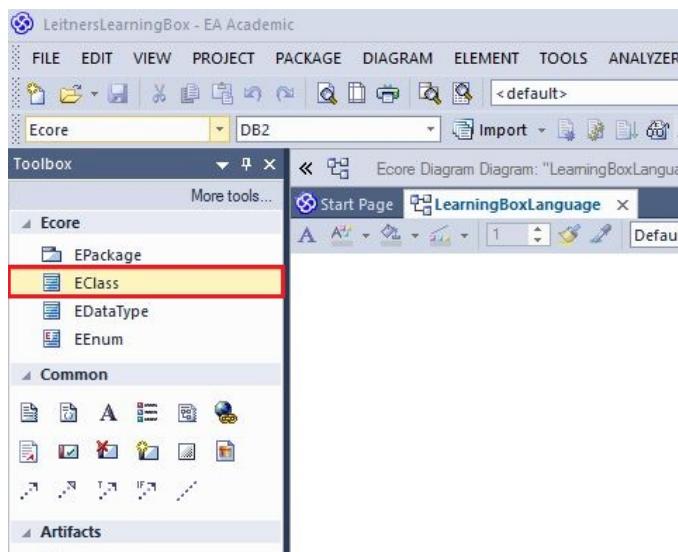


Figure 2.14: Create an EClass

- ▶ In the dialogue that pops-up, set `Box` as the name and click `OK` (Fig. 2.15). This dialogue can always be invoked again by double-clicking the `EClass`, or by pressing `Alt` and single-clicking. It contains many other properties that we'll investigate later in the handbook. In general, a similar properties dialogue can be opened in the same fashion for almost every element in EA.

⁵If not, choose “Diagram/Diagram Toolbox” to show the current toolbox (Alt+ 5)

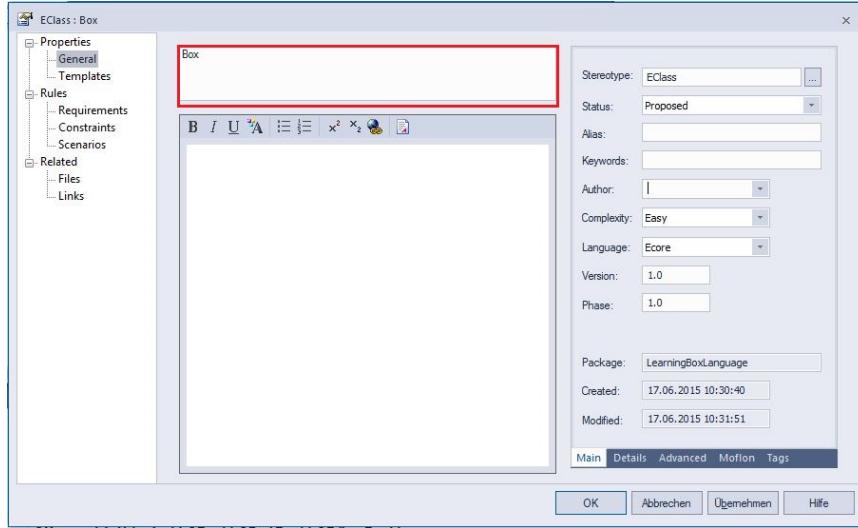


Figure 2.15: Edit the properties of an EClass

- After creating Box, your EA workspace should resemble Fig. 2.16.

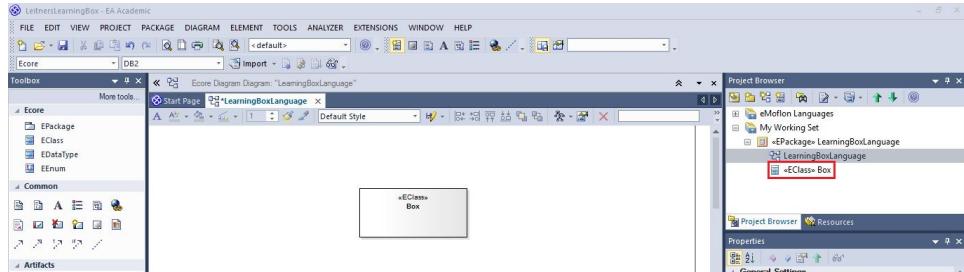


Figure 2.16: State after creating Box

- Now create the Partition and Card EClasses the same way, until your workspace resembles Fig. 2.17. These are the main classes of your learning box metamodel.
- Lets add some attributes! Either right-click on Box to activate the context menu and choose “Features & Properties/Attributes..” (Fig. 2.18), or press F9 to open the editing dialogue.

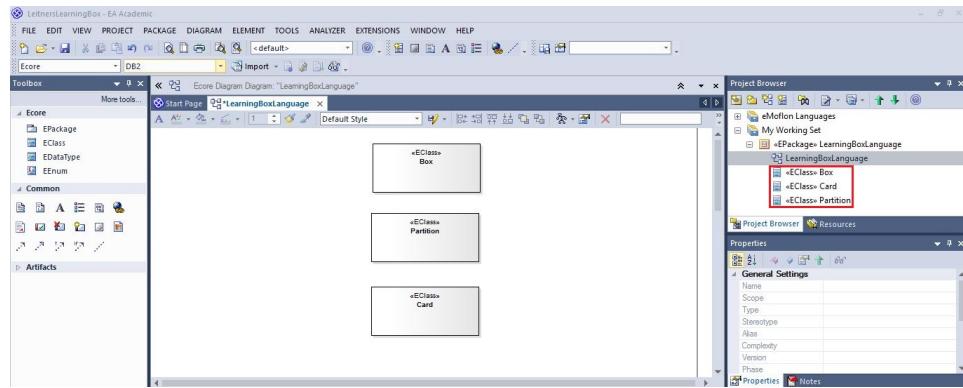


Figure 2.17: All EClasses for the metamodel

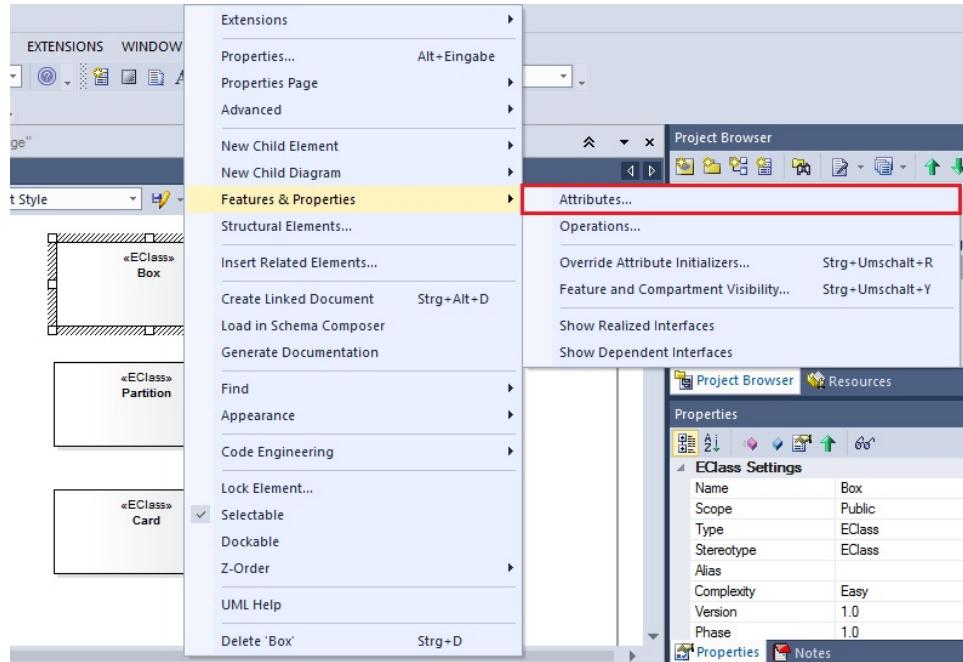


Figure 2.18: Context menu for an EClass

- ▶ Enter **name** as the name of the attribute, select **EString** as its type from the drop-down menu, and press **Close** (Fig. 2.19). New attributes for the same EClass can be added by clicking on **New Attribute...**.

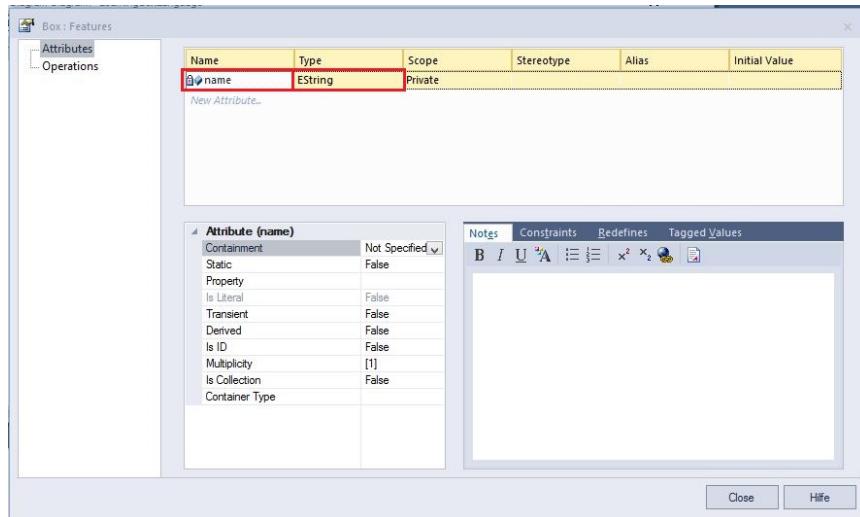


Figure 2.19: Adding attributes to an EClass

- ▶ Add the remaining attributes analogously to each EClass until your workspace resembles Fig. 2.20.
- ▶ Save and export to Eclipse. After refreshing your workspace, your **.ecore** model can now be expanded as it includes every class and attribute from your metamodel. So far, so good!

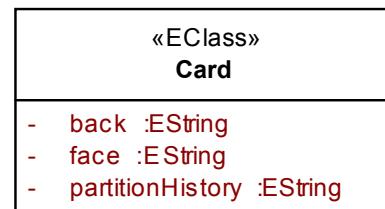
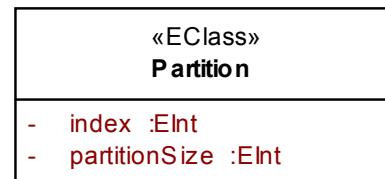
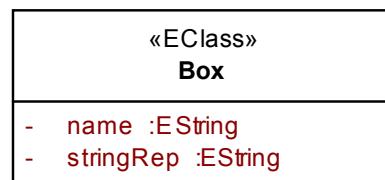


Figure 2.20: Main EClasses declared with their attributes

2.4 Declaring classes and attributes

- Right click your LearningBoxLanguage model and create your first EClass by navigating to “New/EClass.” Name it Box (Fig. 2.21).

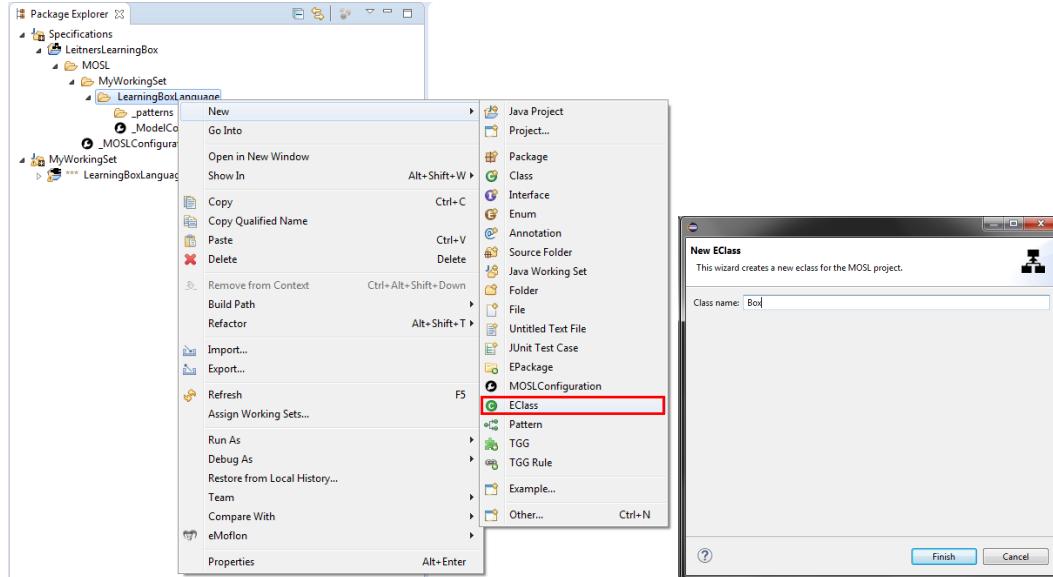


Figure 2.21: Creating a new EClass

- The class editor should automatically open. Let's add the first two EAttributes of our Box, `name` and `stringRep`. eMoflon offers auto-completion templates to help you with this task. Go to an empty line and press **Ctrl + Space**. You'll be provided with a short list of suggestions (Fig. 2.22). The first four items are related to method control flow, so select `attribute` near the bottom to create `name` of type `EString`.
- Your workspace should now resemble (Listing 2.1).

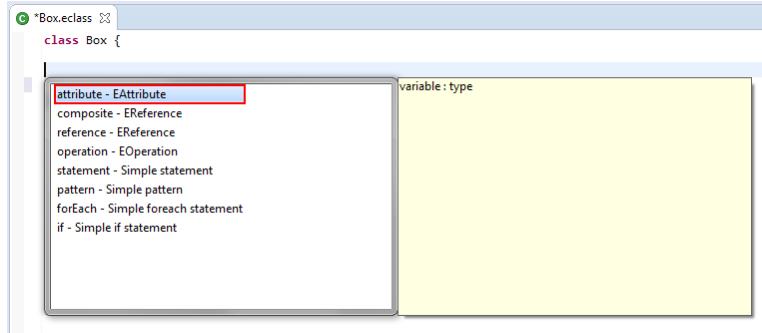


Figure 2.22: eMoflon's auto-completion

```

1 class Box
2 {
3     name : EString
4     stringRep : EString
5 }
```

Listing 2.1: Newly created Box EClass

- ▶ Now create two empty EClasses in your model, **Partition** and **Card**.
- ▶ In **Partition**, add two **EInt** attributes, **index** and **partitionSize** (Listing 2.2).

```

1 class Partition
2 {
3     index : EInt
4     partitionSize : EInt
5 }
```

Listing 2.2: Newly created Partition EClass

- ▶ In **Card**, create three **EString** attributes, **back**, **face** , and **partitionHistory** (Listing 2.3).

```

1 class Card
2 {
3     back : EString
4     face : EString
5     partitionHistory : EString
6 }
```

Listing 2.3: Newly created Card EClass

- ▶ If you've done everything correctly, your workspace should now resemble Fig. 2.23 and Listings 2.1, 2.2 and 2.3 .

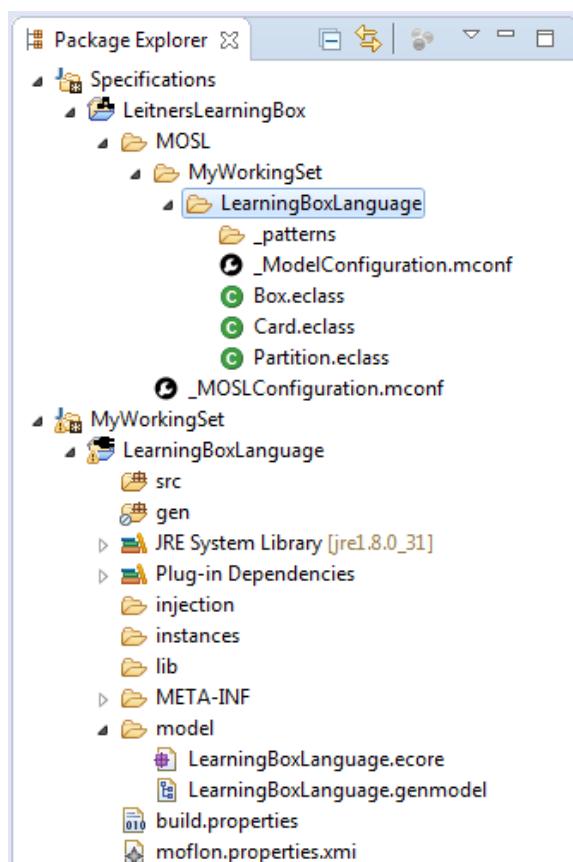


Figure 2.23: The project after creation of Box, Partition and Card

- That's it for declaring class attributes! Feel free to build your project again and view the changes in the `.ecore` mode, and the generated files in "gen" and "src." On a final note, while some languages (such as Java) allow the declaration of several small classes (such as these three) in the same file, when tooling with eMoflon, we keep them separated. Don't worry – we'll explain this later in the handbook. As for now, continue to the next section to start creating references between these EClasses.

2.5 Connecting your classes

At this point, you've declared your types and their attributes, but what good are those if you can't connect them to each other? We need to create some *EReferences*!

EReference

There are four properties that must be set in order to create an *EReference*: the target and source *Role*, *Navigability*, *Multiplicity* and *Aggregation*, all of which are declared differently in each syntax, but let's first review the concepts since they're basically the same.

A *Role* clarifies which way a reference is 'pointing.' In other words, the *Role source* and *target* roles determine the direction of the connection.

Navigable ends are mapped to class attributes with getters and setters in Java, and therefore *must* have a specified name and multiplicity for successful code generation. Corresponding values for *Non-Navigable* ends can be regarded as additional documentation, and do not have to be specified.

The *Multiplicity* of a reference controls if the relation is mapped to a (Java) *Multiplicity* collection ('*', '1..*', '0..*'), or to a single valued class attribute ('1', '0..1'). We'll explain this setting in detail later.

The *Aggregation* values of a reference can be either none or composite. Composite means that the current role is that of a *container* for the opposite role. You'll see in our example that *Box* is a container for several *Partitions*. This has a series of consequences: (1) every element must have a container, (2) an element cannot be in more than one container at the same time, and (3) a container's contents are deleted together with the container. Conversely, non-composite (denoted by "none") means that the current role is not that of a container, and the rules for containment do not hold (in other words, the reference is a simple 'pointer').

Aggregation
Container

Creating EReferences in EA

- A fundamental gesture in EA is *Quick Link*. Quick Link is used to create EReferences between elements in a context-sensitive manner. To use quick link, choose an element and note the little black arrow in its top-right corner (Fig. 2.24).

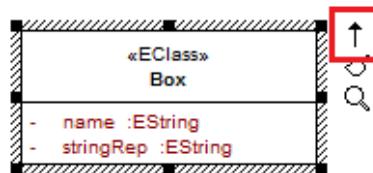


Figure 2.24: Quick Link is a central gesture in EA

- Click this black arrow and ‘pull’ to the element you wish to link to. To start, quick-link from Box to Partition. In the context menu that appears, select “Create Bidirectional EReference” (Fig. 2.25).

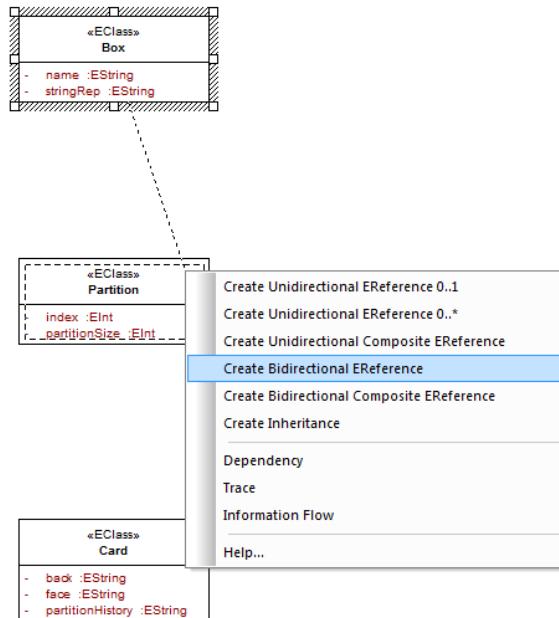


Figure 2.25: Create an EReference via Quick Link

- Double click the EReference to invoke a dialogue. In this window you can adjust all relevant settings. Feel free to leave the **Name** value blank - this property is only used for documentation purposes, and is not relevant for code generation.

- Within this dialogue, go to “Role(s),” and compare the relevant values in Fig. 2.26 for the *source* end of the EReference (the **Box** role). As you can see, the default source is set to the EClass you linked from, while the default target is the EClass you linked to. In this window, do not forget to confirm and modify the **Role**, **Navigability**, **Multiplicity**, and **Aggregation** settings for the source as required. Repeat the process for the **Target Role**.

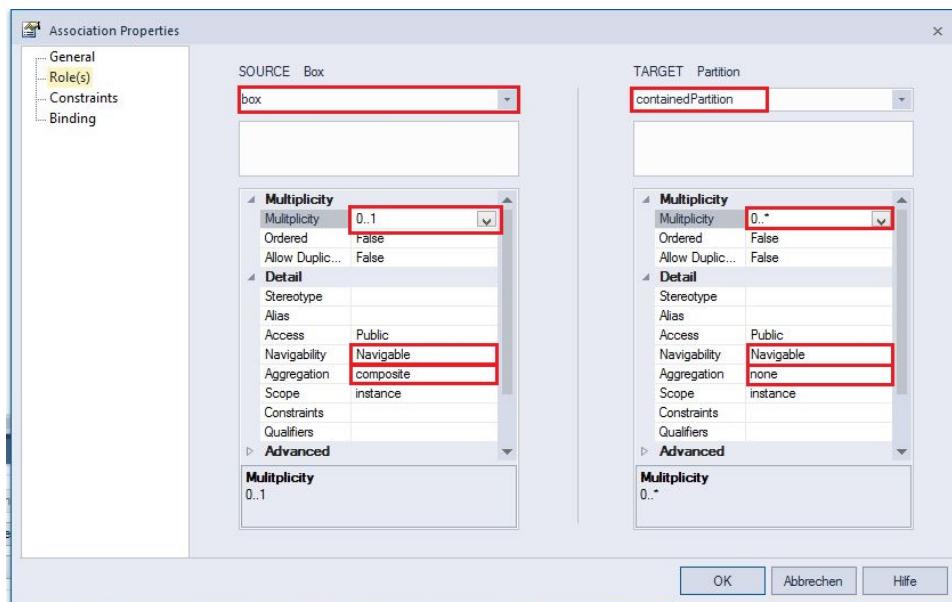


Figure 2.26: Properties for the source and target role of an EReference

To review these properties, the first value you edited was the role name. The **Navigability** value should have been automatically set to **Navigable**. Without these two settings, getter and setter methods will not be generated.

Next, you set the **Multiplicity** value. In your source role (**Box**), you have allowed the creation of up to one target (**Partition**) reference for every connected source (**box**). This means you could not have a single target connected to two sources (i.e., one partition that belongs to two boxes). In the target (**Partition**) role, you have specified that any source (in our case, **box**) can have any positive-sized number of targets. Figure 2.27 sketches this schematically.

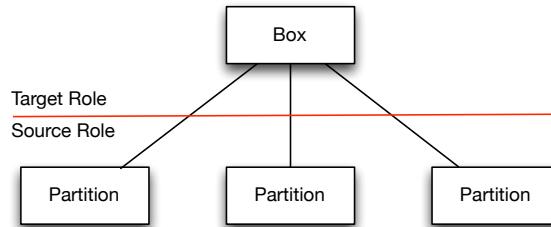


Figure 2.27: The target and source roles of Leitner's Learning Box

Finally, you set the **Aggregation** value. In this case, `box` is a container for `Partitions`, and `containedPartition` is consequently not.

- ▶ Take a moment to review how the **Aggregation** settings extend the **Multiplicity** rules. If you've done everything right, your metamodel should now resemble Figure 2.28, with a single *bidirectional EReference* between `Box` and `Partition`.

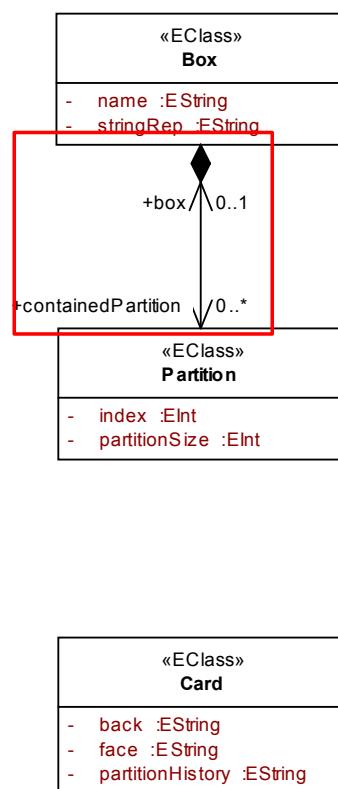


Figure 2.28: Box contains Partitions

- Following the same process, create two unidirectional self-ERefferences for **Partition** (for next and previous **Partitions**), and a second bidirectional composite EReference⁶ between **Partition** and **Card** (Fig. 2.29).

⁶To be precise, *all* EReferences in Ecore are actually unidirectional. A “bidirectional” EReference in our metamodel is really two mapped EReferences that are opposites of each other. We however, believe it is simpler to handle these pairs as single EReferences, and prefer this concise concrete syntax.

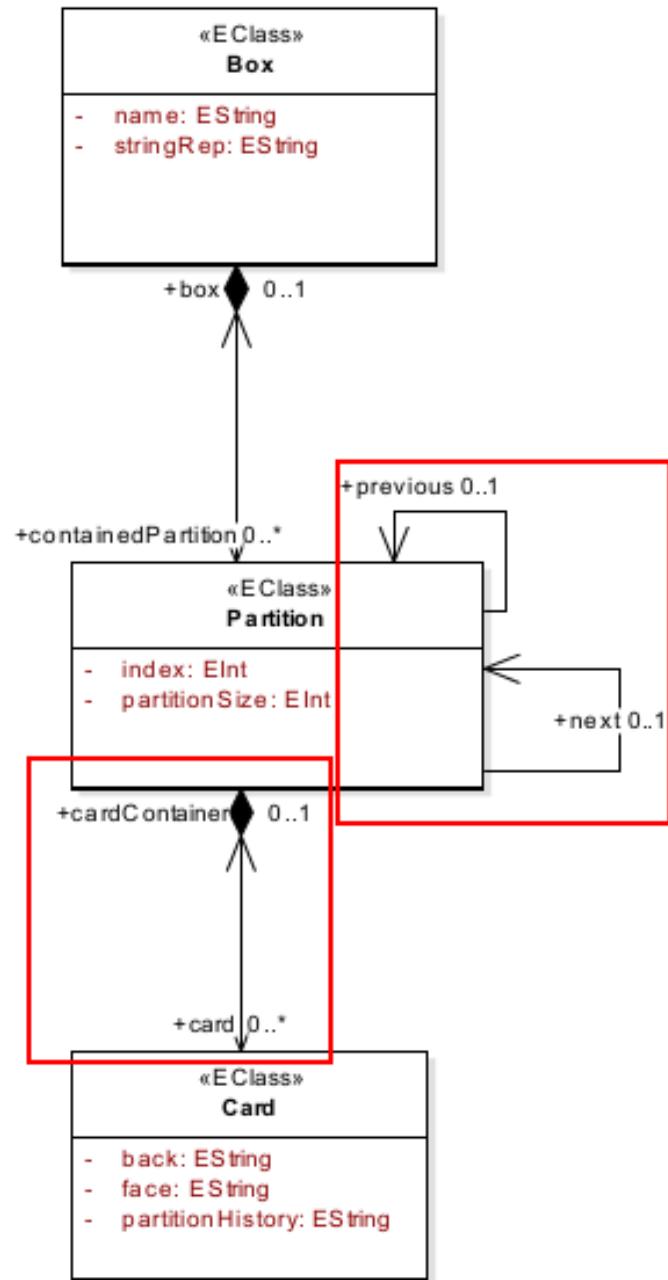


Figure 2.29: All relations in our metamodel

- ▶ You'll notice that the connection between **Card** and **Partition** is similar to that between **Partition** and **Box**. This makes sense as a partition should be able to hold an unlimited amount of cards, but a card can only belong to one partition at a time.

- ▶ Export your diagram to Eclipse and refresh your workspace. Your Ecore metamodel file in “model/LearningBoxLanguage.ecore” should now resemble Figure 2.30.

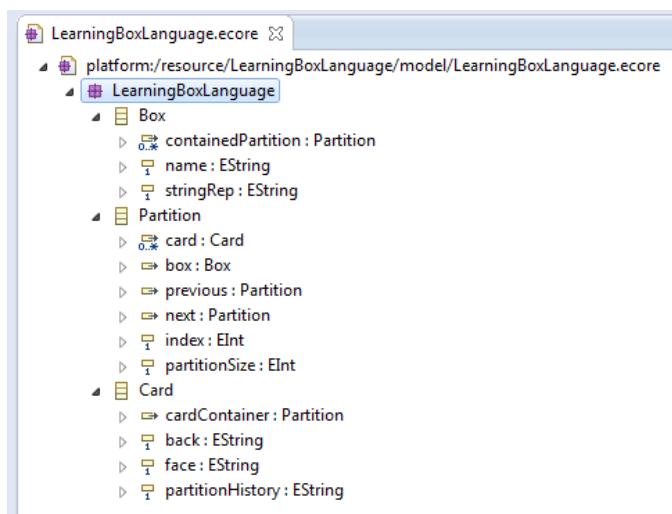


Figure 2.30: Refreshed Ecore file with all EReferences

- ▶ All the required attributes and references for your learning box have now been set up. We encourage you to see how these are declared in the textual syntax, starting on the immediate next page. In particular, check out Listings 2.4, 2.5 and 2.6, where each EClass is fully declared, and Listing 2.7, where bidirectionality is explicitly specified as a constraint.

Creating EReferences with MOSL

In MOSL, the declaration of an EReference is simple - you set each property according to the following syntax (specified in simple EBNF, if you know what that is):

```
[ '<>' ] '-' role_name '(' multiplicity ')' '->' target_type

With:
role_name := STRING
multiplicity := '0..1' | '0..*' | '1' | ...
target_type := STRING
```

The source type is determined by the EClass in which the EReference is placed. You can signal an aggregation EReference by including the sideways diamond before the arrow symbol. Don't worry - you don't have to remember this syntax. Our type completion provides a **reference** template when you activate the hot keys. Try it out!

- ▶ Open `Box.eclass` in the editor and add a *container reference* named `containedPartition` with a multiplicity of zero to infinity, from `Box` to `Partition` (Listing 2.4, Line 6). This EReference means a `Box` can hold an infinite number of partitions.
- ▶ Now add a *simple reference* to `Partition`. Name it `box`, and allow it to hold up to one `Box` (Listing 2.5, Line 7). This means a single partition can belong to either zero, or one `Box`, and that's it. It can't belong to two different boxes at the same time.
- ▶ Congratulations, you have just built your first pair of EReferences! To see how this is depicted visually, check out Fig. 2.28 from the previous subsection.
- ▶ Now, lets create another pair of EReferences between `Partition` and `Card`. If you think about it, it's really not all that different from the relation between `Box` and `Partition`. A `Partition` should be able to hold an unlimited amount of `Cards`, but a `Card` should only be allowed to belong to zero or one `Partitions`. Name the two new EReferences `card`, and `cardContainer` (Listing 2.5, Line 6 and Listing 2.6, Line 7).

- ▶ The next step is to construct two connections between **Partitions** so cards can be moved between their previous and next partitions in the box. Create two new simple references, named **previous**, and **next**, each with a $0..1$ multiplicity (Listing 2.5, Line 8 and 9).
- ▶ If you have done everything correctly, your EClasses should now resemble Listings 2.4, 2.5 and 2.6.

```

1 class Box
2 {
3   name : EString
4   stringRep : EString
5
6   <> - containedPartition (0..*) -> Partition
7 }
```

Listing 2.4: Creating a *container reference* in Box

```

1 class Partition
2 {
3   index : EInt
4   partitionSize : EInt
5
6   <> - card (0..*) -> Card
7   - box (0..1) -> Box
8   - next (0..1) -> Partition
9   - previous (0..1) -> Partition
10 }
```

Listing 2.5: Creating *references* in Partition

```

1 class Card
2 {
3   back : EString
4   face : EString
5   partitionHistory : EString
6
7   - cardContainer(0..1) -> Partition
8 }
```

Listing 2.6: Creating a *simple references* in Card

At this point, all of your EReferences have been created! The problem is, suppose you set the `containedPartition` EReference in a particular `Box`. That's great, you would now have the box containing one partition. However, if you went and examined that partition independently, its `box` EReference would still be null. We still need to set up the link between these EReferences so that when one is updated, the other will be too.

- ▶ Navigate to the `_ModelConfiguration.mconf` file. You can see it has a single `opposites` scope that's currently empty. Constraints follow the syntax below:

```
reference '<->' reference

With:
reference := reference_name ':' source_type
reference_name := STRING
source_type := STRING
```

This statement sets the two EReferences to be opposites of one another, i.e., the connection between EClasses will be bidirectional. As you can see, syntax here is slightly different than that of a standard EReference. Instead of the reference type trailing the colon operator, it has switched to become the source type.

- ▶ To begin, press `Ctrl + Space` and complete the template with the following:

```
containedPartition : Box <-> box : Partition
```

- ▶ Reviewing the `Partition` EClass, its easy to see that `previous` and `next` are certainly not opposites,⁷ but we do need to establish an opposing link between `card` and its `cardContainer`. Follow the same steps until your constraint file resembles Listing 2.7.

```
1 opposites
2 {
3     containedPartition : Box <-> box : Partition
4     card : Partition <-> cardContainer : Card
5 }
```

Listing 2.7: The completed constraints file

⁷Review the rules depicted in Fig. 0.1

- Now the EReferences for your learning box are complete! To see how each of the classes, attributes, and references are depicted in the visual syntax, check out Fig. 2.29 from section 2.1. Otherwise, build your project to make sure there are no errors, and continue to the next section to finalize the declaration of your EClasses.

2.6 Method Signatures

To finish defining our types, let's define the *signatures* of some operations *Operation Signature* that they'll eventually support.

- ▶ Select **Partition** and either right-click to invoke the context-menu (Fig. 2.31) and choose “Features & Properties/Operations..” or simply press F10.

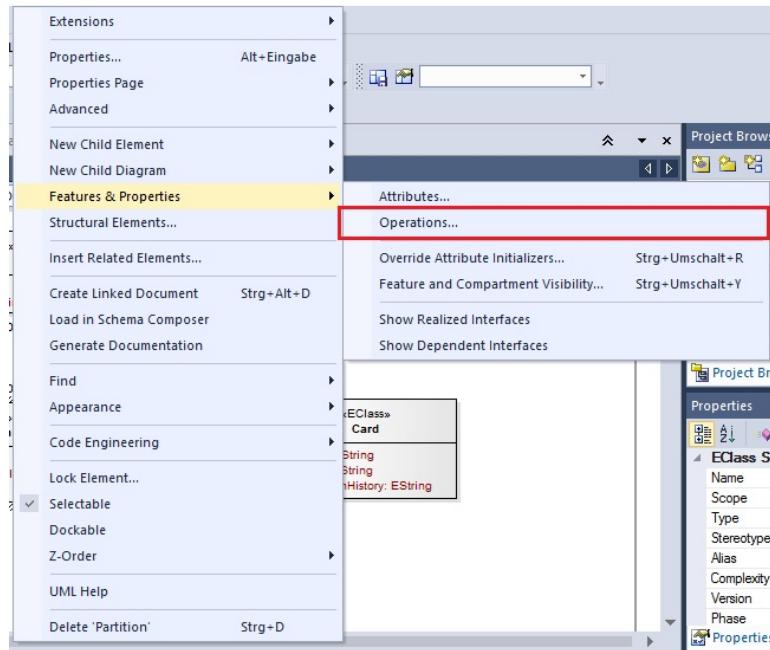


Figure 2.31: Add an operation

- ▶ In the dialogue that pops-up (Fig. 2.32), enter **empty** as the **Name** of the operation and **void** as the **Return Type**.
- ▶ In the same dialogue, click on **New Operation...** to add a second operation, **removeCard**, and edit the values as seen in Figure 2.33. Notice that the **Return Type** can be chosen by either the drop-down menu, or via direct typing. For types you've established in the metamodel (e.g. **Card**) you have to use ‘**Select Type...**’ from the drop-down menu.

Very important: Non-primitive types *must* be chosen via ‘**Select Type...**’ in the drop-down menu. It allows you to browse for the corresponding elements in your project. Simply typing them won't work!

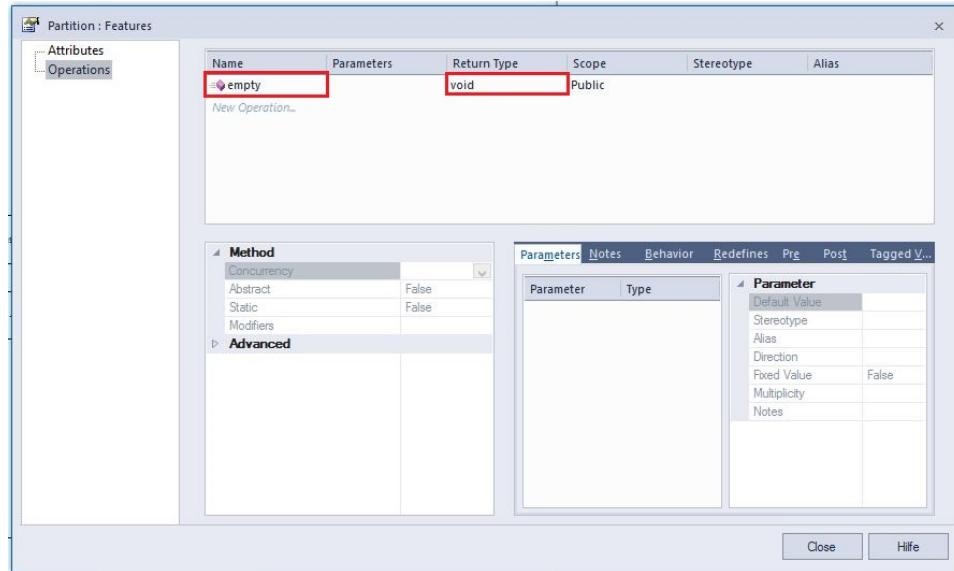


Figure 2.32: EClass properties editor

- ▶ Parameters can be added by selecting **Parameters** and completing the dialogue (Fig. 2.33). Please remember that you must also use either the drop-down menu, or direct typing to select the type or else validation will fail.
- ▶ Repeat this process for the `check` operation (with the two parameters `card:Card`, `guess:EString`) that returns an `EBoolean`.
- ▶ If you've done everything right, your dialogue should now contain three methods - `check`, `empty`, and `removeCard` - each with the corresponding parameters and return types in Fig. 2.34.
- ▶ Add all operations analogously for `Box` and `Card` until your metamodel closely resembles Figure 2.35.⁸
- ▶ To finish, export the metamodel for code generation in Eclipse, and examine the model once again. Each signature should have appeared in their respective EClass.
- ▶ To see how this complete metamodel is represented in the textual syntax, examine Listings 2.10, 2.8 and 2.9 in the following section.

⁸Please note that names of parameters may not be displayed by default in EA

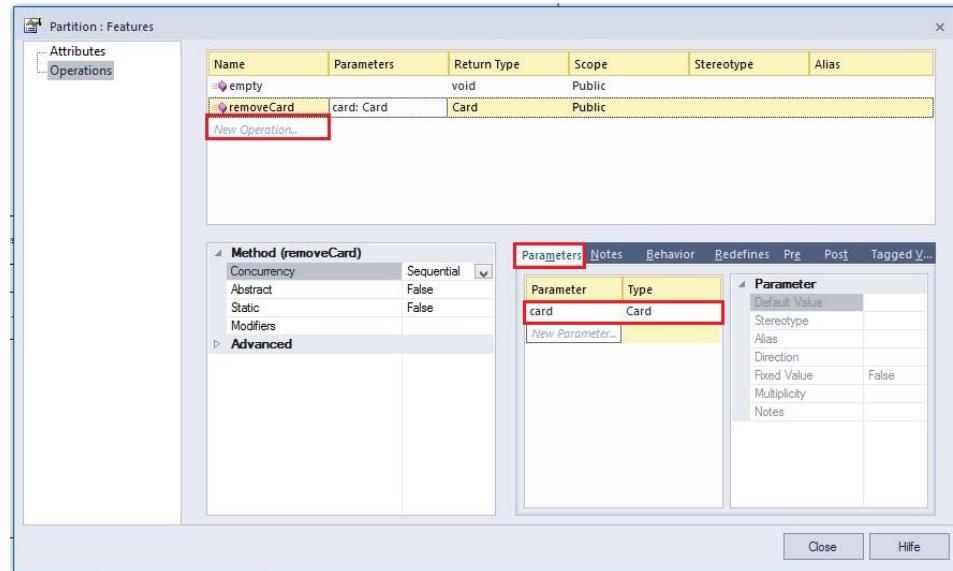


Figure 2.33: Parameters and return type

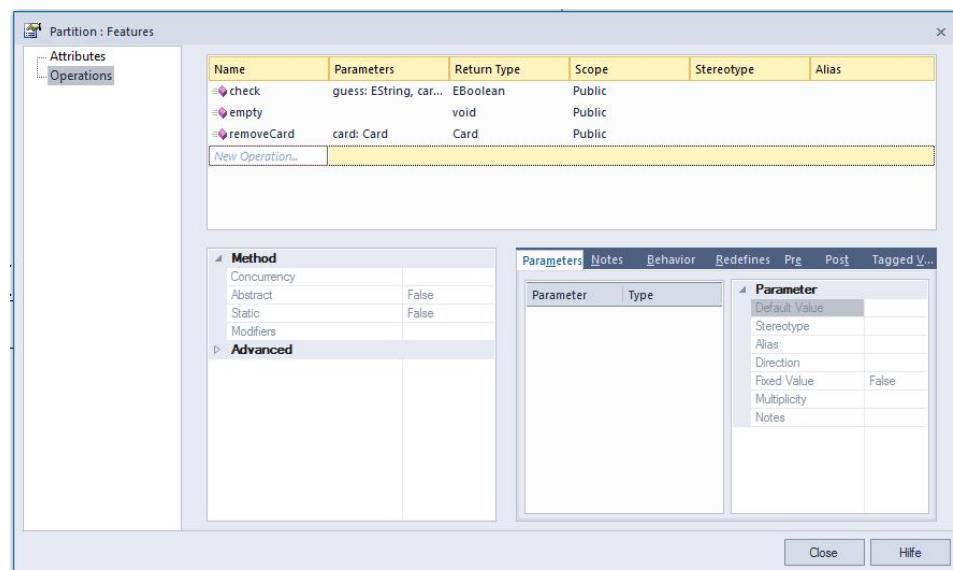


Figure 2.34: All operations in Partition

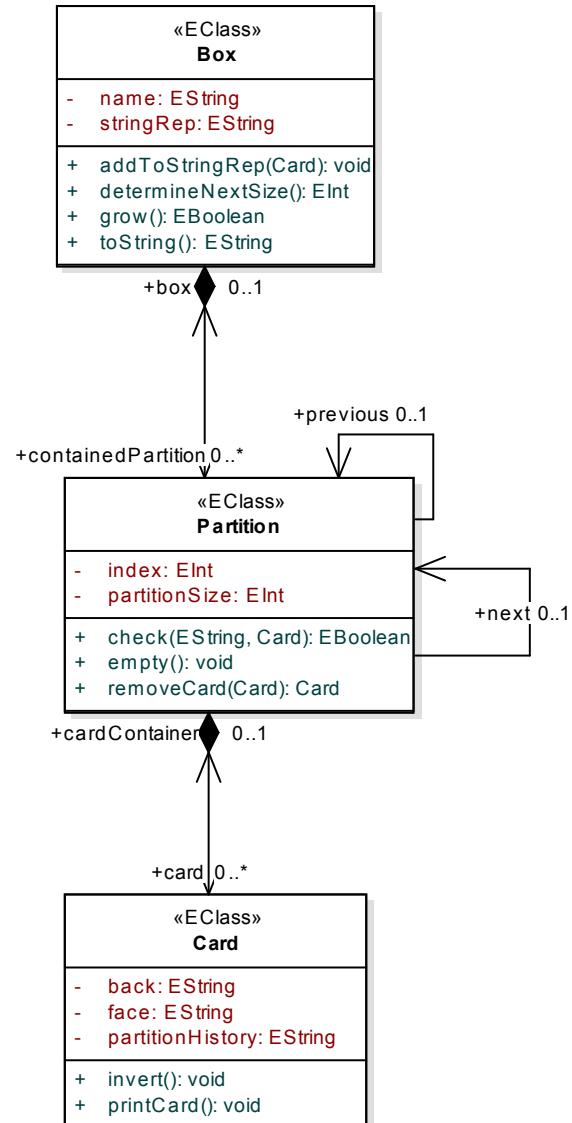


Figure 2.35: Complete metamodel for our learning box

▷ [Next](#)

2.7 Method Signatures

- We're nearing the end of our model creation! One of the last things we need to do is to make the model *do* something. After all, a model that only stores attributes and references is a bit boring, right?
- Let's set up the operations we want each EClass to do by declaring their *signatures* which follow the syntax below:

```
name '(' argument* ')' ':' return_type
```

With:

```
argument := param_name : param_type
name, return_type, param_name, param_type := STRING
```

- Starting with the **Partition** EClass, we want a partition to be able to do three things: compare the answer on a **Card** with a guess and return a true/false response, remove a specific card from the partition, or empty itself of all cards.
 - Start with the **empty** method. It won't need any parameters, and it doesn't return anything. Declare this via:
- ```
empty() : void
```
- Create two more functions for **Partition** the same way. We'll need a **removeCard** method that accepts and returns a **Card**, as well as a EBoolean **check** method that accepts a **Card** and an **EString** **guess**.
  - Your **Partition** EClass should now resemble Listing 2.8.

```
1 class Partition
2 {
3 index : EInt
4 partitionSize : EInt
5
6 <> - card (0..*) -> Card
7 - box (0..1) -> Box
8 - next (0..1) -> Partition
9 - previous (0..1) -> Partition
10
11 empty() : void
12 removeCard(card : Card) : Card
13 check(card : Card, guess : EString) : EBoolean
14 }
```

Listing 2.8: The completed **Partition** EClass

- What needs to be done in the **Card** EClass? Well, in order to check the card, we'll need to be able to look at the flip side. We'll also need

to print whatever is on the current side. Create two parameter-less `void` functions, `invert` and `printCard` (Listing 2.9).

```

1 class Card
2 {
3 back : EString
4 face : EString
5 partitionHistory : EString
6
7 - cardContainer(0..1) -> Partition
8
9 invert() : void
10 printCard() : void
11 }
```

Listing 2.9: The completed `Card` EClass

- Finally, what do we want to do with `Box`? In summary, we want a `Box` to:

`determineNextSize():EInt` Calculate how large a new partition in the box should be (Line 8)  
`grow():EBoolean` Increase the box by adding a new partition (Line 9)  
`toString():EString` Produce a string representation of the box with all its contents (Line 10)  
`addToStringRep(card:Card):void` Update the current string representation to include `card` (Line 11)

```

1 class Box
2 {
3 name : EString
4 stringRep : EString
5
6 <> - containedPartition (0..*) -> Partition
7
8 determineNextSize() : EInt
9 grow() : EBoolean
10 toString() : EString
11 addToStringRep(card:Card) : void
12 }
```

Listing 2.10: The completed `Box` EClass

- Implement the above signatures, and your entire workspace should now resemble Listings 2.10, 2.8 and 2.9.
- Congratulations! You have now created a metamodel for our Learning Box using eMoflon's textual syntax! To see how this looks in the visual syntax, check out Fig. 2.35 from the previous section. As a final step, make sure you build the project and wait for the package explorer to refresh.

## 2.8 eMoflon validation support in EA

Our EA extension provides rudimentary support for validating your metamodel. Validation results are displayed and, in some cases, even “quick fixes” to automatically solve the problems are offered. In addition to reviewing your model, the validation option automatically exports the current model to your eclipse workspace if no problems were detected.

- ▶ If not already active, make the eMoflon control panel visible in EA by choosing “Extensions/Add-In Windows”. This should display a new output window, as depicted in Figure 2.36. Many users prefer this interface as it provides quick access to all of eMoflon’s features, as opposed to the drop down menu under “Extensions/MOFLON::Ecore Addin” which only offers limited functionality.

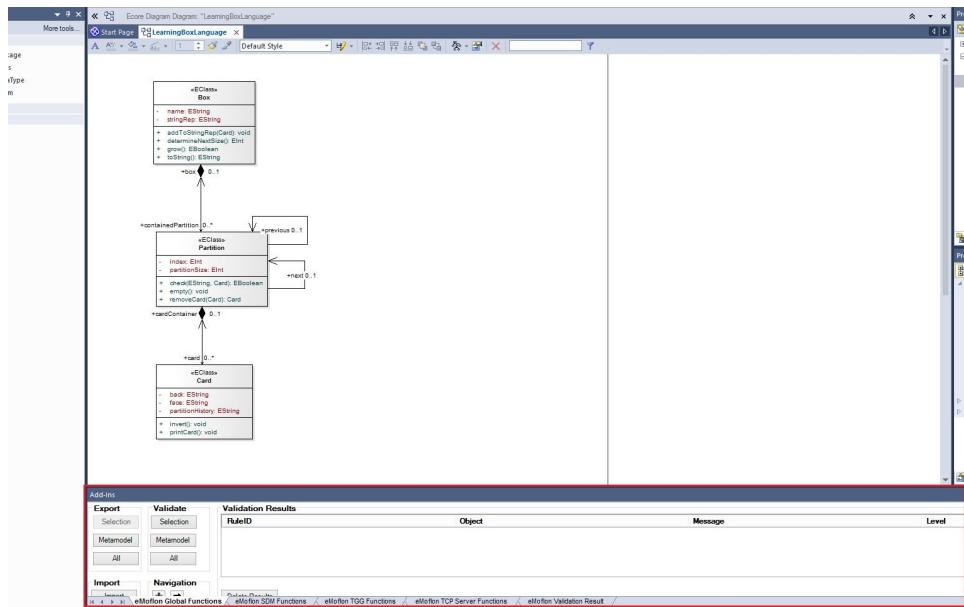


Figure 2.36: Activating the validation output window

- To start the validation, choose “Validate all” in the “Validate” section of the control panel (Fig. 2.37). If you haven’t made any mistakes while modelling your LearningBoxLanguage so far, the validation results window should remain empty, indicating your metamodels are free of errors.

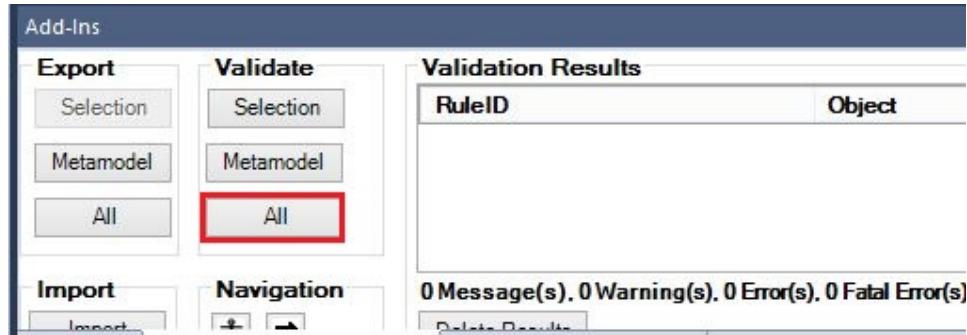


Figure 2.37: Starting the validation

If an error did appear, the validation system attempts to suggest a “Quick Fix.” Why don’t we examine the validation and quick fix features in detail? Let’s add two small modelling errors in LearningBoxLanguage.

- Create a new EClass in the LearningBoxLanguage diagram. You can retain the default name `EClass1`. Let’s assume you wish to delete this class from your metamodel.
- Select the rouge class in the diagram and press the `Delete` button on your keyboard. Note that this only deleted it from the current diagram and `EClass1` still exists in the project browser (and thus in your metamodel).
- Run the validation test, and notice the new `Information` message in the validation output (Fig. 2.38).

Figure 2.38: Validation information error: element still exists

This message informs you that `EClass1` is not on any diagram, and seeing as it is still in the metamodel, that this *could* be a mistake. As you can see, just pressing the `Delete` button is not the proper way of removing an EClass from a metamodel - It only removes it from the current diagram!<sup>9</sup>

---

<sup>9</sup>Deleting elements properly and other EA specific aspects are discussed in detail in Part VI: Miscellaneous

- Suppose you were inspecting a different diagram, and were not on the current screen. To navigate to the problematic element in the **Project Browser**, click *once* on the information message.

- ▶ To check to see if there are any quick fixes available, *double-click* the information message to invoke the “QuickFix” dialogue. In this case, there are two potential solutions - add the element to the current diagram or (properly) delete the element from the metamodel (Fig. 2.39). Since the latter was the original intent, click **Ok**.

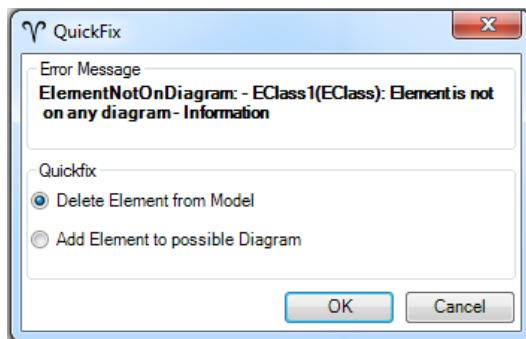


Figure 2.39: Quick fix for elements that are not on any diagram

- ▶ **EClass1** should now be correctly removed from your metamodel. Your metamodel should now be error-free again as indicated by the validation output window.
- ▶ To make an error that leads to a more critical message than “information,” double-click the navigable EReference end **previous** of the **EClass Partition**, and delete its role name as depicted in Fig. 2.40. Affirm with **OK**.
- ▶ You should now see a new **Fatal Error** in the validation output, stating that a navigable end *must* have a role name. Close all windows, then single click on the error once to open the relevant diagram and highlight the invalid element on the diagram. Double click the error to view the quick fix menu (Fig. 2.41). As navigable references are mapped to data members in a Java class, omitting the name of a navigable reference makes code generation impossible (data members ,i.e., class variables, must have a name).
- ▶ Given there are no automatic solutions, correct your metamodel manually by setting the name of the navigable EReference back to **previous**.
- ▶ Ensure that your metamodel closely resembles Figure 2.35 again, and that there are no error messages before proceeding.

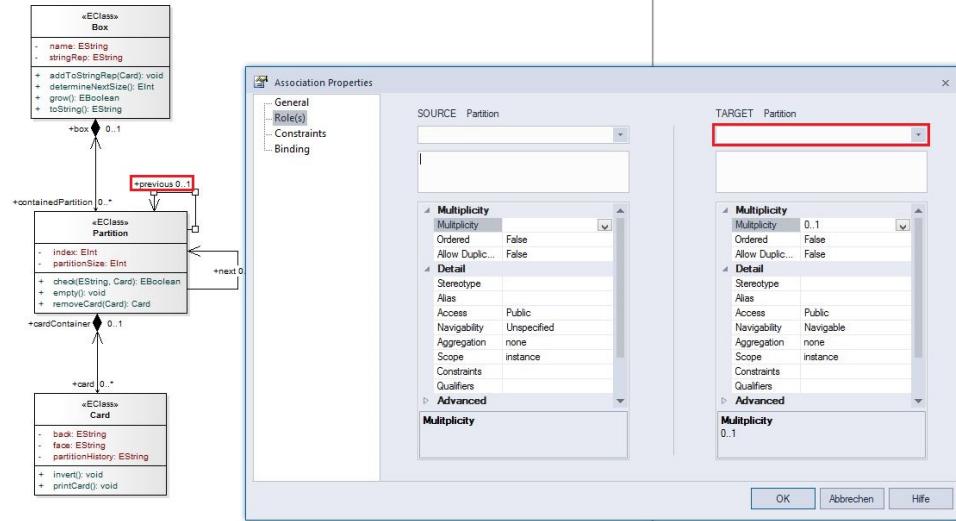


Figure 2.40: Deleting a navigable role name of an EReference

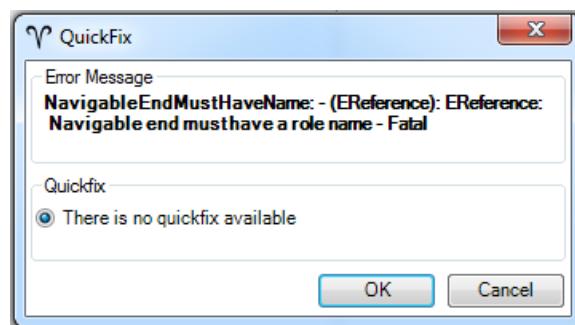


Figure 2.41: Fatal error after deleting a navigable role name

As you may have noticed, eMoflon distinguishes between five different types of validation messages:

**Information:**

This is only a hint for the user and can be safely ignored if you know what you're doing. Export and code generation should be possible, but certain naming/modelling conventions are violated, or a problematic situation has been detected.

**Warning:**

Export and code generation is possible, but only with defaults and automatic corrections applied by the code generator. As this might not be what the user wants, such cases are flagged as warnings (e.g., omitting the multiplicity at references which is automatically set by the code generator to 1). Being as explicit as possible is often better than relying on defaults.

**Error:**

Although the metamodel can be exported from EA, it is not Ecore conform, and code generation will not be possible.

**Fatal Error:**

The metamodel cannot be exported as required information, such as names or classifiers of model elements is incorrectly set or missing.

**Eclipse Error:**

Display error messages produced by our Eclipse plugin after an unsuccessful attempt to generate code.

## 2.9 eMoflon support with the MOSL Builder

Our MOSL language is accompanied with its own builder that proves support for validating your metamodel. Integrated with the Eclipse IDE, if there's an error in your files when you save, a message will appear in the console. Lets try to make an error, just to see how this works.

Go to your `Partition` EClass and change the parameter type in `removeCard` from `Card` to `card`. Press save. An error should immediately appear below the editor to inform you of your terrible mistake. Change your file back to the way it was, and the message should disappear.

In addition to validation on saving files, MOSL also lets you know whether or not your project has recently been built (if code has been generated from the metamodels). You may have noticed this feature in the previous sections. Similar to how Eclipse informs you that your file has changed by placing a `*` beside the name in the editor tab, MOSL places a `***` symbol beside your metamodel folder title in the package explorer to remind you to build your project.

## 2.10 Reviewing your metamodel

Before moving on, lets take a step back and review what we have accomplished. We have modelled a `Box` that can contain an arbitrary amount of `Partitions`. A `Partition` in the `Box` has a `next` and `previous Partition` that can be set. Finally, `Partitions` contain `Cards`.

A `Box` has a `name`, and can be extended by calling `grow`. A `Box` can print out its contents via the `toString` method.

The main method of the learning box is `Partition::check`, which takes a `Card` and the user's `EString` guess, and returns a `true` or `false` value.

A `Partition` can also `remove` a specific `Card`, or empty itself of all existing `Cards`. Last but not least, a `Partition` has a `partitionSize` to indicate how many cards it should hold. Too many cards in the first partition could indicate that not enough time has been dedicated to learning the terms. Too many near the end of the box could show that the vocabulary set is too easy, and probably already mastered.

A `Card` contains the actual content to be learned as a question on the card's `face` and the answer on the card's `back`. Cards also maintain a `partitionHistory`, which can be used to keep track of how often a `Card` has been answered incorrectly. This may indicate how difficult the `Card` is for a specific user, and remind them to spend more time on it. When learning a language, it makes sense to be able to swap the target and source language and this is supported by `Card` via `invert` (turns the card around).

Examine the generated files in "gen", especially the default implementation for those methods that currently just throw an `OperationNotSupportedException`. We shall see in later parts of this handbook how our code generator supports injecting hand-written implementation of methods into generated methods and classes. With eMoflon however, we can actually model a large part of the dynamic semantics with ease, and only need to implement small helper methods (such as those for string manipulation) by hand.

On a final note, we encourage you to review how the metamodel was constructed in the alternative syntax to the one you primarily used. Compare the differences between modelling classes and references visually using a separate program (Enterprise Architect), then exporting to Eclipse to build, versus modelling textually entirely within Eclipse. Which do you find easier to work with?

If you have had problems with this section, and, despite firmly believing everything is correct, things *still* don't work, feel free to contact us at: [contact@emoflon.org](mailto:contact@emoflon.org).

### 3 Creating instances

Before diving into modelling dynamic behaviour in Part III, let's have a brief look at how to create a concrete *instance* of your metamodel in Eclipse.

In the following section, we use *metamodel* and *instance* (model) to differentiate between models that represent the abstract syntax and static semantics of a domain specific language (metamodel), and those that are expressed *in* such a language (instance models of the metamodel).

- ▶ To create an instance model, navigate to the generated `model` folder in your `LearningBoxLanguage` project. Double-click the `LearningBoxLanguage.ecore` model to invoke the *Ecore model editor*.
- ▶ To create a concrete instance of the metamodel, you must select an EClass that will become the root element of the new instance. For our example, right-click `Box`, and navigate to “Create Dynamic Instance” from the context menu, as depicted in Fig. 3.1.
- ▶ A dialogue should appear asking where the instance model file should be persisted. Save your instances according to convention in a folder named “instances,” which is automatically created in every new repository project. Last but not least, enter `Box.xmi` as the name of the instance model (Fig. 3.2).
- ▶ Press **Finish**, and a generic model editor should open for your new instance model. This editor works just like the previous Ecore model editor except it's “generic,” meaning it allows you to create and edit an instance of *any* metamodel, not only instances of Ecore (i.e., metamodels).

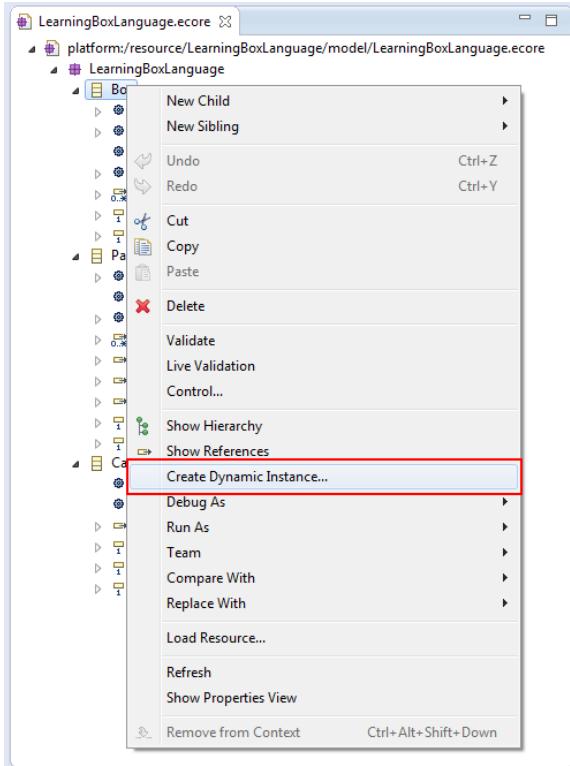


Figure 3.1: Context menu of an EClass in the Ecore editor

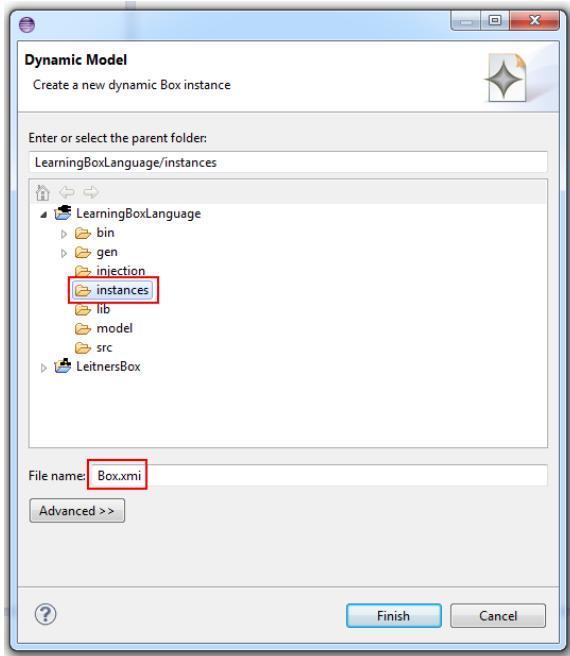


Figure 3.2: Dialogue for creating a dynamic model instance

- ▶ You can populate your instance by adding new children or siblings via a right-click of an element to invoke the context-menu depicted in Figure 3.3. Note that EMF supports you by respecting your metamodel, and reducing the choice of available elements to valid types only.<sup>10</sup>

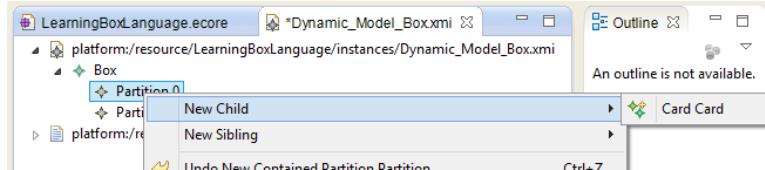


Figure 3.3: Context menu for creating model elements

- ▶ Let's try building a vocabulary set. Fill your box with three partitions, with two cards in each. Save your model by pressing **Ctrl+S** and confirm the save by closing it, then reloading via a simple double-click.
- ▶ Double-click on one of the partitions to bring up the “Properties” tab in the window below the editor (Fig. 3.4). Here you’ll see the attributes you defined earlier in each EClass. The **Box**, **Next**, and **Previous** values are its (undefined) EReferences,<sup>11</sup> while **Index** is a partition’s unique identification value, and **Partition Size** is the recommended number of cards the partition should contain before being tested.

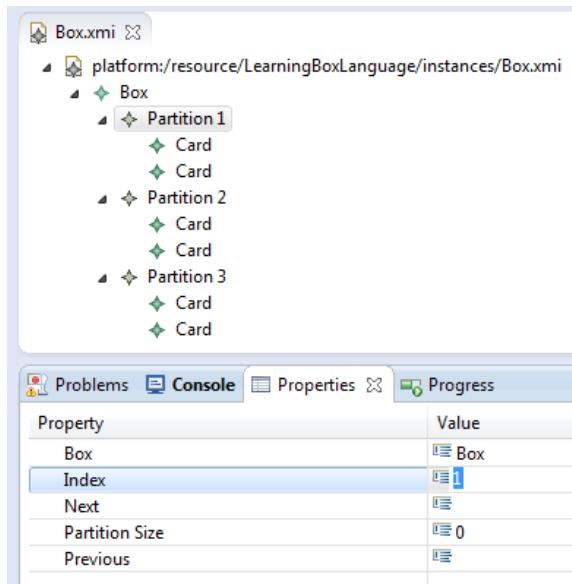


Figure 3.4: Change the **Index** value in the partition’s property tab

<sup>10</sup>This depends on the current context. Try it out!

<sup>11</sup>Please note that the editor tab capitalizes the first letter of each property

- ▶ Pick a number - any one you like - and update each partition's **Index** value. This is their identification value! You'll notice that as soon as you press **Enter**, the values will be reflected in the **.xmi** tree.
- ▶ Now you need to set the **Next** and **Previous** EReferences which will make it possible to move cards through the box. Given that there are no partition before the first, set **Previous** values only for your second and third partition to that first partition. Similarly, only set the **Next** values for the first, and second partition to their respective 'next' partitions.
- ▶ In the same fashion, double click on each **Card** you created and modify their values. In particular, update the **Back** attribute to **One**. This is the value you'll be able to see from the partition and thus, the **.xmi** tree. You'll be experimenting with the **Face** attribute shortly, so provide a **1** value for that as well.
- ▶ Fill in the rest the cards with similar vocabulary-style words (such as two/2, three/3, ...) that you can 'test' yourself on (Fig. 3.5). You now have a unique, customized learning box! Save your model, and ensure no errors exist before proceeding.

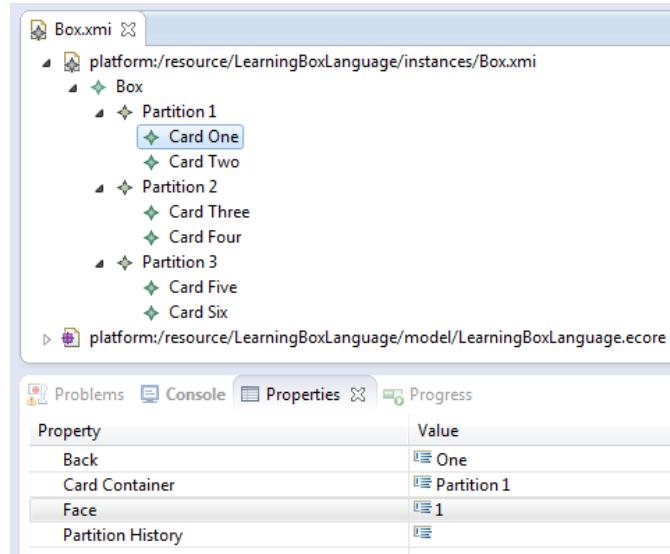


Figure 3.5: A vocabulary-style learning box

## 4 eMoflon's graph viewer

At this point, you should now have a small instance model. While Eclipse's built-in editor offers a nice tree structure and table to view and edit your model, wouldn't it be nice to have a visualisation? eMoflon offers a "Graph View" to do exactly that! You'll find that this is an effective feature to view how each element in your instance model interacts with others.

- ▶ When you first opened the eMoflon perspective, a small window to the right should have appeared with two tabs, "Outline" and "Graph View." <sup>12</sup>
- ▶ Activate the second tab and drag one of your **Partition** instances into the empty window.
- ▶ To the right you can see the partition that was dragged in, double click on it to visualise all referenced objects from **Partition**. (Fig. 4.1).

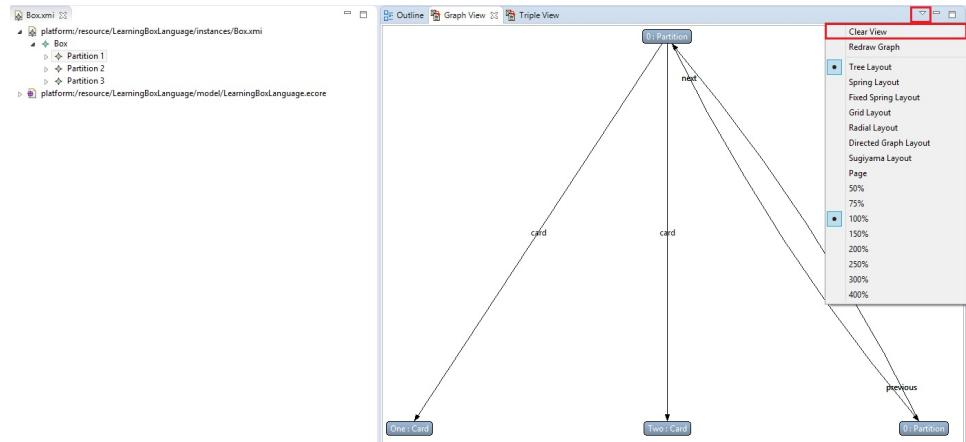


Figure 4.1: A single partition in eMoflon's graph view

- ▶ Clicking any item in the view will highlight it, and hovering over a node will list all its properties in a pop-up dialogue. If you hover over a card, for example, you should be able to see their **back** and **face** values. You can also reposition elements.

---

<sup>12</sup>If this window is not open, you can re-activate it by right-clicking on the "eMoflon" perspective in the main toolbar and pressing "Reset," or by going to "Window>Show View/Other..," then "Other/Graph View"

- ▶ Try dragging a second **Partition** element from the model to the viewer. As you can see, it doesn't replace the current view, it simply places them side-by-side, showing the references between them partitions. To clear the screen, click the upside-down triangle in the top right corner of the window, and select "Clear View" (Fig. 4.1).
- ▶ To make the graph bigger, increase the size of the window, then select "Redraw Graph" from the same upside-down arrow. As you can see, when an instance is already loaded, the graph view is not automatically updated. This option is useful if you change a property of an element (ie., the **back** value of a **Card**) and wish to see it reflected in the graph.
- ▶ Try experimenting with each of the different instance elements, viewer layouts and zoom settings found under the arrow. You'll notice for the "Spring Layout" that each time you press "Redraw Graph," the graph will re-arrange itself, even if you haven't updated any values or changed the window size.
- ▶ The "Graph View" features is not exclusively for instance models; Expand the second root node in the editor, and drag and drop the **LearningBoxLanguage** package into the viewer. Click on some or all nodes to open your entire metamodel completely. Now it is displayed in the viewer (Fig. 4.2). We have found that the "Radial Layout" works best for a view of this complexity. This might be confusing at first, but the view contains your metamodel displayed in its abstract syntax, i.e., as an instance of Ecore. In contrast, the tree view displays the metamodel in UML-like concrete syntax.

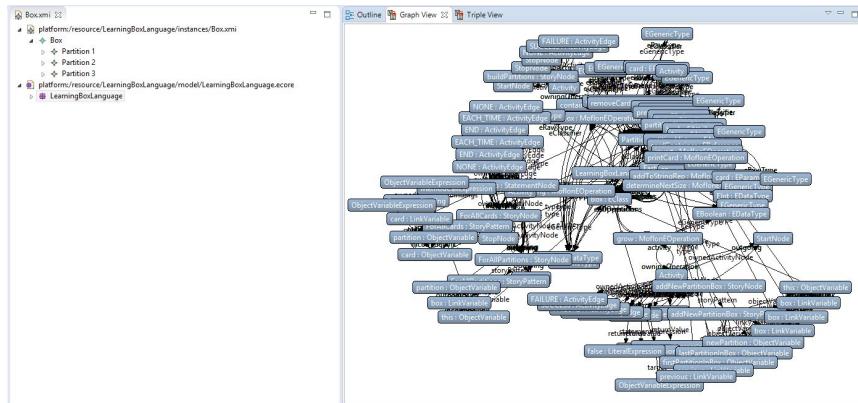


Figure 4.2: Our instance's complete type graph

## 5 Introduction to injections

This short introduction will show you how to implement small methods by adding handwritten code to classes generated from your model. Injections are inspired by partial classes in C#, and are our preferred way of providing a clean separation between generated and handwritten code.

Let's implement the `removeCard` method, declared in the `Partition` EClass. In order to 'remove' a card from a partition, all one needs to do is disable the link between them. Don't forget that (according to the signature) not only does `removeCard` have to pass in a `Card`, it must return one as well.

- ▶ From your working set, open “gen/LearningBoxLanguage.impl/PartitionImpl.java” and enter the following code in the `removeCard` declaration, starting at approximately line 347. Do not remove the first comment, which is necessary to indicate that this code is written by the user and needs to be extracted automatically as an injection. Please also do not copy and paste the following code – the copying process will most likely add invisible characters that eMoflon is unable to handle.

```
public Card removeCard(Card toBeRemovedCard) {
 // [user code injected with eMoflon]
 if(toBeRemovedCard != null){
 toBeRemovedCard.setCardContainer(null);
 }
 return toBeRemovedCard;
}
```

Figure 5.1: Implementation of `removeCard`

- ▶ Save the file, then right-click either on the file in the package explorer, or in the editor window, and choose “eMoflon/ Create/Update Injection for class” (Alt+Shift+E,I) from the context menu (Fig. 5.2).
- ▶ This will create a new file in the “injection” folder of your project with the same package and name structure as the Java class, but with a new `.inject` extension (Fig. 5.3).
- ▶ Double click to open and view this file. It contains the definition of a *partial class* (Fig. 5.4).

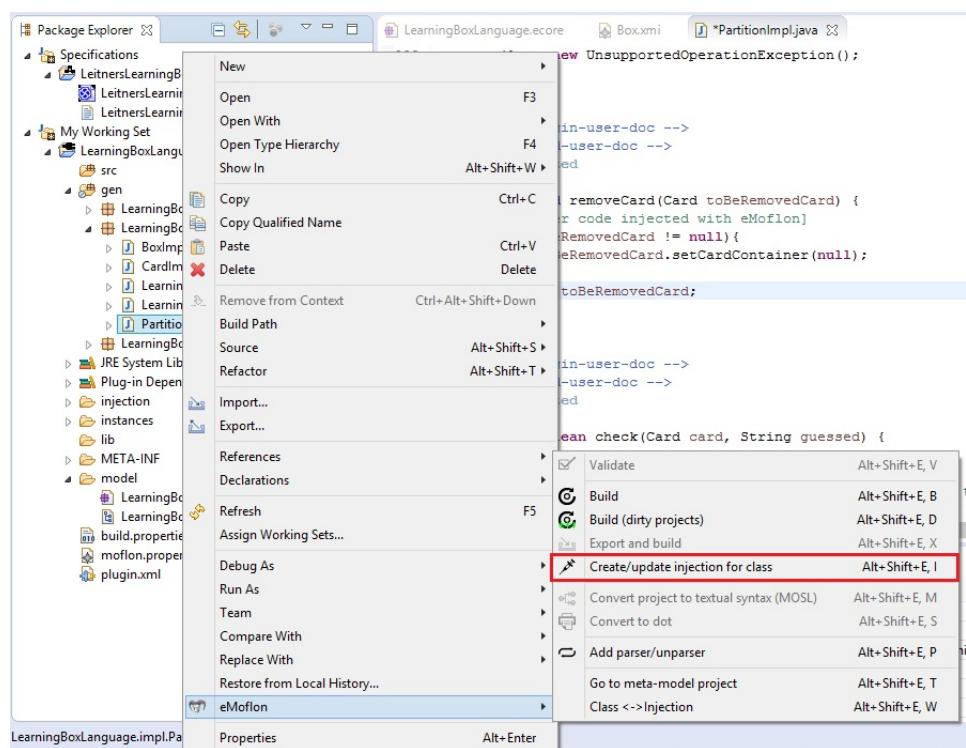


Figure 5.2: Create a new injection

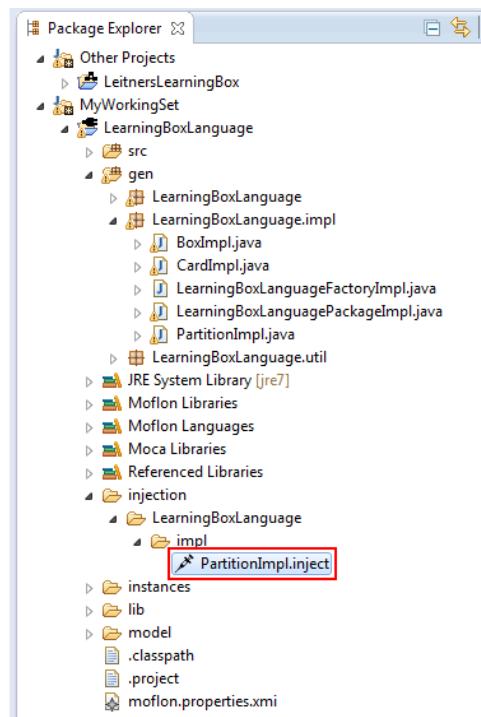


Figure 5.3: Partition injection file

```

1
2 partial class PartitionImpl {
3
4
5 @model removeCard (Card toBeRemovedCard) <-->
6
7 if(toBeRemovedCard != null){
8 toBeRemovedCard.setCardContainer(null);
9 }
10 return toBeRemovedCard;
11
12 -->
13
14 }
```

Figure 5.4: Generated injection file for PartitionImpl.java

- ▶ As a final step, build your metamodel to check that the code is generated and injected properly.
- ▶ By the way, eMoflon allows you to switch quickly between a Java class and its injection file. When inside “PartitionImpl.java”, open the context menu and select “eMoflon/Class <-> Injection” (Alt+Shift+E,W). This brings you to “PartitionImpl.java”.  
Repeat this command and you are back in “PartitionImpl.inject“.
- ▶ That’s it! While injecting handwritten code is a remarkably simple process, it is pretty boring and low level to call all those setters and getters yourself. We’ll return to injections for establishing two simple methods in Part III using this strategy, but we’ll also learn how to implement more complex methods using Story Diagrams.

### Creating injections automatically with save actions

Information loss is a typical pitfall that you may encounter when working with injections: If you forget to create an injection and trigger a rebuild (“eMoflon/Build”, Alt+Shift+E,B), the generated code is entirely dropped – including any unsaved injection code.

To save you from this frustrating experience, eMoflon may automatically save injections whenever you save your Java file.

- ▶ Open the preferences dialog via “Window/Preferences” and navigate to the “Save Actions” (“Java/Editor/Save Actions”).
- ▶ To enable save actions, tick “Perform the selected actions on save” and “Additional actions” (Fig. 5.5).
- ▶ Press “Configure”, switch to the “eMoflon Injections” tab and tick “Enable injection extraction on save” (Fig. 5.6).
- ▶ After confirming the dialog, the bulleted list should contain the entry “Create eMoflon injections”.
- ▶ Open up “PartitionImpl.java”, perform a tiny modification on it, and save the file. The eMoflon console should display a message to confirm that injections have been extracted automatically, e.g.,

```
[handlers.CreateInjectionHandler::115] - Created injection
file for 'PartitionImpl'.
```

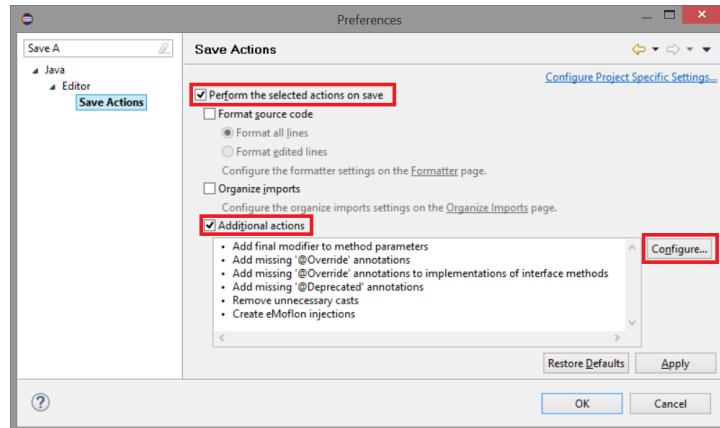


Figure 5.5: Main configuration dialog for save actions

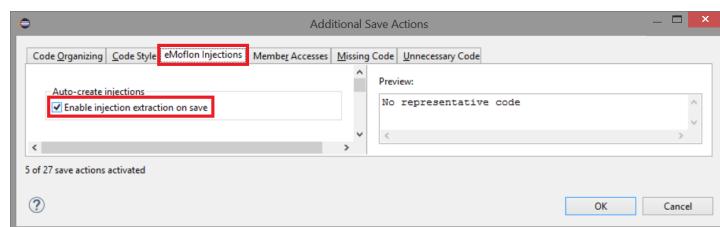


Figure 5.6: Configuration dialog for additional save actions

## 6 Leitner's Box GUI

We would like to now provide you with a simple GUI with which you can take your model for a spin and see it in action.

- ▶ Navigate to the top left of your toolbar and start the “New” wizard.
- ▶ Load “Examples/eMoflon Handbook Examples/Part II GUI Download” (Fig. 6.1). This will load the new project into your workspace.<sup>13</sup> Right click **LeitnersBox** to invoke the context menu and select “Run as/Java application.”

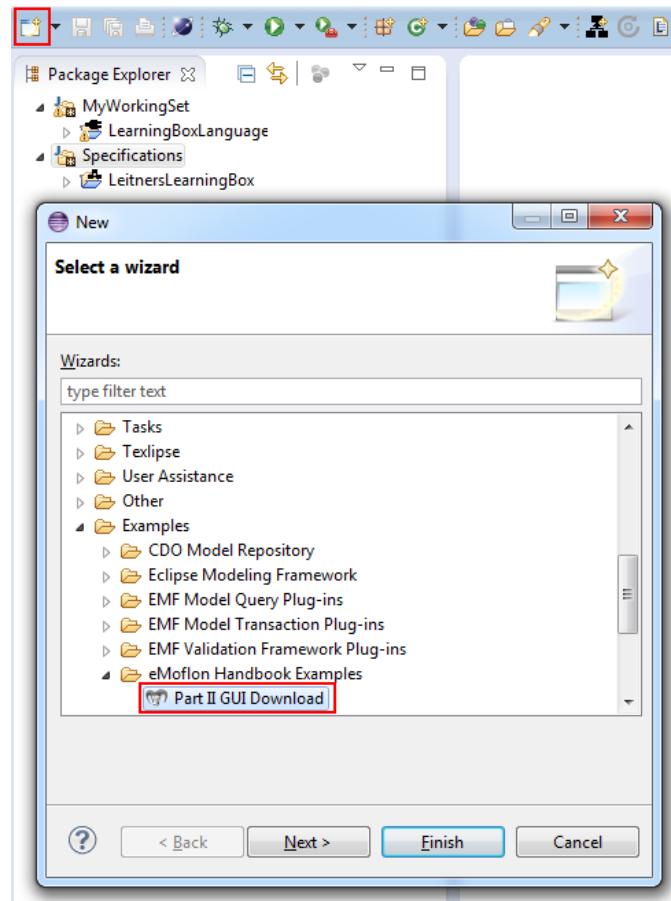


Figure 6.1: Load Leitner's Box GUI

---

<sup>13</sup>If nothing appears, go to the small arrow to the right of the window, and select “Configure Working Sets...” Make sure **Other Projects** is selected, and press **Ok**.

- ▶ The GUI will automatically navigate to the instances folder where you've stored your instance model, then load your partitions and cards into the visualized box. Please note that this will only work if you named your dynamic instance `Box.xmi` and placed it in the `instances` folder as suggested.
- ▶ Navigate to any card, and you'll be able to see two options, "Remove Card" and "Check Card" (Fig 6.2). While we'll implement "Check Card" in Part III, "Remove Card" is currently active, implemented by the injection you just created.

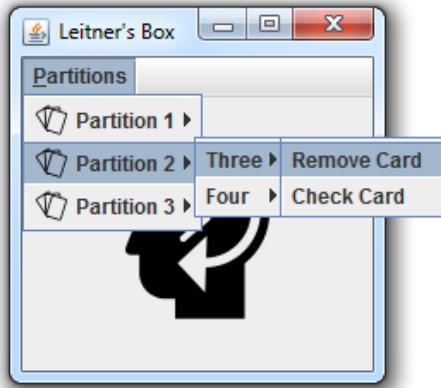


Figure 6.2: Using the GUI with your instance

- ▶ Experiment with your instance model and confirm your injection works by removing some items from the GUI. You'll notice the change immediately in the `Box.xmi` file. You can also close the GUI and add, remove, or rename more elements in the model, then observe how the changes are reflected in the GUI.
- ▶ Expand the "Other Projects" node and explore `LeitnersBoxController` and `LeitnersBoxView` files to get an understanding of the GUI. Can you find where the controller loads and connects to the model?

## 7 Conclusion and next steps

Whoo, this has been quite a busy handbook. Great job, you've finished Part II! This part contained some key skills for eMoflon, as we learned how to create the abstract syntax in Ecore for our learning box! Both the visual and textual specifications are now complete with all the classes, attributes, references, and method signatures that make up the type graph for a working learning box. Additionally, we also learned how to insert a small handwritten method into generated code, and tested all our work in an interactive GUI.

If you enjoyed this section and wish to fully develop *all* the methods we just declared, we invite you to carry on to Part III: Story Diagrams<sup>14</sup>! Story Diagrams are a powerful feature of eMoflon as we can model a large part of a system's dynamic semantics via high-level pattern rules.

Of course, you're always free to pick a different part of the handbook if you feel like skipping ahead and checking out some of the other features eMoflon has to offer. Check out Triple Graph Grammars (TGGs) in Part IV<sup>15</sup>, or Model-to-Text Transformations in Part V<sup>16</sup>. We'll provide instructions on how to easily download all the required resources so you can start without having to complete the previous parts.

For a detailed description of all parts, please refer to Part 0<sup>17</sup>.

Cheers!

---

<sup>14</sup>Download: <http://tiny.cc/emoflon-rel-handbook/part3.pdf>

<sup>15</sup>Download: <http://tiny.cc/emoflon-rel-handbook/part4.pdf>

<sup>16</sup>Download: <http://tiny.cc/emoflon-rel-handbook/part5.pdf>

<sup>17</sup>Download: <http://tiny.cc/emoflon-rel-handbook/part0.pdf>

# Glossary

**Abstract Syntax** Defines the valid static structure of members of a language.

**Concrete Syntax** How members of a language are represented. This is often done textually or visually.

**Constraint Language** Typically used to specify complex constraints (as part of the static semantics of a language) that cannot be expressed in a metamodel.

**Dynamic Semantics** Defines the dynamic behaviour for members of a language.

**Grammar** A set of rules that can be used to generate a language.

**Graph Grammar** A grammar that describes a graph language. This can be used instead of a metamodel or type graph to define the abstract syntax of a language.

**Meta-Language** A language that can be used to define another language.

**Meta-metamodel** A *modeling language* for specifying metamodels.

**Metamodel** Defines the abstract syntax of a language including some aspects of the static semantics such as multiplicities.

**Model** Graphs which conform to some metamodel.

**Modelling Language** Used to specify languages. Typically contains concepts such as classes and connections between classes.

**Static Semantics** Constraints members of a language must obey in addition to being conform to the abstract syntax of the language.

**Type Graph** The graph that defines all types and relations that form a language. Equivalent to a metamodel but without any static semantics.

**Unification** An extension of the object oriented “Everything is an object” principle, where everything is regarded as a model, even the metamodel which defines other models.

# An Introduction to Metamodelling and Graph Transformations

---

*with eMoflon*



---

## Part III: Story Driven Modelling

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

For further information contact us at [contact@emoflon.org](mailto:contact@emoflon.org).

*The eMoflon team*  
Darmstadt, Germany (August 2015)

# Contents

|    |                                                       |     |
|----|-------------------------------------------------------|-----|
| 1  | Leitner's learning box reviewed . . . . .             | 2   |
| 2  | Transformations explained . . . . .                   | 4   |
| 3  | Removing a card . . . . .                             | 8   |
| 4  | Checking a card . . . . .                             | 30  |
| 5  | Running the Leitner's Box GUI . . . . .               | 51  |
| 6  | Emptying a partition of all cards . . . . .           | 52  |
| 7  | Inverting a card . . . . .                            | 58  |
| 8  | Growing the box . . . . .                             | 65  |
| 9  | Conditional branching . . . . .                       | 75  |
| 10 | A string representation of our learning box . . . . . | 82  |
| 11 | Fast cards! . . . . .                                 | 90  |
| 12 | Reviewing eMoflon's expressions . . . . .             | 100 |
| 13 | Complex Attribute Manipulation . . . . .              | 102 |
| 14 | Conclusion and next steps . . . . .                   | 112 |
|    | Glossary . . . . .                                    | 113 |

## Part III:

# Story Driven Modelling

Approximate time to complete: 3h

URL of this document: <http://tiny.cc/emoflon-rel-handbook/part3.pdf>

Welcome to Part III, an introduction to unidirectional model transformations with programmed graph transformations via Story Driven Modelling (SDM). SDMs are used to describe behaviour, so the plan is to implement the methods declared in Part II with story diagrams. In other words, this is where you'll complete your metamodel's dynamic semantics! Don't let the size of this part frighten you off. We have included thorough explanations (with an ample number of figures) to ensure the concepts are clear.

In Part II, we learnt that we can implement methods in a fairly straightforward manner with injections and Java, so why bother with SDMs?

Overall, SDMs are a simpler, pattern-based way of specifying behavior. Rather than writing verbose Java code yourself, you can model each method and generate the corresponding code. With the visual syntax, you'll be using familiar, easy-to-understand UML activity and object diagrams to establish your methods. Textually, SDMs employ a simple Java-like syntax for imperative control flow, and a declarative pattern language with a syntax similar to list pattern matching constructs common in many functional programming languages

If you're just joining us, read the next section for a brief overview of our running example so far, and how to download some files that will help you get started right away. Alternatively, if you've just completed Part II, click the link below to continue right away with your constructed learning box metamodel. Please note that the handbook has been tested only with the prepared cheat packages provided in eMoflon.

▷ Continue from Part II...

## 1 Leitner's learning box reviewed

*Leitner's learning box*<sup>1</sup> is a simple, but ingenious little contraption to support the tedious process of memorization, especially prominent when trying to learn, for example, a new language. As depicted in Fig. 1.1, a box consists of a series of partitions with a strict set of rules. The contents to be memorized are written on little cards and placed in the first container. Every time the user correctly answers a card, that card is promoted to the next partition. Once it reaches the final partition, it can be considered memorized, and no longer needs to be practiced. Every time the user incorrectly answers a card however, it is returned to the original starting partition, and the learning process is restarted.

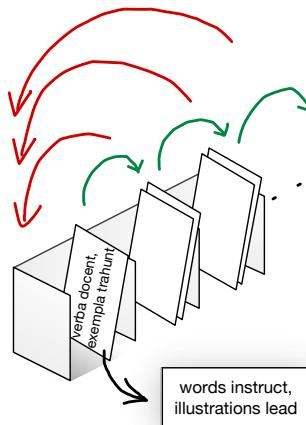


Figure 1.1: Static Structure of a Leitner's Learning Box

For a more detailed overview of the box and our goals, we recommend you read the introduction to Part II. But for now, enough discussion!

- ▶ To get started in Eclipse, press the **New** button and navigate to “Examples/eMoflon Handbook Examples/” (Fig. 1.2).
- ▶ Choose the cheat package for Part III and the syntax you prefer (textual or visual). Refer to Section 1 from Part I for details on the differences between our syntaxes. The cheat package contains all files created up to the example in this point, as well as a small GUI that will enable you to experiment with your metamodel.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Leitner\\_system](http://en.wikipedia.org/wiki/Leitner_system)

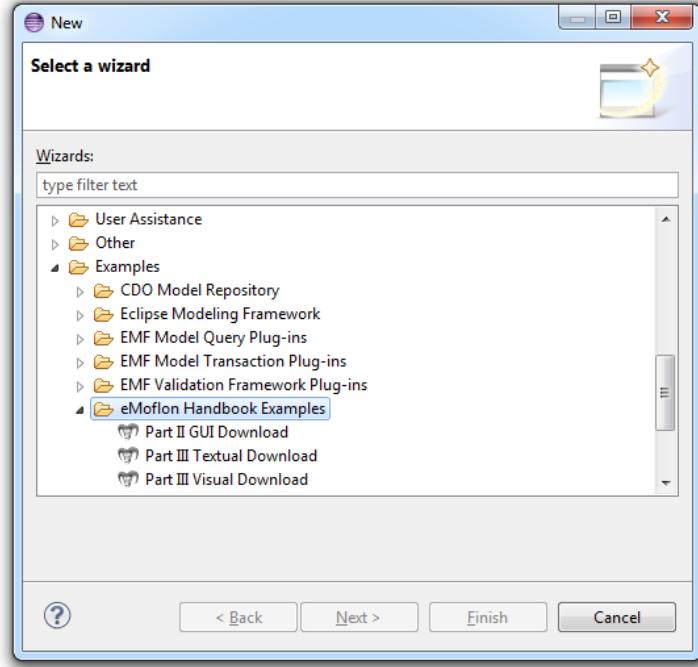


Figure 1.2: Choose a cheat package

- ▶ In order to start working with the cheat package, you have to generate code by (i) opening the .eap file in `LeitnersLearningBox`, (ii) exporting it using Enterprise Architect, (iii) refreshing the project containing the .eap and (iv) rebuilding the `LearningBoxLanguage` project. For more details on the code generation process, refer to Part I, Section 2.
- ▶ Inspect the files in both projects until you feel comfortable with what you'll be working with. In particular, look at the files found under “gen.” Each Java file has a corresponding .impl file, where all generated method implementations will be placed.
- ▶ Be sure to also review the Ecore model in “LearningBoxLanguage/-model/” and the dynamic model found in “instances.” While you can make and customize your own instances,<sup>2</sup> we have included a small sample to help you get started.

Well, that's it! A quick review, paired with a fine cheat package makes an excellent appetizer to SDMs. Let's get started.

---

<sup>2</sup>To learn how to make your own instance models, review Part II, Section 4

## 2 Transformations explained

The core idea when modeling behaviour is to regard dynamic aspects of a system (let's call this a model from now on) as bringing about a change of state. This means a model in state  $S$  can evolve to state  $S^*$  via a transformation  $\Delta : S \xrightarrow{\Delta} S^*$ . In this light, dynamic or behavioural aspects of a model are synonymous with *model transformations*, and the dynamic semantics of a language equates simply to a suitable set of model transformations. This approach is once again quite similar to the object oriented (OO) paradigm, where objects have a state, and can *do* things via methods that manipulate their state.

So how do we *model* model transformations? There are quite a few possibilities. We could employ a suitably concise imperative programming language in which we simply say how the system morphs in a step-by-step manner. There actually exist quite a few very successful languages and tools in this direction. But isn't this almost like just programming directly in Java? There's got to be a better way!

From the relatively mature field of graph grammars and graph transformations, we take a *declarative* and *rule-based* approach. Declarative in this context means that we do not want to specify exactly how, and in what order, changes to the model must be carried out to achieve a transformation. We just want to say under what conditions the transformation can be executed (precondition), and the state of the model after executing the transformation (postcondition). The actual task of going from precondition to postcondition should be taken over by a transformation engine, where all related details are basically regarded as a black box.

So, inspired by string grammars and this new, refined idea of a model transformation (which is of the form  $(pre, post)$ ), let's call this black box transformation a *rule*. It follows that the precondition is the left-hand side of the rule,  $L$ , and the postcondition is the right-hand side,  $R$ .

A rule,  $r : (L, R)$ , can be *applied* to a model (a typed graph)  $G$  by:

1. *Finding* an occurrence of the precondition  $L$  in  $G$  via a *match*,  $m$
2. *Cutting out* or *Destroying*  $(L \setminus R)$ , i.e., the elements that are present in the precondition but not in the postcondition are deleted from  $G$  to form  $(G \setminus Destroy)$
3. *Pasting* or *Creating*  $(R \setminus L)$ , i.e., new elements that are present in the postcondition but not in the precondition and are to be created in the hole left in  $(G \setminus Destroy)$  to form a new graph,  $H = (G \setminus Destroy) \cup Create$  (Fig.2.1).

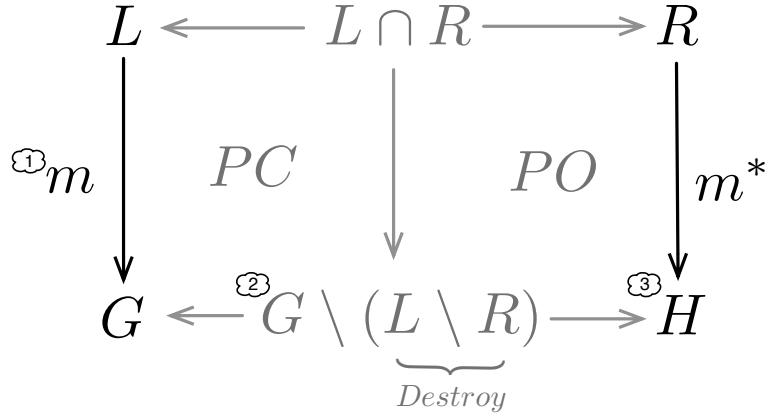


Figure 2.1: Applying a rule  $r : (L, R)$  to  $G$  to yield  $H$

Let's review this application.

- (1) is determined by a process called *graph pattern matching* i.e., finding an *Pattern Matching* occurrence or *match* of the precondition or *pattern* in the model  $G$ .
- (2) is determined by building a *push-out complement*  $PC = (G \setminus \text{Destroy})$ , such that  $L \cup PC = G$ .
- (3) is determined by building a *push-out*  $PO = H$ , so that  $(G \setminus \text{Destroy}) \cup R = H$ .

A push-out (complement) is a generalised union (subtraction) defined on typed graphs. Since we are dealing with graphs, it is not such a trivial task to define (1) – (3) in precise terms, with conditions for when a rule can or cannot be applied. A substantial amount of theory already exists to satisfy this goal.

Since this black box formalisation involves two push-outs - one when cutting  $\text{Destroy} := (L \setminus R)$  from  $G$  to yield  $(G \setminus \text{Destroy})$  (deletion), and one when inserting  $\text{Create} := (R \setminus L)$  in  $(G \setminus \text{Destroy})$  to yield  $H$  (creation) - this construction is referred to as a *double push-out*. We won't go into further details in this handbook, but the interested reader can refer to [?] for the exciting details.

Now that we know what rules are, let's take a look at a simple example for our learning box. What would a rule application look like for moving a card from one partition to the next? Figure 2.2 depicts this *moveCard* rule.

As already indicated by the colours used for *moveCard*, we employ a compact representation of rules formed by merging  $(L, R)$  into a single *story*

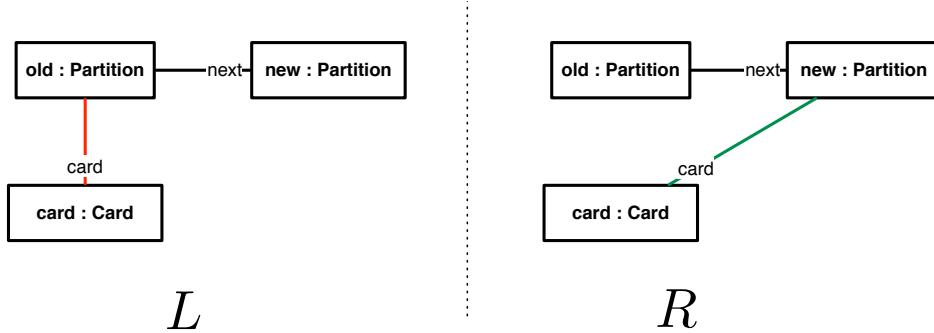


Figure 2.2: *moveCard* as a graph transformation rule

*pattern* composed of *Destroy* :=  $(L \setminus R)$  in red, *Retain* :=  $L \cap R$  in black, *Story Pattern* and *Create* :=  $(R \setminus L)$  in green (Fig. 2.3).

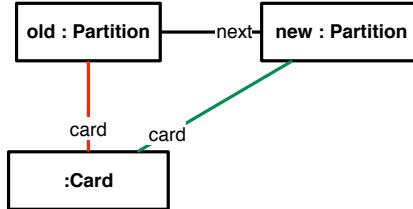


Figure 2.3: Compact representation of *moveCard* as a single *story pattern*

As we shall see in a moment, this representation is quite intuitive, as one can just forget the details of rule application and think in terms of what is to be deleted, retained, and created. We can therefore apply *moveCard* to a learning box in terms of steps (1) – (3), as depicted in Fig. 2.4.

Despite being able to merge rules together to form one story pattern, the individual rules still have to be applied in a suitable sequence to realise complex model transformations consisting of many steps! This can be specified with simplified activity diagrams, where every *activity node* or *story node* contains a single *story pattern*, and are combined with the usual imperative constructs to form a control flow structure. The entire transformation can therefore be viewed as two separate layers: an imperative layer to define the top-level control flow via activities (i.e., if/else statements, loops, etc.), and a pattern layer where each story pattern specifies (via a graph transformation rule) how the model is to be manipulated.

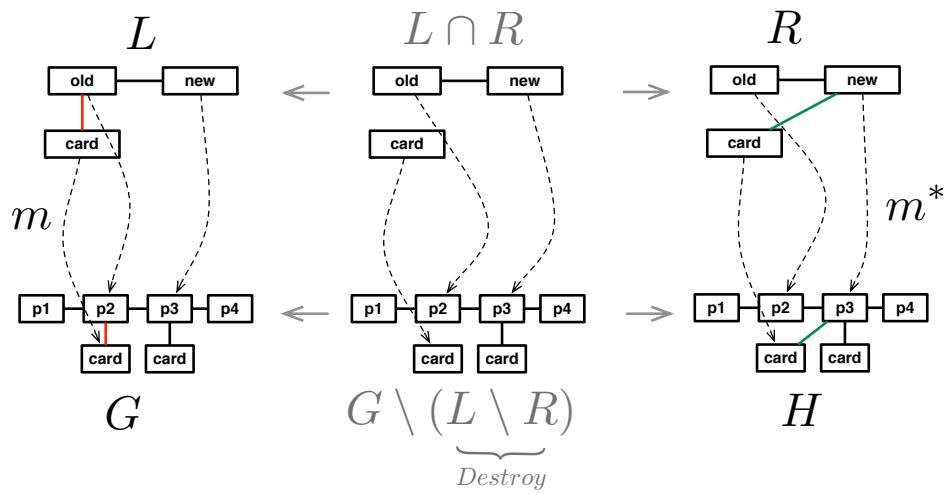


Figure 2.4: Applying *moveCard* to a learning box

Enough theory! Grab your mouse and let's get cracking with SDMs...

### 3 Removing a card

Since we're just getting started with SDMs,<sup>3</sup> let's re-implement the method previously specified directly in Java as an injection.<sup>4</sup> The goal of this method is to remove a single card from its current partition, which can be done by destroying the link between the two items (Fig. 3.1).

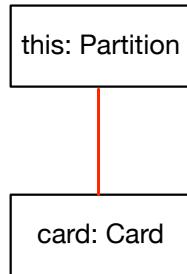


Figure 3.1: Removing a card from its partition

According to the signature of the method `removeCard`, we should return the card that has been deleted. Although this might strike you as slightly odd, considering that we already passed in the card as an argument, it still makes sense as it allows for chaining method calls:

```
aPartition.removeCard(aCard).invert()
```

Before we implement this change as a story diagram, let's remove the old injection content to avoid potential conflicts.

- ▶ Delete the `PartitionImpl.inject` file from your working set (Fig 3.2).
- ▶ Now right-click on `LearningBoxLanguage` and go to “eMoflon/Build”
- ▶ You'll be able to see the changes in `PartitionImpl.java`. The `removeCard` declaration should now be empty and look identical to the other unimplemented methods.

---

<sup>3</sup>As you may have already noticed, we use “SDM” or “Story Diagrams” interchangeably to mean both our graph transformation language *or* a concrete transformation used to implement a method, consisting of an activity with activity nodes containing story patterns.

<sup>4</sup>Refer to Part II, Section 6

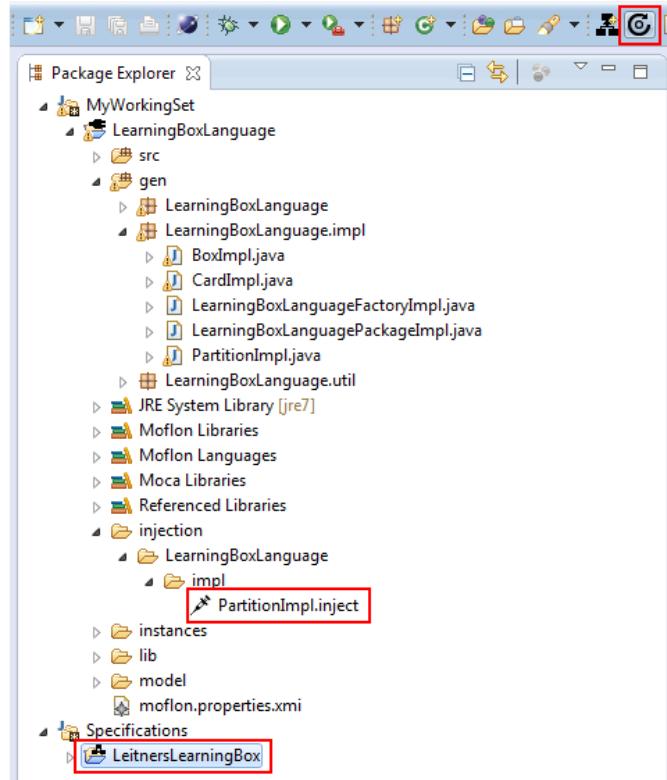


Figure 3.2: Remove injection content

That's it! We now have a fresh start for `removeCard`. Let's briefly discuss what we need to establish the transformation.

One of the goals of SDM is to allow you to focus less on *how* a method will do something, but rather on *what* the method will do. Integrated as an atomic step in the overall control flow, a single graph transformation step (such as link deletion) can be embedded as a *story pattern*.

These patterns declare *object variables*, place holders for actual objects in a model. During *pattern matching*, objects in the current model are assigned to the object variables in the pattern according to the indicated type and other conditions.<sup>5</sup>

*Object Variable*

---

<sup>5</sup>We shall learn what further conditions may be specified in later SDMs.

In `removeCard`, the SDM requires just two object variables: a `this` partition (named according to Java convention) referring to the object whose method is invoked, and `card`, the parameter that will be removed.

Patterns also declare *link variables* to match references in the model. Given *Link Variable* that we're concerned with removing a certain card from a specific partition, `removeCard` will therefore have a single link variable that connects these two objects together.

In general, pattern matching is non-deterministic, i.e., variables in the pattern are bound to *any* objects that happen to match. How can this be influenced so that, as required for `removeCard`, the pattern matcher chooses the correct `card` (that which is passed in as a parameter)?

The *binding state* of an object variable determines how it is found. By *Binding State* default, every object variable is *unbound*, or a *free variable*. Values for *Free Variable* these variables can be determined automatically by the pattern matcher. By declaring an object variable that is to be *bound* however, it will have *Bound* a fixed value determined from previous activity nodes. The appropriate binding is implicitly determined via the *name* of the bound object variable. As a rule, `this` variables, and any method parameters (i.e., `card`) are always bound.

On a final note, every object or link variable can also set its *binding operator* to `Check Only`, `Create`, or `Destroy`. For a rule  $r : (L, R)$ , as discussed in Section 2, this marks the variable as belonging to the set of elements to be retained ( $L \cap R$ ), the set of elements to be newly created ( $R \setminus L$ ), or the set of elements to be deleted ( $L \setminus R$ ).

If you're feeling overwhelmed by all the new terms and concepts, don't worry! We will define them again in the context of your chosen syntax with the concrete example. For quick reference, we have also defined the most important terms at the end of this part in a glossary.

### 3.1 Implementing removeCard

- Open LearningBoxLanguage.eap in Enterprise Architect (EA) by double clicking it in Eclipse. Carefully do the following: (1) Click *once* on Partition to select it, then (2) Click *once* on the method `removeCard` to highlight it (Fig. 3.3), and (3) *Double-click* on the chosen method to indicate that you want to implement it.

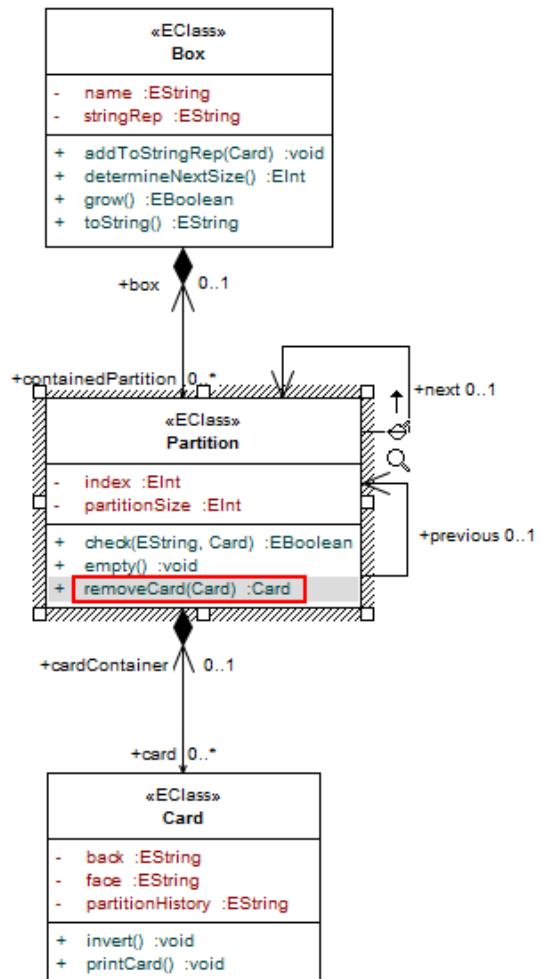


Figure 3.3: Double-click a method to implement it

- If you did everything right (and answered the question which popped up about creating a new SDM diagram with Yes), a new *activity diagram* should be created and opened in a new tab with a cute anchor in the corner, and a *start node* labelled with the signature of the method

(Fig. 3.4).

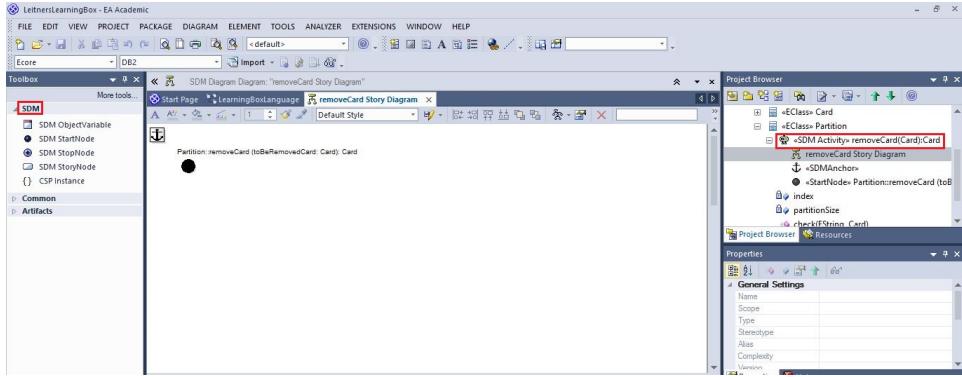


Figure 3.4: Generated SDM diagram and start node

- ▶ This diagram is where you'll model `removeCard`'s *control flow*. In other words, this is `removeCard`'s imperative top-level diagram. We refer to the whole activity diagram simply as the *activity*, which always starts with a *start node*, contains *activity nodes* connected via *activity edges*, then finally terminates with a *stop node*. Before creating these however, let's quickly familiarise ourselves with the EA workspace.
- ▶ First, inspect the project browser and notice that an <<SDM Activity>> container has been created for the method `removeCard`. This container will eventually host every artifact related to this pattern (i.e., object variables, stop nodes, etc.). Please note that if you're ever unhappy with an SDM, you can always delete the appropriate container in the project browser (such as this one), and start from scratch.
- ▶ Next, note the new SDM toolbox that has been automatically opened for the diagram and placed to the left above the common toolbox. This provides quick access to SDM items that you'll frequently use in your diagram. You can also invoke the active toolbox in a pop-up context menu anywhere in the diagram by pressing the **space** bar.
- ▶ Finally, in the top left corner of the diagram, you'll notice a small anchor. Double click on this icon to quickly jump back to the meta-model. From there, double click the method again to jump back to the SDM. This is just a small trick to help you quickly navigate between diagrams.

- To begin, select the start node, and note the small black arrow that appears (Fig. 3.5).

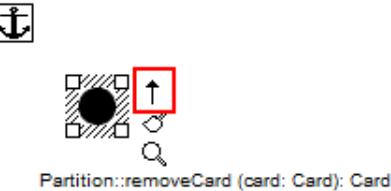


Figure 3.5: Quick link in SDM diagram to create new activity node

- Similar to quick linking,<sup>6</sup> a second fundamental gesture in EA is *Quick Create*. To quick-create an element, pull the arrow and click on an empty spot in the diagram. This is basically “quick linking” to a non-existent element.
- EA notices that there is nothing to quick-link to, and pops up a small, context-sensitive dialogue offering to create an element which can be connected to the source element.
- As illustrated in Fig. 3.6, choose **Append StoryNode** to create a *Story Node*.

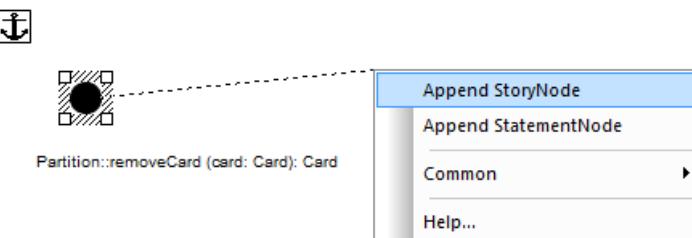


Figure 3.6: Create new activity node

- If you quick-created correctly, you should now have a start node, one node called **ActivityNode1**, and an edge connecting the two items. Complete the activity by quick-creating a stop node (Fig. 3.7).

<sup>6</sup>This was discussed in Part II, Section 2.5

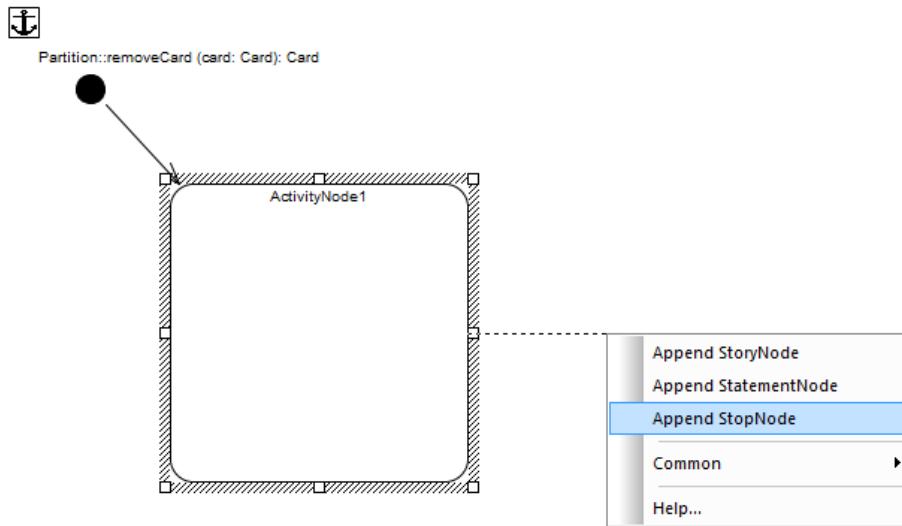


Figure 3.7: Complete the activity with a stop node

- ▶ If everything is correct, you should now have a fully constructed activity that models the method's control flow.
- ▶ While a *stop node* is rather self explanatory, you may be wondering about the differences between the other two menu options, the *story node* and *statement node*. Since not all activity nodes can contain story patterns (e.g., start and stop nodes), those that *can* are called *Story Node*. *Statement nodes* cannot and are used instead to invoke *Action*, such as method execution. We'll encounter this in a later SDM.
- ▶ To complete this activity, double-click *ActivityNode1* to prompt the dialogue depicted in Fig. 3.8. Enter `removeCardFromPartition` as the name of the story node, and select *Create this Object*. Click *OK*. The activity node now has a single *bound object variable*, *this*.
- ▶ To create a new object variable, choose *SDM ObjectVariable* from the toolbox then click inside the activity node (Fig. 3.9). A properties window will automatically appear (Fig. 3.10).

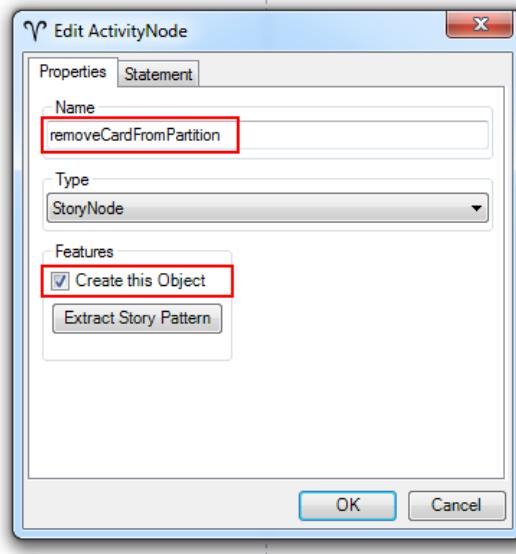


Figure 3.8: Initializing a story node

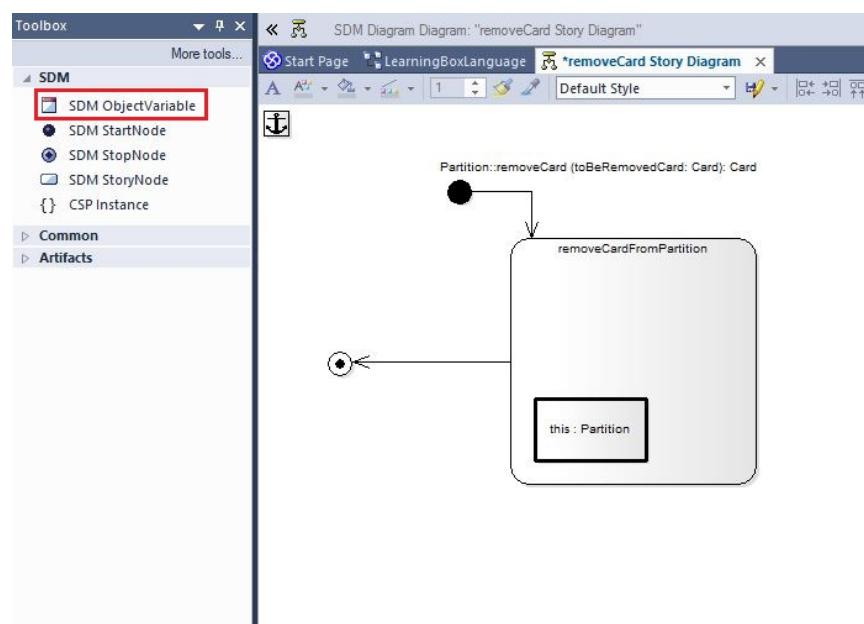


Figure 3.9: Add a new object variable from the toolbox

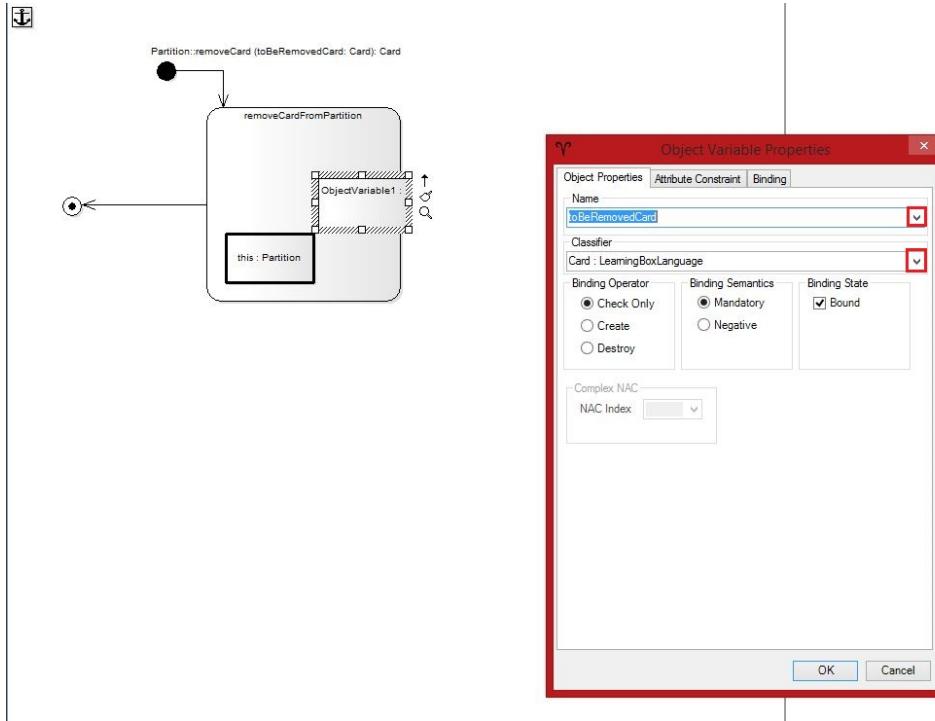


Figure 3.10: Specify properties of the added object variable

- ▶ Using the drop-down menus, choose `toBeRemovedCard` as the name of the object, and set `Card` as its type. Since `card` is a parameter of the method, it is offered as a possible name which can be directly chosen to avoid annoying typing mistakes.
- ▶ In this dialogue, note that the `Bound` option is automatically set. We have now seen two cases in this activity for bound object variables: an assignment to `this`, and an assignment to a method parameter. Setting `toBeRemovedCard` to bound means that it will be implicitly assigned to the parameter with the same name.
- ▶ To create a *link variable* between the current partition and the card to be removed, choose the object variable `this` and quick link it to `toBeRemovedCard` (Fig. 3.11).
- ▶ According to the metamodel, there is only one possible link between a partition and card. Select this and set the `Binding Operator` to `Destroy` (Fig. 3.12). The reference names will automatically appear in the diagram.

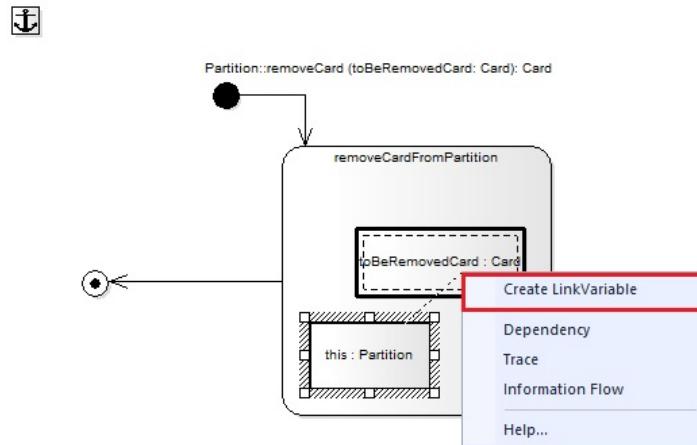


Figure 3.11: Create a link variable

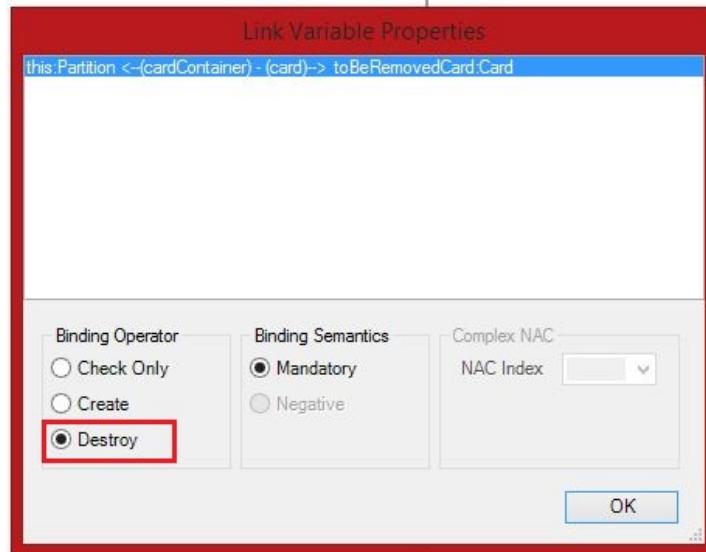


Figure 3.12: Specify properties for created link variable

- Remember how we said that this method should return the same card that was passed in? As luck would have it, a return value for an SDM can be specified in the stop node. As depicted in Fig. 3.13, double-click the stop node to prompt the Edit StopNode dialogue.

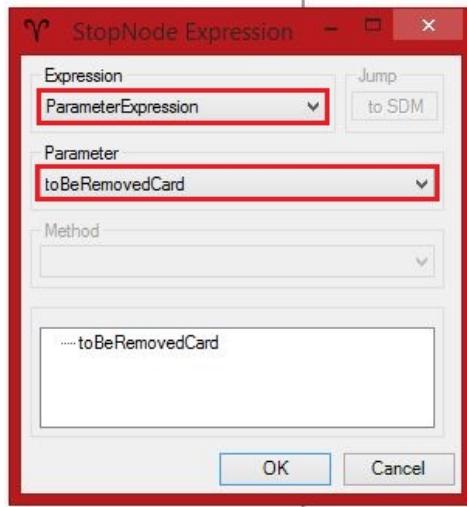


Figure 3.13: Adding a return value to the stop node

- ▶ In the **Expression** field, choose the **ParameterExpression** option. As *ParameterExpression* suggested by its name, a *ParameterExpression* is an expression that exclusively accesses method parameters. Given that `toBeRemovedCard` is the sole parameter, the `object` will be automatically set to this value. In other words, the returned object is now implicitly *bound* by having the same name.

We're nearly done! As you can see, eMoflon uses a series of dialogues to provide a simple context-sensitive expression language for specifying values. In the following SDM implementations, we'll learn and discuss some other expression types eMoflon supports.

- ▶ Returning to the activity, if you've done everything right, your first SDM should resemble Fig. 3.14, where `removeCard`'s entire pattern layer is modeled inside the sole *activity node*. The method's return value is now indicated below the stop node.
- ▶ Don't forget to save your files, validate and export your pattern to the Eclipse workspace,<sup>7</sup> then build your metamodel's code from the package explorer.

---

<sup>7</sup>Go to “Extensions” and select **Add-In Windows** to activate eMoflon’s console. If you’re unsure how to validate, export, or use this window, review Part I, Section 2.1.

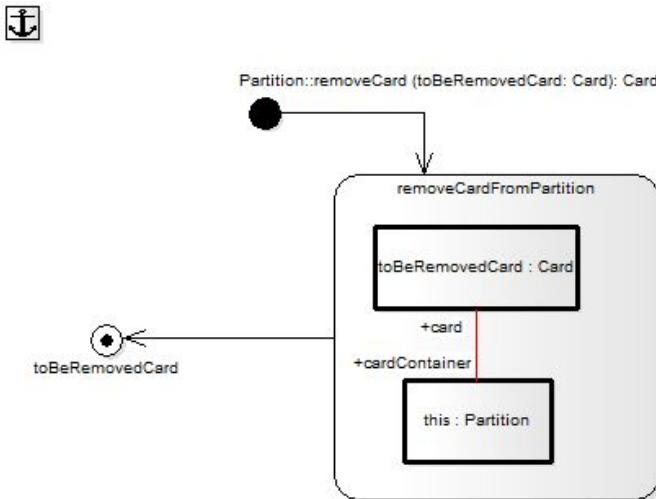


Figure 3.14: Complete SDM for `Partition::removeCard`

- ▶ If you’re unable to export or generate code successfully, compare your SDM carefully with Fig. 3.14 and make sure you haven’t forgotten anything.
- ▶ If you’d like to see how this SDM is implemented in the textual syntax, check out Fig. ?? in the next section.

## Removing SDMs

Imagine that you have just created an SDM for a method, but now you want to implement the method using an injection. This section describes how to get rid of this old SDM.

Let’s assume that you want to remove the SDM of `Partition::removeCard`, which you just created.

- ▶ In the project browser, navigate to the method of which you want to remove the SDM. This can be accomplished by right-clicking the method in the Ecore diagram and then selecting “Find in project browser” (Fig. 3.15), or by navigating manually through the project browser.
- ▶ Inside of the EClass “Partition”, you find the one entry for the method (“`removeCard(Card)`”) and one entry for its SDM (“`<<SDM Activity>> removeCard(Card)`”) (Fig. 3.16).

▷ [Next](#)

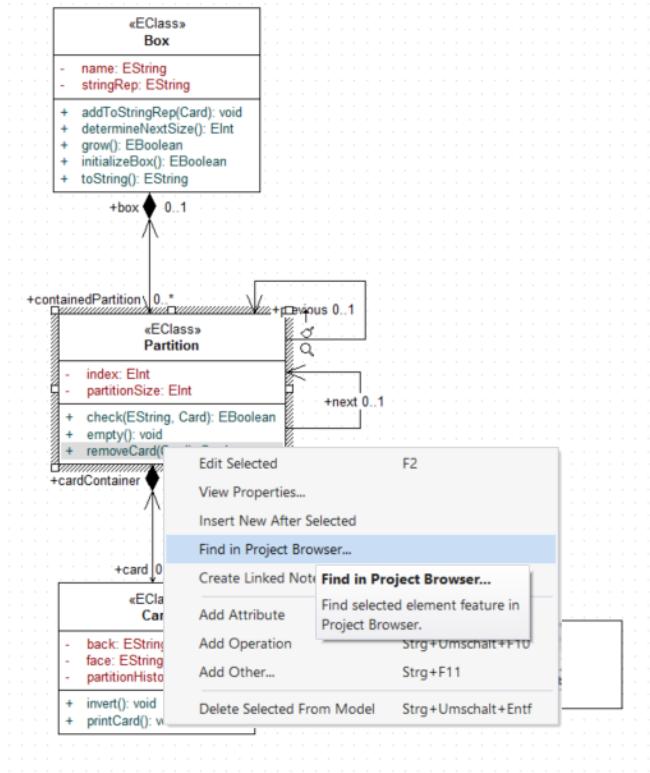


Figure 3.15: Navigate to method in project browser

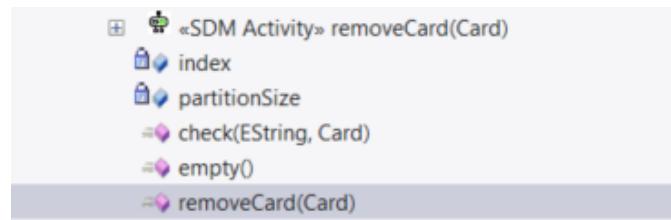


Figure 3.16: Project browser entries for method “removeCard” and its SDM

▷ Next

- ▶ Select and remove the SDM Activity to get rid of the SDM associated with “removeCard(Card)” either by using the context menu entry “Delete ’<<SDM Activity>> removeCard(Card)’” or via *Ctrl+Delete*.  
**Important:** This action cannot be undone!
- ▶ After a successful export and rebuild of your project, you may add an injection for “Partition::removeCard”.

### 3.2 Implementing removeCard

Please note that the textual syntax is not as thoroughly tested as the visual syntax because most of our projects are built with the visual syntax. This means: Whenever something goes wrong even though you are sure to have followed the instructions precisely, do not hesitate to contact us via [contact@emoflon.org](mailto:contact@emoflon.org).

- ▶ Open `Partition.eclass`, go to the `removeCard` signature and add a pair of curly brackets so that it looks like a proper method declaration. This entire scope can be referred to as the method’s *activity*, where the control flow (imperative top-layer) of a transformation is defined.
- ▶ Complete the activity with a single pattern and return statement as depicted in Listing 3.1. Don’t forget – you can use eMoflon’s auto-complete feature here! Press **Ctrl + Space** on an empty line, then select the pattern template to establish `deleteSingleCard`.
- ▶ Note that in this context, the ‘\$’ operator indicates a *ParameterExpression*. This expression implicitly refers to parameters from arguments which are assigned by the operation. In `removeCard`’s case, the *ParameterExpression* returned parameter refers to the `card` parameter.

```

12 removeCard(card : Card) : Card
13 {
14 [deleteSingleCard]
15 return $card
16 }
```

Listing 3.1: Control flow for `removeCard`

- ▶ It should be mentioned that MOSL limits method definitions exclusively to the method’s control flow. All actual transformation rules are modelled in separate *pattern* files. In this case, `removeCard`’s link deletion will be modelled in `[deleteSingleCard]`.
- ▶ Save `Partition.eclass`. An error should immediately appear below the editor. In the “Problems” tab, the message states that the builder cannot find the specified pattern file (Fig. 3.17a). Well, this makes sense. You haven’t created it yet! Click this message and press **Ctrl + 1** to invoke a “Quick Fix” dialogue (Fig. 3.17b). It offers to create a pattern file for you. Given that’s exactly what you’d like, select the option and press **Finish**.

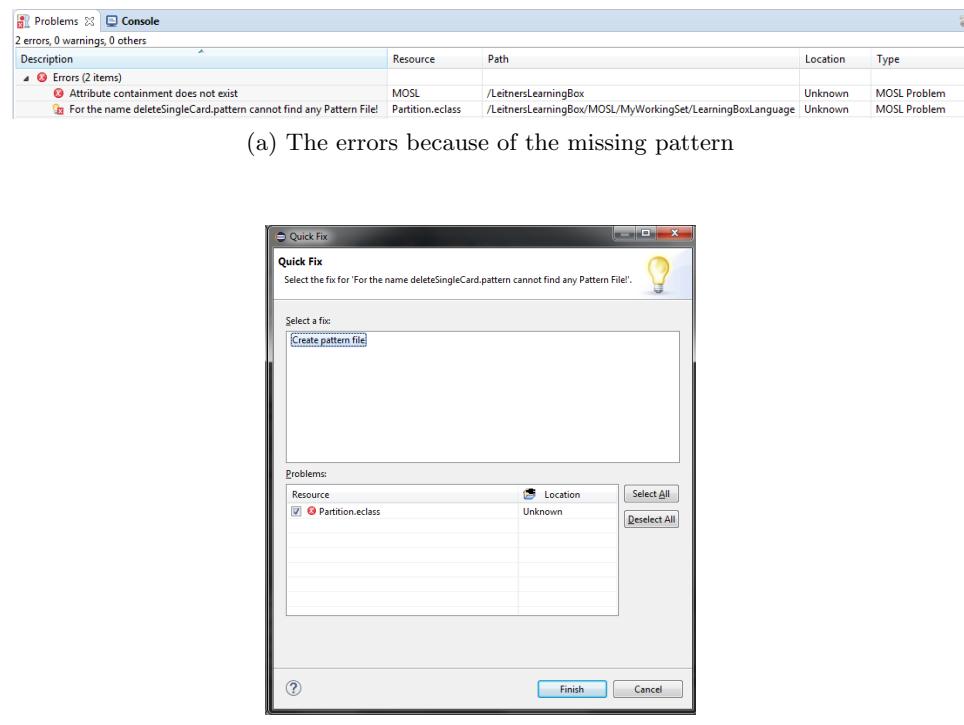
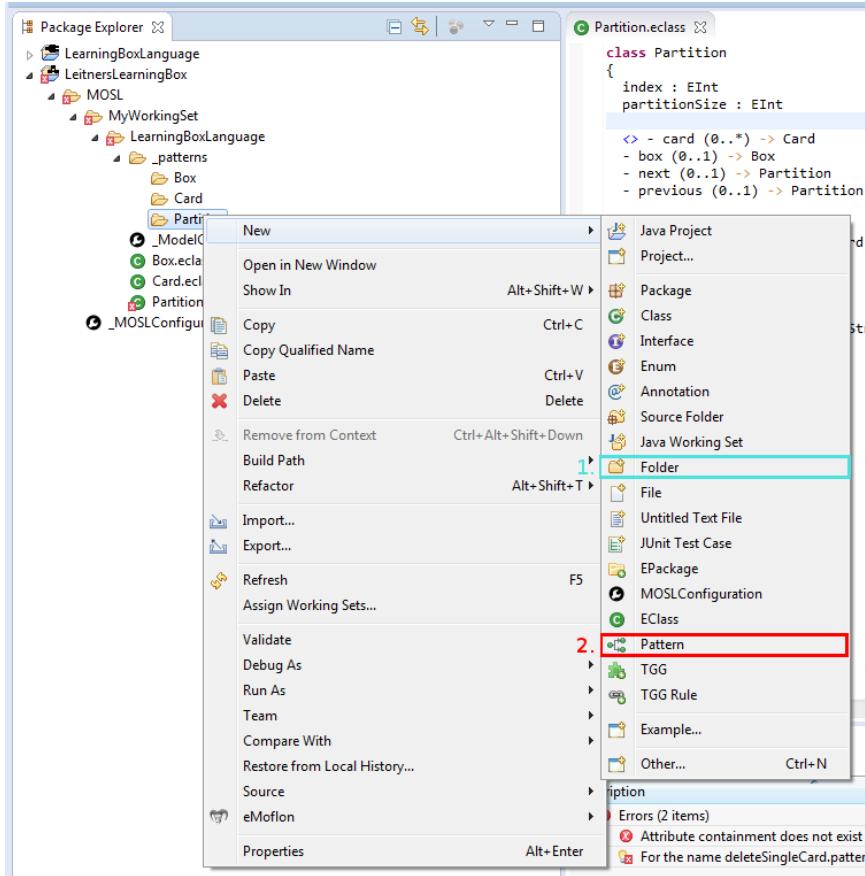
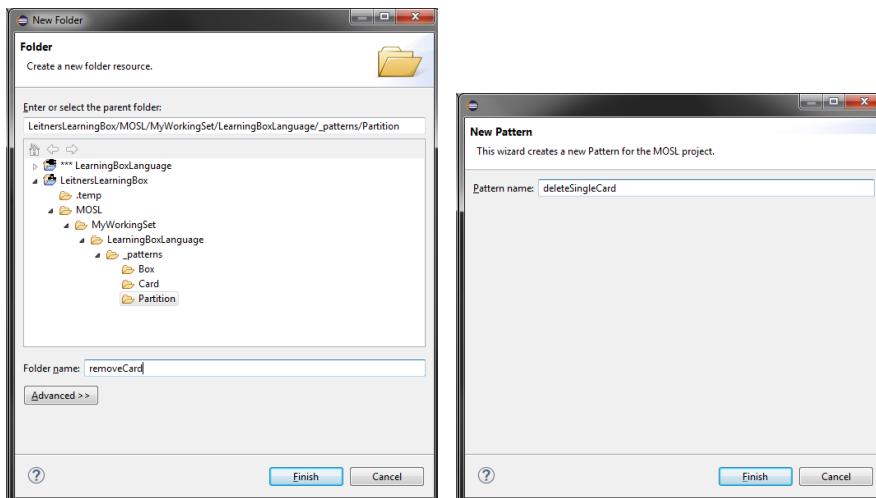


Figure 3.17: Missing pattern error handling

- ▶ Alternatively, you can create a new folder with the name of the operation, in this case `removeCard` and create the `deleteSingleCard` pattern with the wizard.



(a) Menu to create a pattern manually



(b) Wizard to create the removeCard folder (c) Wizard to create the deleteSingleCard pattern

Figure 3.18: Manual way to create a pattern

- ▶ The new file will open in the editor, and you'll be able to see a new directory structure under “LearningBoxLanguage/\_patterns” (Fig. 3.19). To explain, `deleteSingleCard.pattern` is invoked by the method `removeCard`, which is in the `Partition` EClass. `Partition` will eventually contain a folder for each method that uses a pattern.
- ▶ The content of any pattern file is simply a list *object variable scopes*, and then, within such a scope, operations such as ‘delete/create/find’ this outgoing reference. Remember - the main goal of SDMs is to focus here is not on *how*, but *what*.

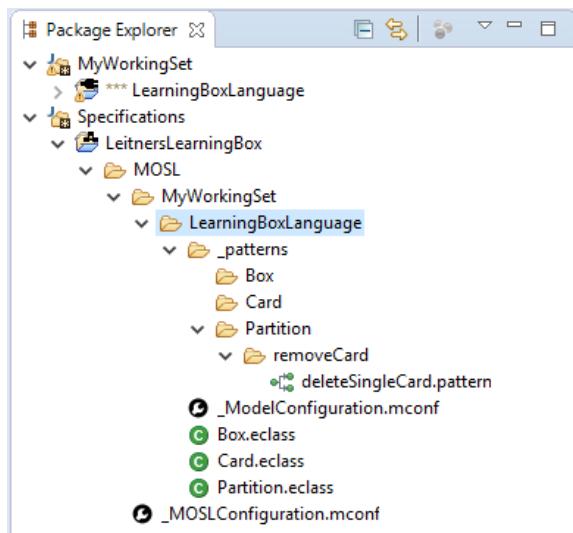


Figure 3.19: Directory structure for a pattern

- ▶ Create two object variables, `@this : Partition` and `@card : Card` (Listings 3.2, Line 3 and 8). When working with MOSL patterns, ‘@’ indicates a *ObjectVariableExpression*. This expression implicitly refers to object variables from the preceding story node. It also indicates that the variable is *bound*. `this` is bound to the object whose method is invoked, while `card` is bound to the value of the parameter with the same name.
- ▶ Object variable scopes determine the changes to be applied to the any exiting references of the variable. Therefore, add:

```
-- -card-> card
```

to `@this` to destroy the `card` reference targeting the `card` object. Your pattern should now resemble Listing 3.2, Line 3.

```

1 pattern deleteSingleCard
2 {
3 @this : Partition
4 {
5 -- - card -> card
6 }
7 @card : Card
8 }
9 }
```

Listing 3.2: Object variables for `removeCard` and destroy the link between a card and its partition

- ▶ In summary, any *outgoing link variable* follows this syntax:

*Outgoing Link Variable*

`[action] '-' nameOfOutgoingLV'-> 'targetOV`

With:

`action := '--' | '++' | '!' | '!.' index`  
`index := NUM`  
`nameOfOutgoingLV := STRING`  
`targetOV := STRING`

- ▶ If you ever need to quickly remind yourself of specific reference or attribute names, press `alt` and the `left` arrow to jump back from your pattern to your EClass. Conversely, to quickly open or jump to a pattern, hover over the pattern name while holding `Ctrl` until it's underlined, then click!
- ▶ Remember, links between classes can be specified as *bidirectional EReferences*,<sup>8</sup> linked together as opposites in “LearningBoxLanguage/\_ModelConfiguration.mconf.” In this case, therefore, we don't need to worry about declaring `-- -cardContainer-> Card` inside `card`, as it would be redundant.
- ▶ Save and build your metamodel. If any errors occur, double check and make sure your activity and pattern match ours.
- ▶ To see how the same method is crafted in the visual syntax, check out Fig. 3.14 from the previous section.

---

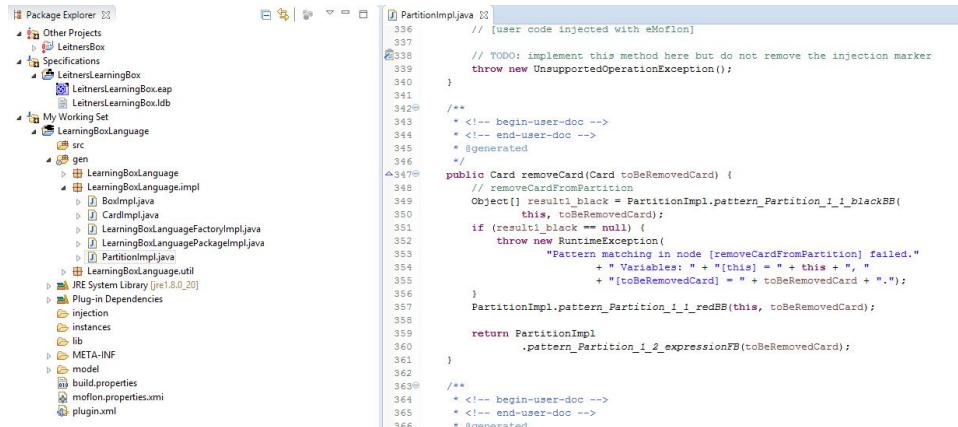
<sup>8</sup>Technically two *unidirectional EReferences*. Refer to Part II, Section 2.5.

## Concluding removeCard

Fantastic work! You have now implemented a simple method via patterns. As you can see, SDMs are effective for implementing structural changes in a high-level, intuitive manner.

Let's take a step back and briefly review what we have specified: if `p.removeCard(c)` is invoked for a partition `p`, with a card `c` as its argument, the specified pattern will *match* only if that card is contained in the partition. After determining matches for all variables, the link between the partition and the card is deleted, effectively “removing” the card from the partition. If the card is *not* contained in the partition, the pattern won't match, and nothing will happen. In both cases, the card that's passed in is returned.

- ▶ If your code generation was successful, navigate to “LearningBox-Language/gen/LearningBoxLanguage/impl/PartitionImpl.java” to the `removeCard` declaration (approximately line 347). Inspect the generated implementation for your method (Fig. 3.20). Notice the null check that is automatically created - only a very conscientious (and probably slightly paranoid) programmer would program so defensively!



```

336 // [user code injected with eMoflon]
337
338 // TODO: implement this method here but do not remove the injection marker
339 throw new UnsupportedOperationException();
340 }
341
342 /**
343 * <!-- begin-user-doc -->
344 * <!-- end-user-doc -->
345 * @generated
346 */
347 public Card removeCard(Card toBeRemovedCard) {
348 // removeCardFromPartition
349 Object[] result_black = PartitionImpl.pattern_Partition_1_1_blackBB(
350 this, toBeRemovedCard);
351 if (result_black == null) {
352 throw new RuntimeException(
353 "Pattern matching in node [removeCardFromPartition] failed."
354 + " Variables: " + "[this] = " + this + ", "
355 + "[toBeRemovedCard] = " + toBeRemovedCard + ".");
356 }
357 PartitionImpl.pattern_Partition_1_1_redBB(this, toBeRemovedCard);
358
359 return PartitionImpl
360 .pattern_Partition_1_2_expressionFB(toBeRemovedCard);
361 }
362
363 /**
364 * <!-- begin-user-doc -->
365 * <!-- end-user-doc -->
366 * @generated
367 */

```

Figure 3.20: Generated implementation code

---

Near the end of Part II (after using injections), you were able to test this method's implementation using our `LeitnersBoxGui`. Let's run it again to make sure *this* version of `removeCard` works!

- ▶ Load and run the GUI as an application,<sup>9</sup> then go to any partition and select `Remove Card` (Fig. 3.21). It should immediately refresh, and you'll no longer be able to see the card in either the GUI or in the `Box.xmi` tree in the “instances” folder. Pretty cool, eh?

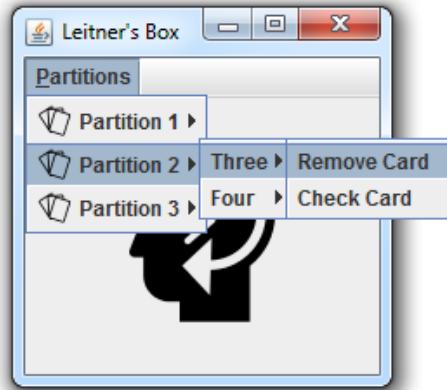


Figure 3.21: Testing `removeCard`

---

<sup>9</sup>Refer to Part II, Section 6 for details on our GUI

## 4 Checking a card

The next method we shall model is probably the most important for our learning box. This method will be invoked when a user decides to test themselves on a card in the learning box. They'll be able to see the `back` attribute of a card from the box, make a guess as to what's on the `face`, then check their answer (Fig. 4.1). Following our rules established in Fig. 1.1, if their guess was correct the card will be *promoted* to the next partition. If wrong, the card will be *penalized* and returned to the first.

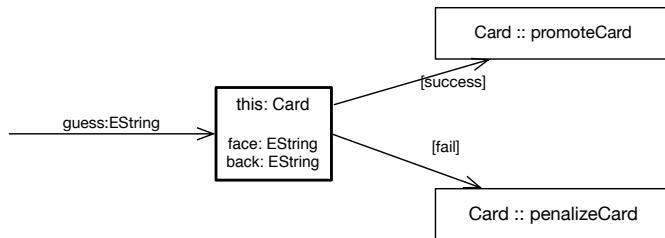


Figure 4.1: Checking a card with a guess

As you can see, checking `guess` is a simple assertion on string values. The actual movements of the card however, must be implemented as separate patterns. Fig. 4.2 briefly shows the intended create and destroy transformations.

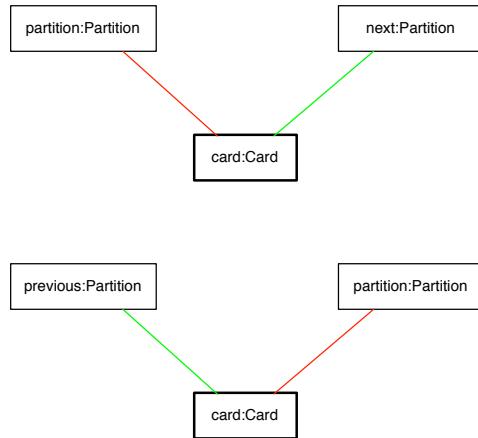


Figure 4.2: Promote (above) and penalize (below) card patterns

Overall, this means the control flow must utilize an *if/else* construct. The `guess` conditional also needs to be an *attribute constraint*, a non-structural condition that must be satisfied in order for a story pattern to match.

*Attribute Constraint*

- ▷ Next [visual]
- ▷ Next [textual]

#### 4.1 Implementing check

- ▶ Since you're nearly an SDM wizard already, try using concepts we have already learnt to create the control flow for `Partition::check` as depicted in Fig. 4.3.

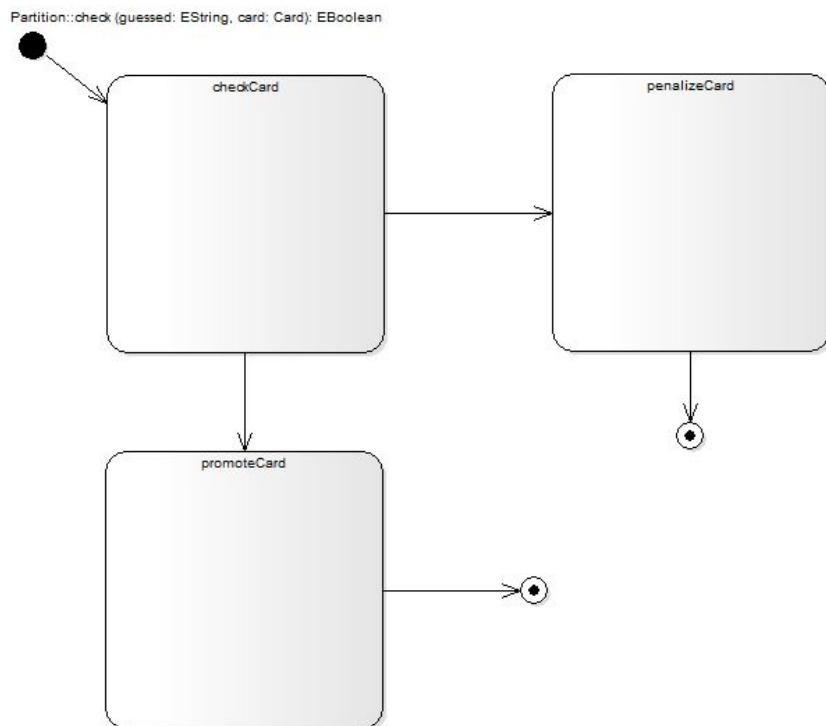


Figure 4.3: Activity diagram for `Partition::check`

- ▶ In `checkCard`, create an object variable that is bound to the parameter argument, `card` (Fig. 4.4). This will represent the card the user picked from the learning box. Remember, the binding for this variable is implicitly defined because its name is the same as the argument's.

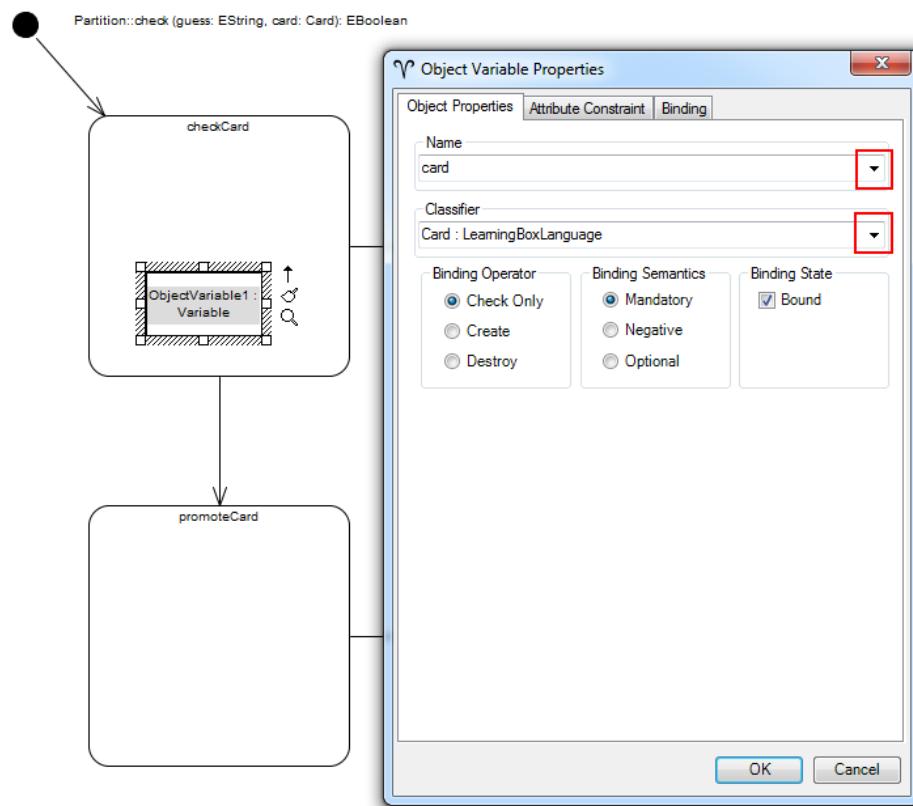


Figure 4.4: Creating the card variable

- Now that the pattern has the correct card to check, it needs to compare the user's guess against the unseen `face` value on the opposite side. To do this, we need to specify an *attribute constraint*. Open the **attribute constraint** tab for `card` as depicted in Fig. 4.5, and select the correct **Attribute** and **Operator**.

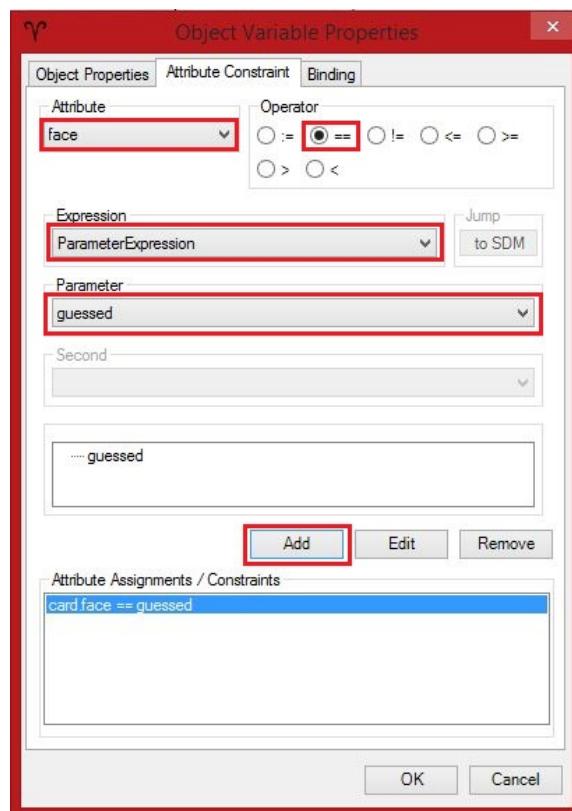


Figure 4.5: Creating an attribute constraint

- Similar to how the return value was specified in the previous SDM, set the `ParameterExpression` to refer to `guess`, i.e., the user's EString input. Press `Add`, and admire your first conditional.

Before building the other two activity nodes, let's quickly return to the control flow. Currently, the pattern branches off into two separate patterns after completing the initial check, and it is unclear how to terminate the method. As the code generator does not know what to do here, this is flagged as a validation error (you're free to press the validation button and take cover). We need to add *edge guards* to change this into an *if/else Edge Guards* construct based on the results of the *attribute constraint*.

- To add a guard to the edge leading from `checkIfGuessIsCorrect` to `penalizeCard`, double click the edge and set the *Guard Type* to **Failure** (Fig. 4.6). Repeat the process for the Success edge leading to `promoteCard`.

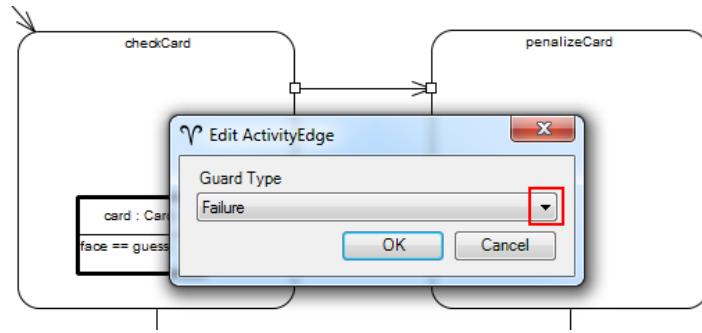


Figure 4.6: Add a transition with a guard

**Edge guards: Success, Failure, or None?** You may now wonder why we did not use edge guards already for implementing `removeCard`. In fact, an unguarded activity edge is equivalent with a Success edge. More precisely, if you create an activity node only one outgoing unguarded edge you assert that the pattern contained in this activity node will *always* be successful. In case the pattern is not applicable, a runtime exception will be thrown. In contrast, having an outgoing Success and Failure edge describes that you allow your pattern to not match.

**Extracting Story Patterns** One great feature of eMoflon (with EA) is a means of coping with large patterns. It might be nice to visualise *small* story patterns directly in their nodes (such as `removeCardFromPartition`), but for large patterns or complex control flow, such diagrams would get extremely cumbersome and unwieldy *very* quickly! This is indeed a popular argument against visual languages and it might have already crossed your mind – “This is cute, but it’ll *never* scale!” With the right tools and concepts however, even huge diagrams can be mastered. eMoflon supports *extracting* story patterns into their own diagrams, and unless the pattern is really concise with only 2 or 3 object variables, we recommend this course of action. In other words, eMoflon supports separating your transformation’s pattern layer from its imperative control flow layer.

- To try this, double-click the `promoteCard` story node and choose **Extract Story Pattern** (Fig. 4.7). Note the new diagram that is immediately created and opened in the project browser (Fig. 4.8).

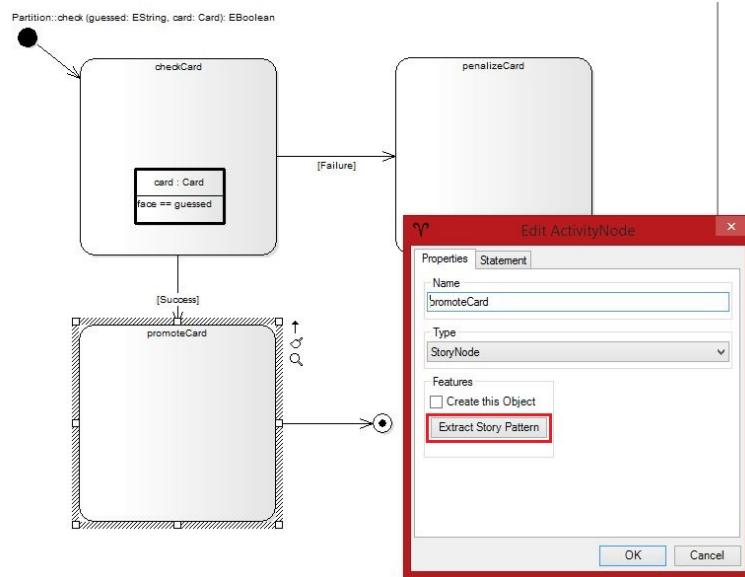


Figure 4.7: Extract a story pattern for more space and a better overview

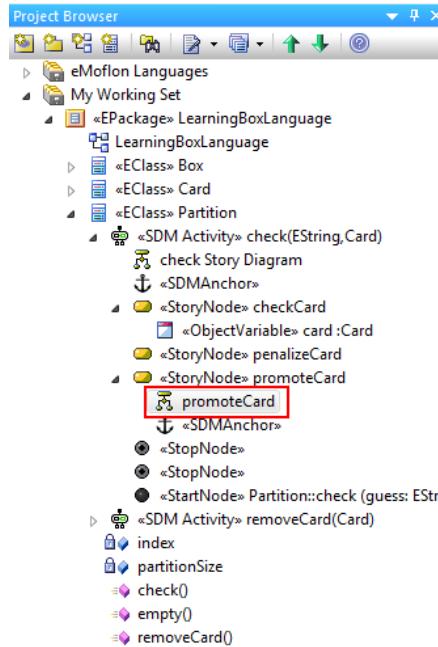


Figure 4.8: A new subdiagram is created automatically

Another EA gesture<sup>10</sup> you could start to take advantage of here is good ol' *Drag-and-Drop* from the project browser into a diagram. We can use this action as an alternative to creating new objects (with known types) from the SDM toolbox.

The main advantage of drag-and-drop is that the **Object Variable Properties** dialogue will have the type of the object pre-configured. Choosing the type in the project browser and dragging it in is (for some people) a more natural gesture than choosing the type from a long drop-down menu (as we had to when using the SDM toolbar). This can be a great time saver for large metamodels.<sup>11</sup>

- ▶ To put this into practice, create a new **Card** object variable by drag-and-dropping the class from the project browser into the new (extracted) pattern diagram (Fig. 4.9).
- ▶ A dialogue will appear asking what kind of visual element should be

<sup>10</sup>The other two gestures we have learnt are “Quick Link” and “Quick Create”

<sup>11</sup>Drag-and-drop is also possible in embedded story patterns (those still visualised in their story nodes). You must ensure however, that the object variable is *completely* contained inside the story node, and does not stick out over any edge.

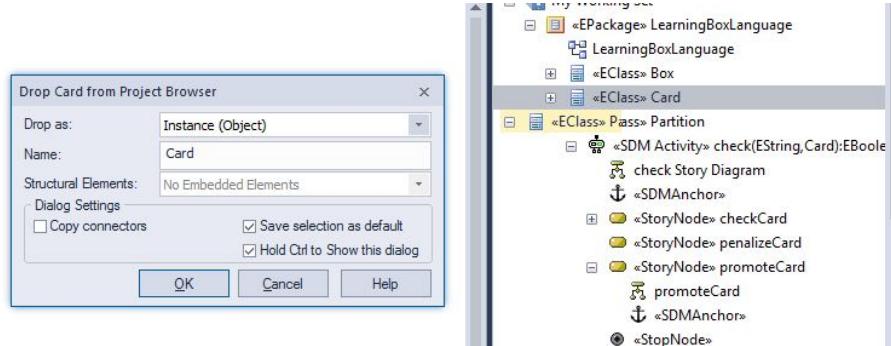


Figure 4.9: Add a new object variable per drag-and-drop

created. You can create (1) a simple link (which would refer to and be represented by the class `Card`), (2) create an instance of `Card` as an object variable, or (3) as an invocation (which has no meaning for eMoflon diagrams). Paste `Card` as an `Instance`, and select `Autosave Selection as Default` under “Options” so option (2) will be used next time by default. You should also select `Use Ctrl + Mouse drag to show this Dialog`, so this dialogue doesn’t appear every time you use this gesture. Don’t worry – if you ever need option (1), hold `Ctrl` when dragging to invoke the dialogue again.

- ▶ After creating the object, the object properties dialogue will open. Set the `Name` to `card` and confirm its `Binding State` is `Bound` (Fig. 4.10).
- ▶ Currently, we have the single `card` that we want to promote through the box. Drag-and-drop two partition objects, `this`, and `nextPartition` as depicted in Fig. 4.11.

An important point to note here is that `this` and `card` are visually differentiated from `nextPartition` by their bold border lines. This is how we differentiate *bound* from *unbound (free)* variables. We already know that matches for bound variables are completely determined by the current context. On the other hand, matches for unbound variables have to be determined by the pattern matcher. Such matches are “found” by navigating and searching the current model for possible matches that satisfy all specified constraints (i.e., type of the variable, links connecting it to other variables, and attribute constraints). In our case, `nextPartition` should be determined by navigating from `this` via the `next` link variable.

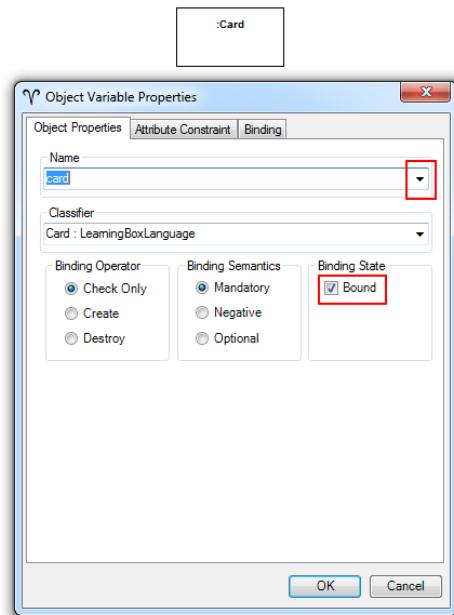


Figure 4.10: Object variable properties of the new card

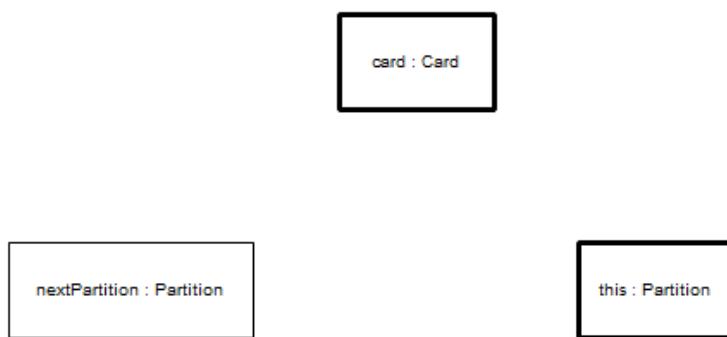


Figure 4.11: All object variables for story pattern `promoteCard`

- To specify this, quick link from `this` to `nextPartition` (or vice-versa) to establish `next`, as shown in Fig. 4.12. As you can see, there are several more options than what was seen in `removeCard`. The goal is to have the current partition to proceed (or point) to the `nextPartition` via the `next` reference, so select the second option. Alternatively, you could define the reference from `nextPartition` by setting the link variable `previous` to `this`.

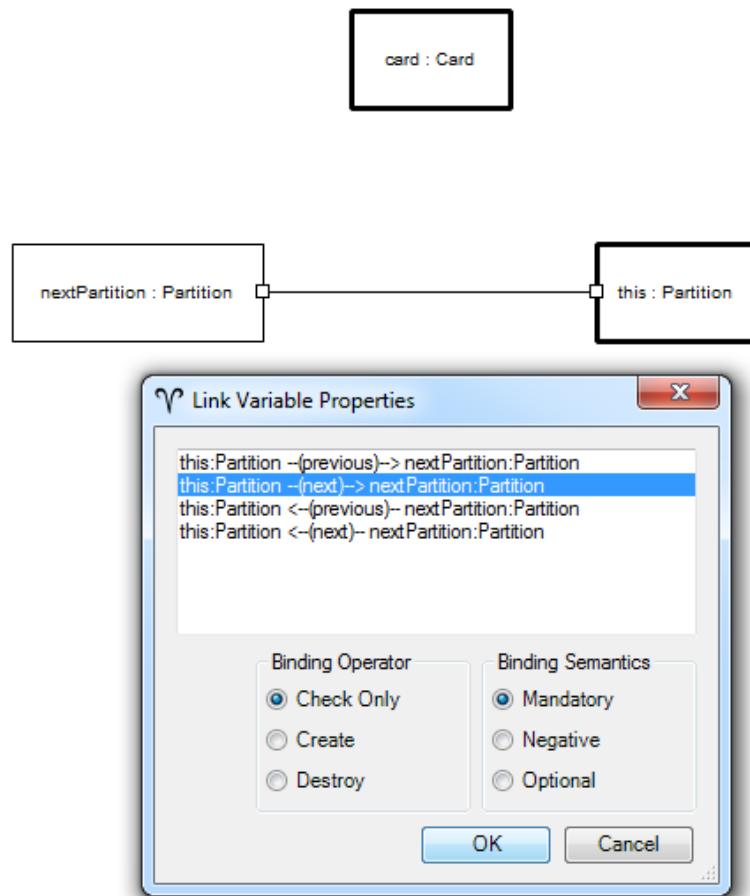


Figure 4.12: Possible links between `this` and `nextPartition`

- Continue by creating links between `card` and each partition. Remember - you want to *destroy* the reference to `this`, and *create* a new connection to `nextPartition`. If everything is set up correctly, `promoteCard` should now closely resemble Fig. 4.13.

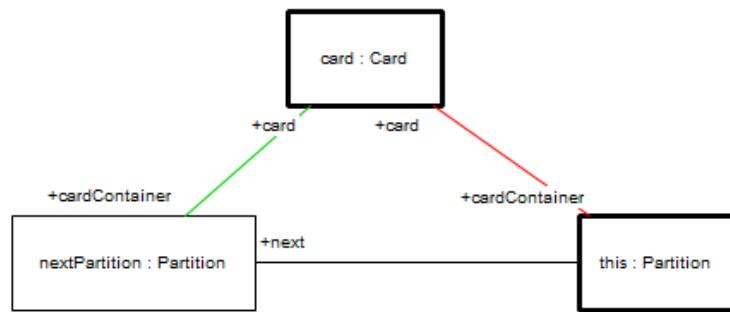


Figure 4.13: Complete story pattern for `promoteCard`

- Double click the anchor in the top left corner and repeat the process for `penalizeCard`: First extract the story pattern, then create the necessary variables and links as depicted in Fig. 4.14. As you can see, this pattern is nearly identical to `promoteCard`, except it moves `card` to a `previousPartition`.

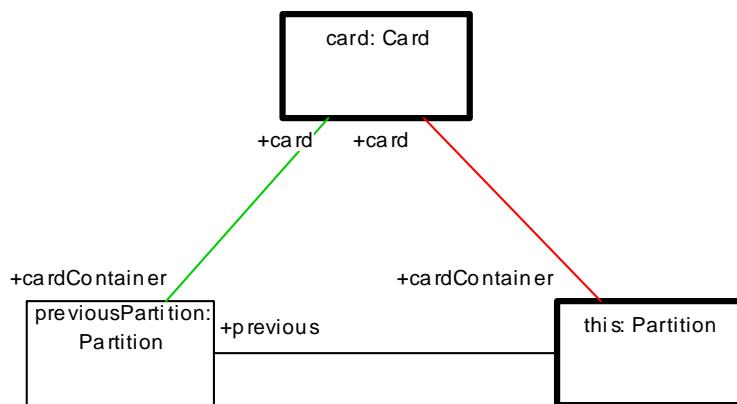


Figure 4.14: Complete story pattern for `penalizeCard`

To complete the `check` activity, we need to signal (as a return value) the result of the check - was the card promoted or penalized? We have no object to return so instead, we need to edit the stop nodes so they return a *LiteralExpression*. This expression type can be used to specify arbitrary text, but should really only be used for true literals like 42, “foo” or `true`. It can be (mis)used for formulating any (Java) expression that will simply be transferred “literally” into the generated code, but this is obviously really dirty<sup>12</sup> and should be avoided when possible.

*LiteralExpression*

- To implement a literal, double click the stop node stemming from `promoteCard`, and change the expression type from `void` to `LiteralExpression` (Fig. 4.15). Change the value in the window below to `true`. Press `OK`, then finish the SDM by returning `false` after `penalizeCard` in the same manner. Note that it is also possible that `promoteCard` or `penalizeCard` fails<sup>13</sup>. This is handled by placing parallel Success

<sup>12</sup>It defeats, for example, any attempt to guarantee type safety

<sup>13</sup>For example, if the actual partition doesn't have a “proper” successor or predecessor – in this case, unfortunately, we can't give a real reward or penalty for the guess, which means that we still return `true` or `false`.

and Failure edges after the `promoteCard` and `penalizeCard` activities. After doing these, your diagram should resemble Fig. 4.16.

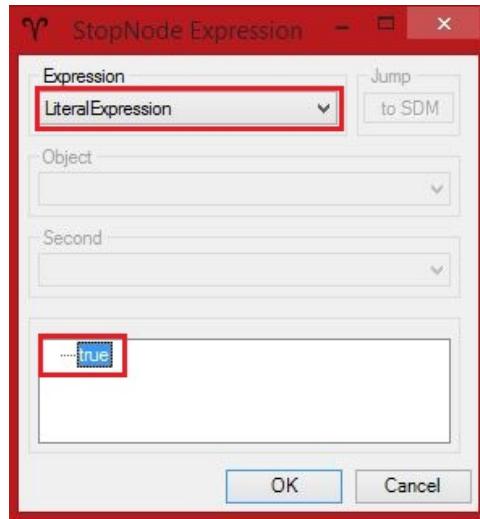


Figure 4.15: Add a return value with a literal expression

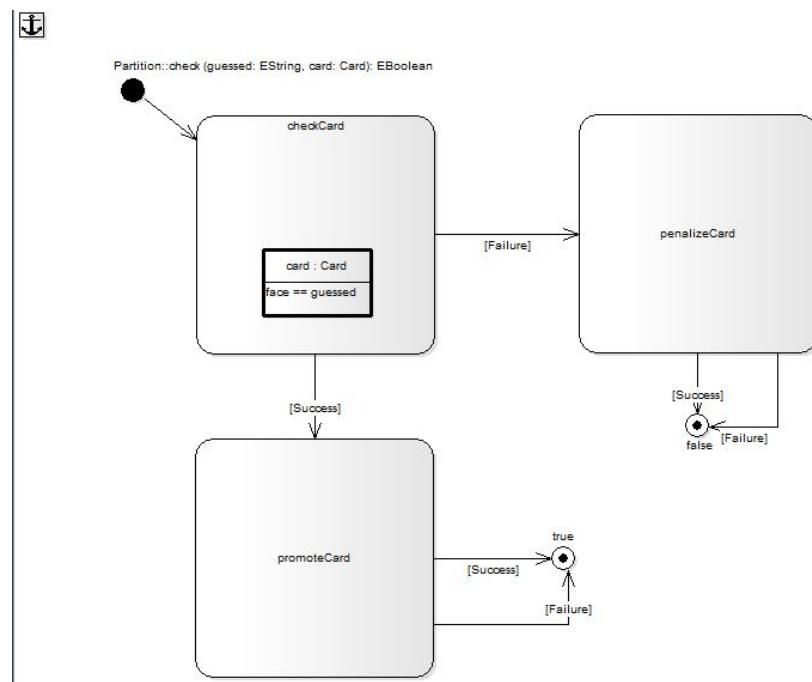


Figure 4.16: Complete SDM for `Partition::check`

- ▶ Great job – the SDM is now complete! Validate and export your project, then inspect the implementation code for `check`. We strongly recommend that you even write a simple JUnit test (take a look at our simple test case from Part I for inspiration) to take your brand new SDM for a test-spin.
- ▶ To see how this is implemented in the textual syntax, see Figs. 4.21 and 4.1 in the following section.

## 4.2 Implementing check

- ▶ In this SDM, guess assertion, card promotion, and card penalization must each be implemented as patterns. Given that every action is determined as the result of a conditional statement, we need an *if/else* construct.
- ▶ In the `check` method, create the basic *if/else* construct with three patterns, as illustrated in Listing 4.1, Line 19, 21 and 26. Our auto completion feature includes a template for this.
- ▶ Upon saving, use the “Quick Fix” wizard again to generate the required pattern files. Your package explorer should now resemble Fig. 4.17.

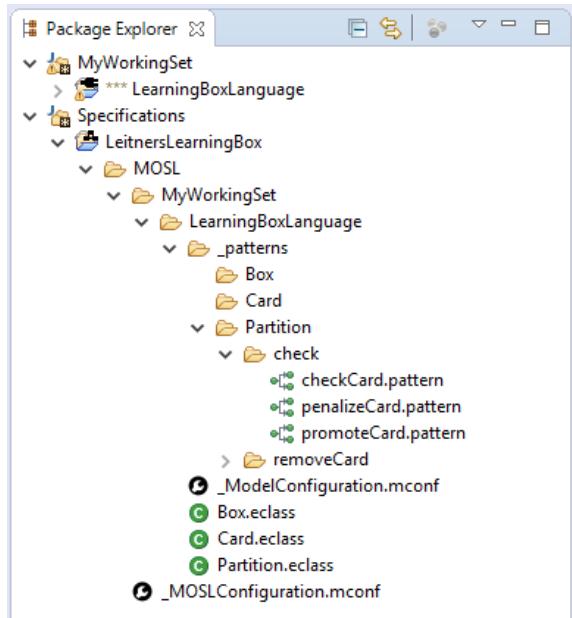


Figure 4.17: Project directory after creating patterns

- ▶ Open the `checkCard` pattern. In order to validate the user’s guess, we need to establish an *attribute constraint* between the `face` attribute of the current `card`, and the primitive `EString` parameter, `guess`.
- ▶ As done in Java, referencing the `face` of a card is done in MOSL via the ‘dot’ operator, so begin the statement with `card.face`
- ▶ Attribute constraints have similar operators as Java comparators, so we’ll want to use ‘`==`’ to equate the values.<sup>14</sup>

---

<sup>14</sup>See the Quick Reference at the end of this part for a listing of all operators

- The tricky part of the overall statement is referencing the parameter value. Given that we have not created an object variable, we'll need to use another expression type, `ParameterExpression`, to access it. *ParameterExpression* This type exclusively refers to parameter values, and its syntax is as follows:

```
parameter_expression := '$'ID
```

With:

```
ID := STRING
```

- Thus, the complete attribute constraint statement is:

```
card.face == $guess
```

Your pattern should now resemble Fig. 4.18.

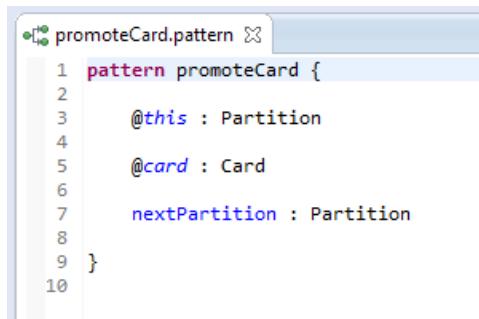
```
1 pattern checkCard {
2
3 @card : Card {
4 card.face == $guess
5 }
6
7 }
```

Figure 4.18: Completed `checkCard` pattern

- Now let's specify the `promoteCard` pattern. It requires three object variables: the current partition (`this`), the card to be promoted, and the partition the card will move to. Open and edit so that it resembles Fig. 4.19.
- Given that `this` is bound while `next` partition is free, we need to establish a reference link between the `box` and partition. This will allow the card to be moved around. Create an *outgoing link variable*, `next`, with target object variable `nextPartition`. Your file should now resemble Fig. 4.20.
- Finally, let's update the `cardContainer` reference of `card`. Simply delete the link to `this`, and create a new link to `nextPartition`.<sup>15</sup> Your pattern is now complete.

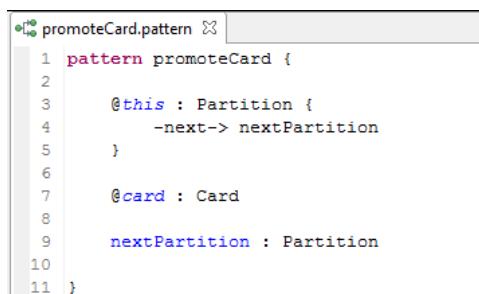
---

<sup>15</sup>There are several different ways you could have implemented this movement, such as updating the link from `nextPartition` to `card`.



```
❶ pattern promoteCard {
❷ @this : Partition
❸ @card : Card
❹ nextPartition : Partition
❺ }
```

Figure 4.19: Object variables for promoting a card



```
❶ pattern promoteCard {
❷ @this : Partition {
❸ -next-> nextPartition
❹ }
❺ @card : Card
❻ nextPartition : Partition
❼ }
```

Figure 4.20: The @this object variable

- Compare the differences between the `promoteCard` and `penalizeCard` concepts. They both do the exact same thing, except the destination partition is different. Knowing this, try to complete `penalizeCard` entirely on your own. Your workspace should come to resemble Fig. 4.21.

```

1 pattern promoteCard {
2
3 @this : Partition {
4 -next-> nextPartition
5 }
6
7 @card : Card {
8 -- -cardContainer-> this
9 ++ -cardContainer-> nextPartition
10 }
11
12 nextPartition : Partition
13
14 }

```

```

1 pattern penalizeCard {
2
3 @card : Card {
4 -- -cardContainer-> this
5 ++ -cardContainer-> previousPartition
6 }
7
8 @this : Partition {
9 -previous-> previousPartition
10 }
11
12 previousPartition : Partition
13
14 }

```

Figure 4.21: Both movement patterns completed

- ▶ Did you notice that the order of the `this` and `card` object variable scopes are reversed in the figures above? In general, it doesn't matter in which order object or variables are specified in patterns. Everything just needs to be correctly stated.
- ▶ We nearly forgot to complete the control flow! We have specified the assertion and card movements, but we haven't returned a `EBoolean` result as the method requires. We'll need to implement a new expression type, a *LiteralExpression*. This type can be used to specify *LiteralExpression* arbitrary text, but should really only be used for true literals like `42`, `"foo"` or `true`. The syntax for any `LiteralExpression` is simply:

```
LiteralExpression ::= boolean_literal | integer_literal |
any_literal
```

With:

```
boolean_literal ::= true, false
integer_literal ::= ['+' | '-'] ('1' | ... | '9')
any_literal ::= SINGLE_QUOTED_STRING
```

- ▶ Knowing this, and that `guess` was successful when the card was promoted, return `true` beneath `promoteCard`. If it was penalized, return `false`. Your `check` method should now resemble Listing 4.1.

```

17 check(card : Card, guess : EString) : EBoolean
18 {
19 if [checkCard]
20 {
21 [promoteCard]
22 return true
23 }
24 else
25 {
26 [penalizeCard]
27 return false
28 }
29 }
```

---

Listing 4.1: Completed control flow for `check` with an if/else construct

- ▶ Save and build your metamodel, then try viewing the changes in “PartitionImpl.Java” under `check`. You’ll be able to see the *if/else* construct, as well as the link manipulations.
- ▶ Great job! You have just enabled checking a card with guesses via a new control flow construct, *if/else*, and two patterns that move the cards appropriately. To see how this SDM is implemented visually, check out Fig. 4.16 for the control flow, and Figs. 4.13 and 4.14 for the movement patterns, all in the previous visual section.

## 5 Running the Leitner's Box GUI

In addition to `removeCard`, the GUI is already able to access and execute the `check` method based on your SDM implementation. First, double check that your metamodel is saved and built, then run the GUI.

- ▶ Pick a card from any of your partitions, then run `check`. You'll be prompted with a dialogue box to make your guess in (Fig. 5.1).



Figure 5.1: Enter your guess

- ▶ Enter your word, then press `OK`. You should immediately see any movement changes in the drop-down menus, and shortly in `box.xmi` after refreshing.
- ▶ Fully test your implementation by making right and wrong guesses. Watch how the cards move around – do they behave as expected, following the rules of Leitner's Box?
- ▶ At this point, we invite you to browse the `LeitnersBoxController.java` file. Can you see how `removeCard` and `check` were called and executed? You are encouraged to modify this file so that you may be able to test your future SDM implementations.

## 6 Emptying a partition of all cards

This next SDM should *empty* a partition by removing every card contained within it. Since we can assume that there is more than one card in the partition,<sup>16</sup> we obviously need some construct for repeatedly deleting each card in the partition (Fig. 6.1).

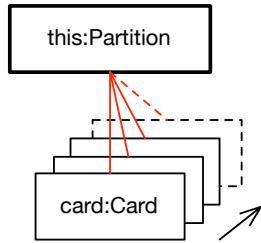


Figure 6.1: Emptying a partition of every card

In SDM, this is accomplished via a *for each* story node. It performs the *For Each* specified actions for *every* match of its pattern (i.e., every **Card** that matches the pattern will be deleted). This however, gives us two interesting points to discuss. Firstly, how would the pattern be interpreted if the story node were a normal, simple control flow node, not a *for each* node?

The pattern would specify that *a* card should be matched and deleted from the current partition - that's it. The *exact* card is not specified, meaning that the actual choice of the card is *non-deterministic* (random), and it is only done once. This randomness is a common property of graph pattern matching, and it's something that takes time getting used to. In general, there are no guarantees concerning the choice and order of valid matches. The *for each* construct however, ensures that *all* cards will be matched and deleted.

The second point is determining if we actually need to destroy the link between **this** and **card**. Would the pattern be interpreted differently if we destroyed **card** and left the link?

The answer is no, the pattern would yield the same result, regardless of whether or not the link is explicitly destroyed! This is due to the transformation engine eMoflon uses.<sup>17</sup> It ensures that there are never any *dangling edges* in a model. Since deleting just the **card** would result in a dangling edge attached to **this**, that link is deleted as well. Explicitly destroying the links as well is therefore a matter of taste, but ... why not be as explicit as possible?

<sup>16</sup>If there was only one, we would just invoke `removeCard`

<sup>17</sup>CodeGen2, a part of the Fujaba toolsuite <http://www.fujaba.de/>

▷ Next [visual]

▷ Next [textual]

## 6.1 Implementing empty

- ▶ Create a new activity diagram for `Partition.empty()`. To begin building the *for each* pattern, quick create a new story node and edit its properties. Name it `deleteCardsInPartition` and change its Type from `StoryNode` to `ForEach`. You'll also want to create the invoking `Partition` object, `this` (Fig. 6.2). Press `OK`, and you'll see that a *for each* node is represented as a stacked node to indicate the potential for repetition.

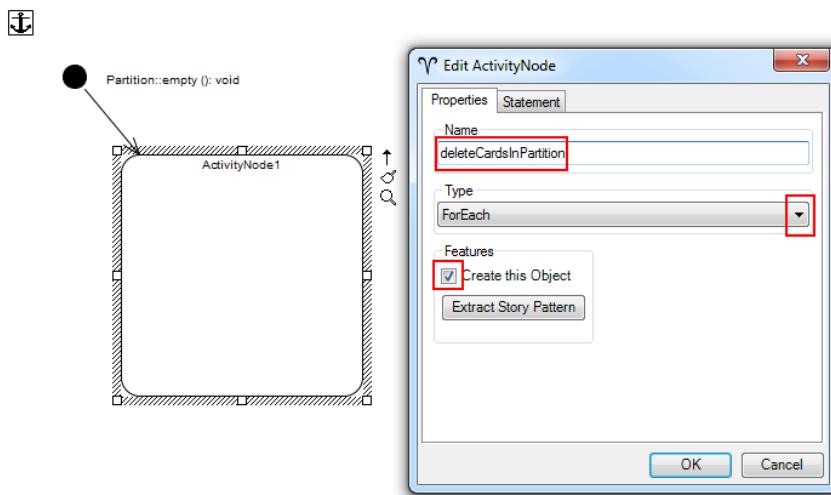


Figure 6.2: Creating a looping story node

- ▶ Now create the `card` object variable needed to complete this SDM. Unlike `removeCard` (Fig. 3.14) however, the goal of `emptyCards` is not just to remove the link between the selected partition and card, we want the matched `card` to be *completely* deleted. This means in the properties tab, after setting the name and binding state, you'll need to set the Binding Operator to Destroy (Fig. 6.3).
- ▶ Complete the story pattern as indicated in Fig. 6.4. Notice that the guard that terminates the looping node has an `[end]` edge guard. Indeed, a *for each* story node *must* execute an `end` activity when all matches in the pattern have been handled. `empty` is defined as a `void` method, so don't worry about setting any return value in the stop node.

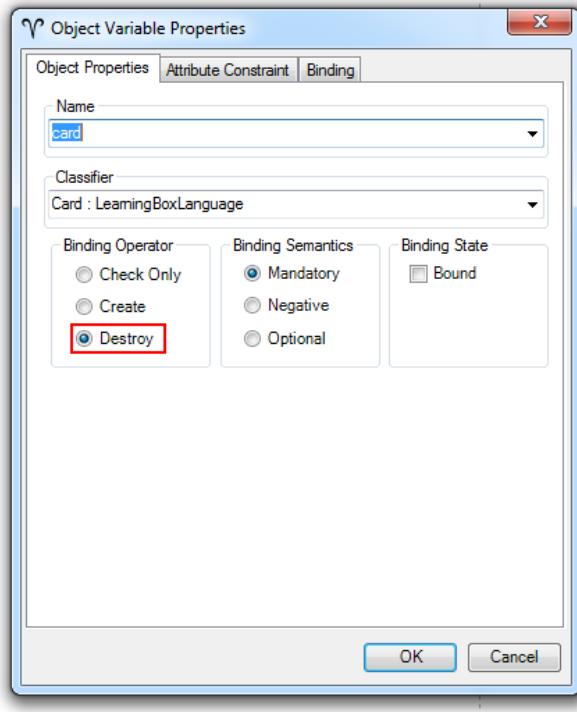


Figure 6.3: Editing `card` so it gets destroyed

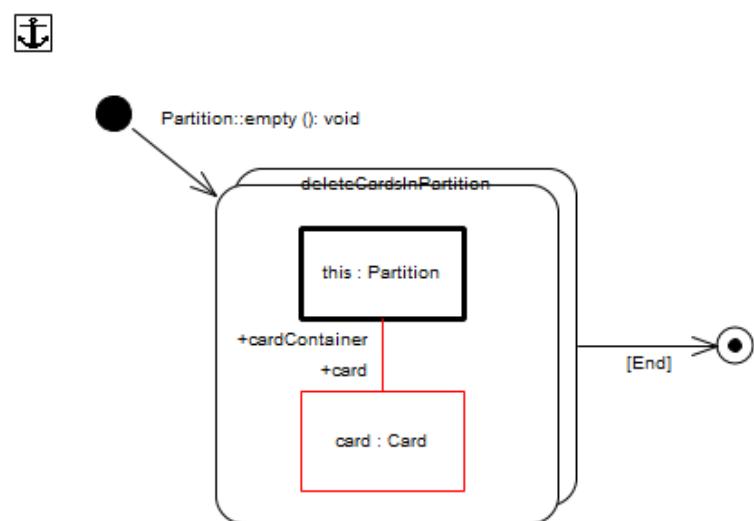


Figure 6.4: Completed `empty` story pattern

- ▶ Done! You've now learnt that in order to create a repeating action, all you need to do is change a standard story node into a `for each` node, and use appropriate *edge guards*.
- ▶ As always, save and build your metamodel. Inspect Fig. 6.6 and Fig. 6.7 to see how this SDM is implemented textually.
- ▶ Although the Learning Box GUI does not have an explicit action that invokes this SDM, feel free to extend it and see your SDM in action!

## 6.2 Implementing empty

- To initialize your new control flow, you can once again take advantage of eMoflon's auto completion. Inside the `empty` declaration, press **Ctrl + spacebar** and select `forEach` from the menu (Fig. 6.5).

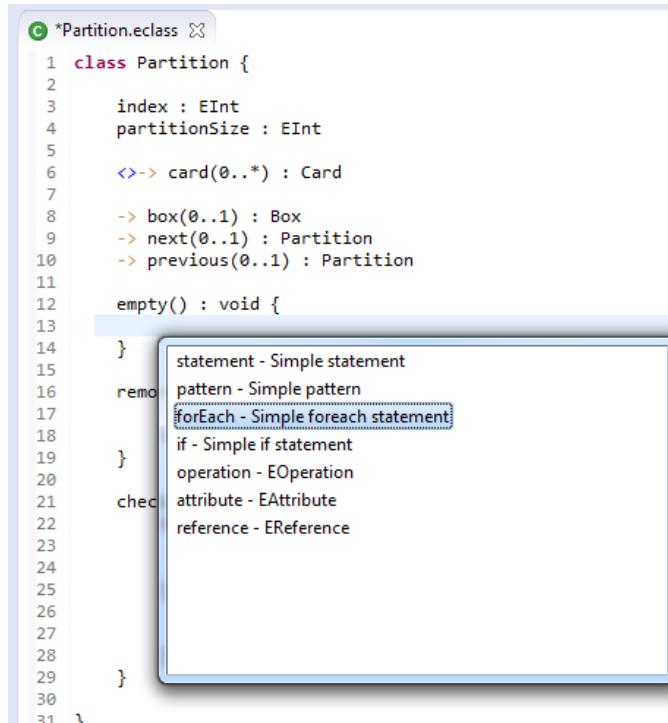


Figure 6.5: eMoflon's auto completion

- Create a single pattern, `deleteCardsInPartition`. Remove the suggested second pattern as you only need to complete the deletion – no extra steps are required in this simple case!
- Your activity should now resemble Fig. 6.6.

```
--
12 empty() : void {
13 forEach [deleteCardsInPartition]
14 return
15 }
16
```

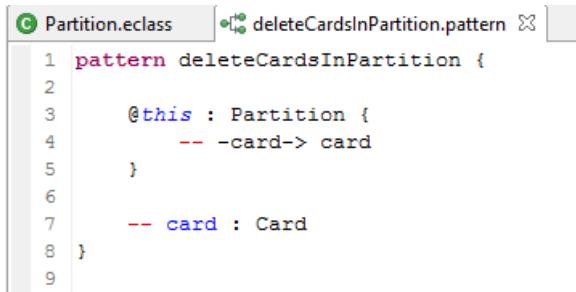
Figure 6.6: Control flow for `partition.empty()`

- ▶ While similar to `removeCard` (Fig. ??), this new pattern will go one step further by requesting a full destruction of `card`, instead of just deleting the link between the object variables. This means that in addition to destroying the link between the partition and `card`, we need to destroy the object variable `card` as well.
- ▶ Create a `@this` object variable, and delete its link to `card` via

`-- -card-> card`

Then create another object variable `card`, deleting it by prefixing its name with the same ‘`--`’ operator.

- ▶ Your pattern should now resemble Fig. 6.7.



```
C Partition.eclass deleteCardsInPartition.pattern X
1 pattern deleteCardsInPartition {
2
3 @this : Partition {
4 -- -card-> card
5 }
6
7 -- card : Card
8 }
9
```

Figure 6.7: Destroying both `card` and the link variable

- ▶ That’s it! Look at you go... you’re just speeding through these SDMs now! To see how `empty` is specified in the visual syntax, review Fig. 6.4 from the previous section.
- ▶ Although the Learning Box GUI does not have an explicit action that invokes this SDM, feel free to extend it and see your SDM in action!

## 7 Inverting a card

This next SDM *inverts* a card by swapping its back and face values (Fig. 7.1). This therefore “turns a card around” in the learning box. This action makes sense if a user wants to try learning, for example, the definition of a word in the other (target) language. Instead of guessing the definition of every word when presented with the term, perhaps they would like to guess the term when presented with the definition. This method doesn’t need to accept any parameters – it’ll use a bound `this` object variable.

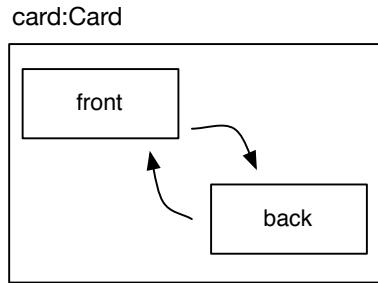


Figure 7.1: Inverting the attributes of a Card

Something new that we’ll use in this SDM are *assignments* to set the attributes of a `temp` object variable with `card`, then again to actually swap the `card` values. An assignment is simply an attribute constraint<sup>18</sup> with a ‘`::=`’ operator. Though it may be slightly confusing to refer to an assignment as a constraint, if you think about it, *everything* can be considered as a constraint that must be fulfilled using different strategies.

With `invert`, a successful match is achieved not by searching as you would with a comparison (`==`, `>`, `<`, `...`), but by *performing* the above assignment. If the assignment cannot be completed, the match is invalid. Similarly, non-context elements (set to create or destroy) can be viewed as structural constraints that are fulfilled when the corresponding element is created or destroyed. A constraint is therefore a unifying concept similar to “everything is an object” from OO, and “everything is a model” from metamodelling. If you’re interested in why *unification* is considered cool, check out [?].

*Assignments*

<sup>18</sup>Which we first encountered in `check`

## 7.1 Implementing invert

- If you've completed all the work so far, you've got to be *really* good at SDMs now. Model the simple story diagram depicted in Fig. 7.2.

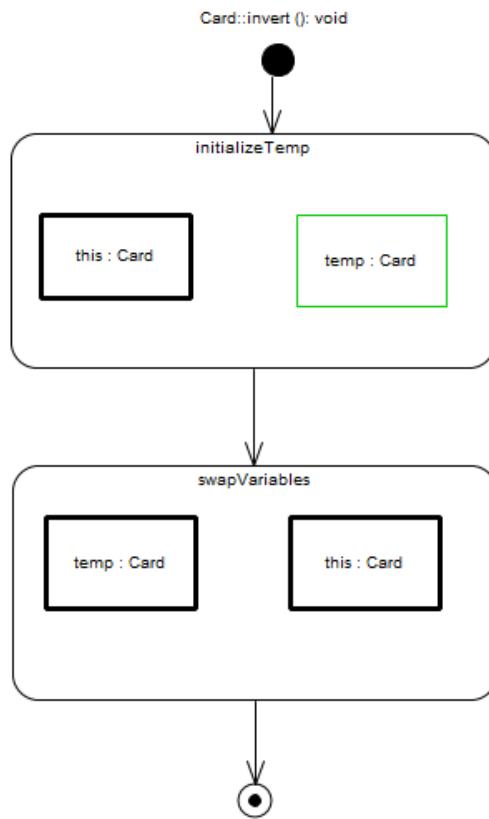


Figure 7.2: Imperative control layer for inverting a card

- Note that the binding operator on the first `temp` object variable is set to `create` (thus the green border). This means that we actually *create* a new object, and do not pattern match to an existing one in our model.
- This activity will need four assignment constraints - two in `initializeTemp` (to store the “opposite” values), and two in `swapVariables` (to switch the values). Create your first assignment constraint by going to the created `temp` card and using the ‘`:=`’ operator to set the `temp.back` value to `this.face` (Fig. 7.3).

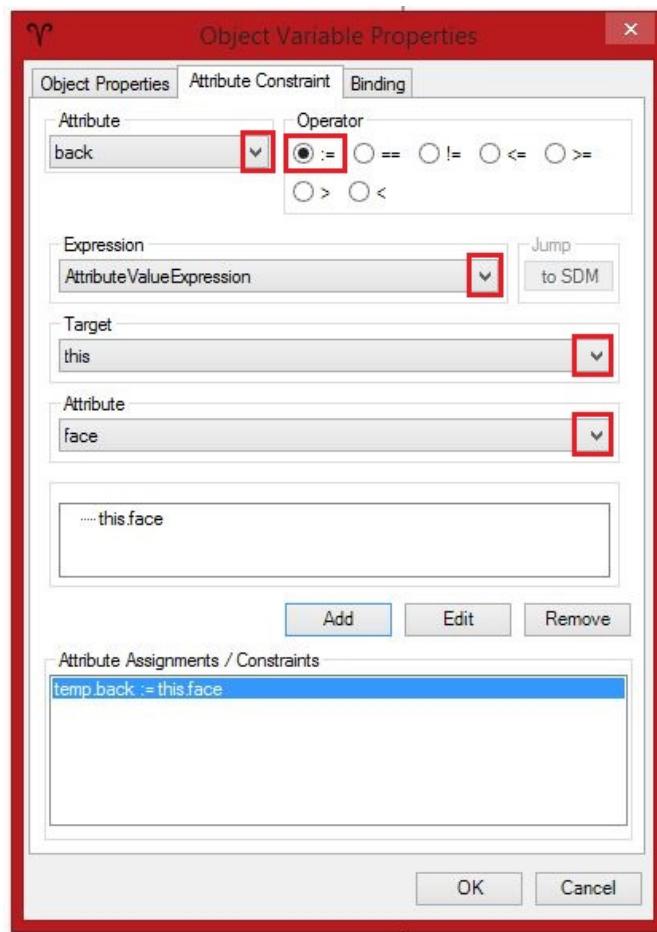


Figure 7.3: Store the `back` and `face` values of the card in `temp`

- Complete the SDM with the remaining constraints according to Fig. 7.4 below.

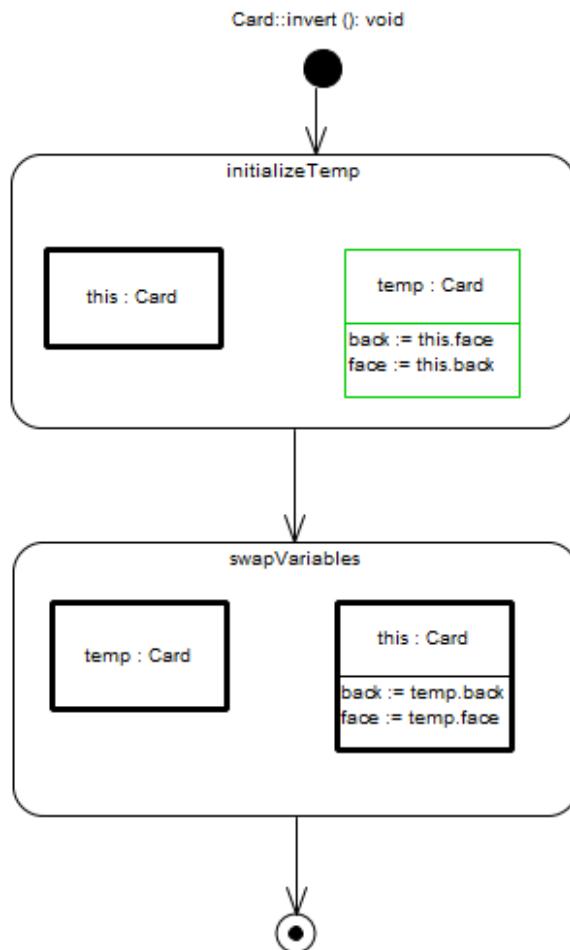


Figure 7.4: Swap back and face of the card

- Believe or not, that's it! Check out how this method is implemented in the textual syntax by reviewing Fig. 7.7 in the next section. You don't *have* to export and build to Eclipse, but it's always nice to confirm your work is error free.

▷ [Next](#)

## 7.2 Implementing invert

- ▶ You're no longer an SDM beginner, so create a simple control flow with two patterns for `card.invert()` named `initializeTemp` and `swapVariable`. First make sure they're presented in the right order, then confirm that you've included a return statement (Fig. 7.5).

```

8 invert() : Card {
9 [createTempCard]
10 [swapVariables]
11 return @this
12 }

```

Figure 7.5: Control flow for `card.invert`

- ▶ Create the `this` and `temp` `Card` variables in each pattern until your workspace resembles Fig. 7.6.

```

createTempCard.pattern
1 pattern createTempCard {
2
3 @this : Card
4
5 ++ temp : Card
6
7 }

swapVariables.pattern
1 pattern swapVariables {
2
3 @temp : Card
4
5 @this : Card
6
7 }

```

Figure 7.6: Swapping the `card` values

- ▶ Notice that the binding operator on `temp` in `initializeTemp` is set to create. This is so that we actually *create* a new object, and do not pattern match to an existing one in the model. It won't be persisted in the model afterwards which truly makes this a *temporary* variable.
- ▶ Next, we need to declare four assignments within the `temp` and `card` scopes, the first pair to assign the values to `temp`, and the latter to actually switch the values. Your workspace should now resemble Fig. 7.7.

```
createTempCard.pattern ✘
1 pattern createTempCard {
2
3 @this : Card
4
5 ++ temp : Card {
6 temp.face := this.face
7 temp.back := this.back
8 }
9
10 <
swapVariables.pattern ✘
1 pattern swapVariables {
2
3 @temp : Card
4
5 @this : Card {
6 this.face := temp.back
7 this.back := temp.face
8 }
9
10 }
```

Figure 7.7: Swapping the card values

- Believe it or not, that's it! We recommend building at this point to confirm you have made no mistakes. To see this method in the visual syntax, review Fig. 7.4 in the previous section.

## Inversion review

Before we start the next SDM, let's quickly review one point. Have you considered why the `temp` object variable is bound in the second pattern for `invert`, (`swap variables`), but not where it's first defined in `initialize temp`?<sup>19</sup> This is a new case for bound variables that we haven't treated yet!

Until now, we have seen object variables that can be bound to (1) an argument of the method (set when the method is invoked), or (2) the current object (`this`) whose method is invoked. In both cases, the object to be matched is completely determined by the context of the method before the pattern matcher starts. This means that it does not need to be determined or found by the pattern matcher.

Setting `temp` as bound in `Swap variables` is a third case in which an object variable is bound to a value determined in a *previous* activity node without using a special expression type. In this SDM, this means `temp` will be bound to the value determined for a variable of the same name in the previous node, `Initialize temp`. This binding feature enables you to refer to previous matches for object variables in the preceding control flow.

On a separate note, you're just over halfway through completing this part of the eMoflon handbook, so give your brain a small break. Take a walk, pour yourself another coffee, and check out one of my favourite jokes:

How do you wake up Lady Gaga?

Poke her face!

---

<sup>19</sup>See Fig. 7.4 (Visual) or Fig. 7.7 (Textual)

## 8 Growing the box

Ok, back to business. In this SDM, we shall explicitly specify how our learning box is to be built up. We create a specific pattern that will append new partition elements to the end of a Box that follow our established movement rules (Fig. 1.1). This means the new partition will become the `next` reference of the current last partition, and its `previous` reference must be connected to the first partition in the box (Fig. 8.1).

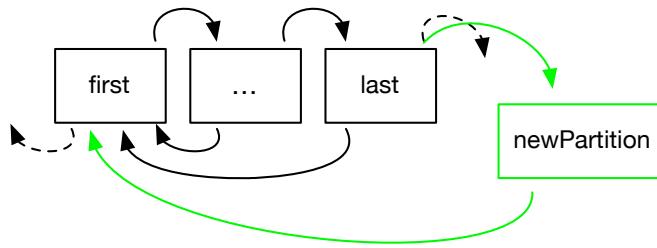


Figure 8.1: Growing a box by inserting a new partition

SDMs provide a declarative means of identifying specific partitions via *Negative Application Conditions*, simply referred to as NACs.<sup>20</sup> NACs express *NAC* structures that are forbidden to exist before applying a transformation rule. In this SDM, the NAC will be an object variable that must not be assigned a value during pattern matching. In the theory of algebraic graph transformations [?], NACs can be arbitrarily complex graphs that are much more general and powerful than what we currently support in our implementation,<sup>21</sup> namely only single negative elements (object or link variables).

As depicted in Fig. 8.1, to create an appropriate NAC that constrains possible matches, we'll need to check to see if the currently matched pattern can be extended to include the negative elements. Suppose the current potential last partition has a `nextPartition`. This means it is *not* the absolute last partition, and so the match becomes invalid. We only want to insert a new partition when the `nextPartition` of the current potential last partition is null. Similarly, if the current potential first partition has a `previousPartition`, the match is invalid. The complete match is therefore made unique through NACs and thus becomes *deterministic* by construction. In other words, if you *grow* the box with this method, there will always be exactly one first and one last partition of the box.

Of course, to complete this method we still need to determine the size of the

---

<sup>20</sup>Pronounced \'*nak*\'

<sup>21</sup>To be precise, in CodeGen2 from Fujaba

new partition. Since the size must be calculated depending on the rest of the partitions currently in the box (partitions usually get bigger) we'll need to call a helper method, `determineNextSize` via a *MethodCallExpression*. *MethodCallExpression* As the name suggests, it is designed to access any method defined in *any* class in the current project.

Due to the algorithmic and non-structural nature of `determineNextSize`, it will be easier to implement this method via a Java *injection*, rather than an SDM. We've already declared this method in our metamodel, so its signature will be available for editing in `BoxImpl.java`.

- ▶ Open “gen/LearningBoxLanguage.impl/BoxImpl.java.” Scroll to the method declaration, and replace the contents with the code in Fig. 8.2. Remember not to remove the first comment, which is necessary to indicate that the code is handwritten and needs to be extracted automatically as an injection. Please do not copy and paste the following code – the copying process from your pdf viewer to the Eclipse IDE will likely add invisible characters to the code that eMoflon is unable to handle.

```
public int determineNextSize() {
 // [user code injected with eMoflon]
 return getContainedPartition().size()*10;
}
```

Figure 8.2: Implementation of `removeCard`

- ▶ Save the file, then right-click on it, either in the package explorer or in the editor window, and choose “eMoflon/ Create/Update Injection for class” from the context menu.
- ▶ Confirm the update in the new `BoxImpl.inject` file's partial class. `determineNextSize` is now ready to be used by your metamodel!

## 8.1 Implementing grow

- ▶ Start by creating the simple story pattern depicted in Fig. 8.3. This matches the box and *any* two partitions.<sup>22</sup>

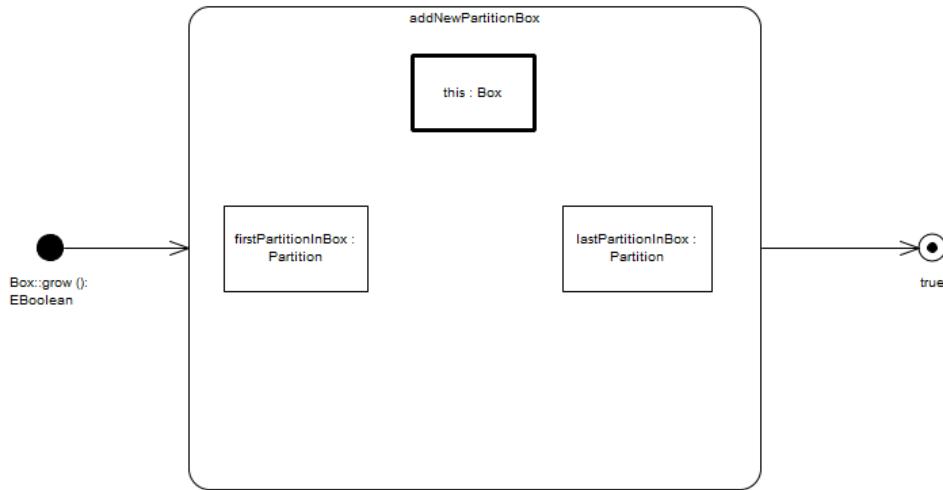


Figure 8.3: Context elements for SDM

- ▶ To create an appropriate NAC to constrain the possible matches for `lastPartitionInBox`, create a new `Partition` object variable `nextPartition` and set its *binding semantics* to `negative` (Fig. 8.4). The object variable should now be visualised as being cancelled or struck out.
- ▶ Now, quick link `nextPartition` to `lastPartitionInBox`. Be sure to choose the link type carefully! The `nextPartition` should play the role of `next` with respect to `lastPartitionInBox`. This combination (the negative binding and reference) tells the pattern matcher that if the (assumed) last partition has an element connected via its `next` reference, the current match is invalid.
- ▶ Great work – the first NAC is complete! In a similar fashion, create the NAC for `firstPartitionInBox`. Name the negative element `previousPartition`, and again, be sure to double-check the link variable.
- ▶ Finally, complete the pattern so that it closely resembles Fig. 8.5.

---

<sup>22</sup>Remember, the *pattern matcher* is non-deterministic.

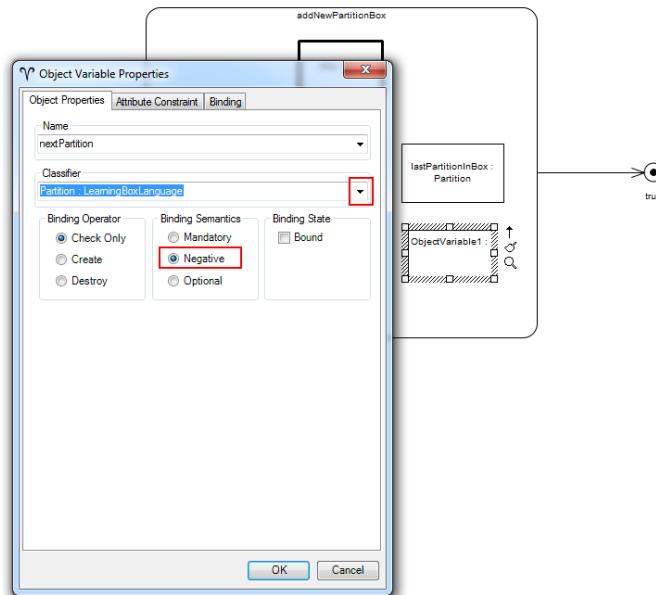


Figure 8.4: Adding a negative element

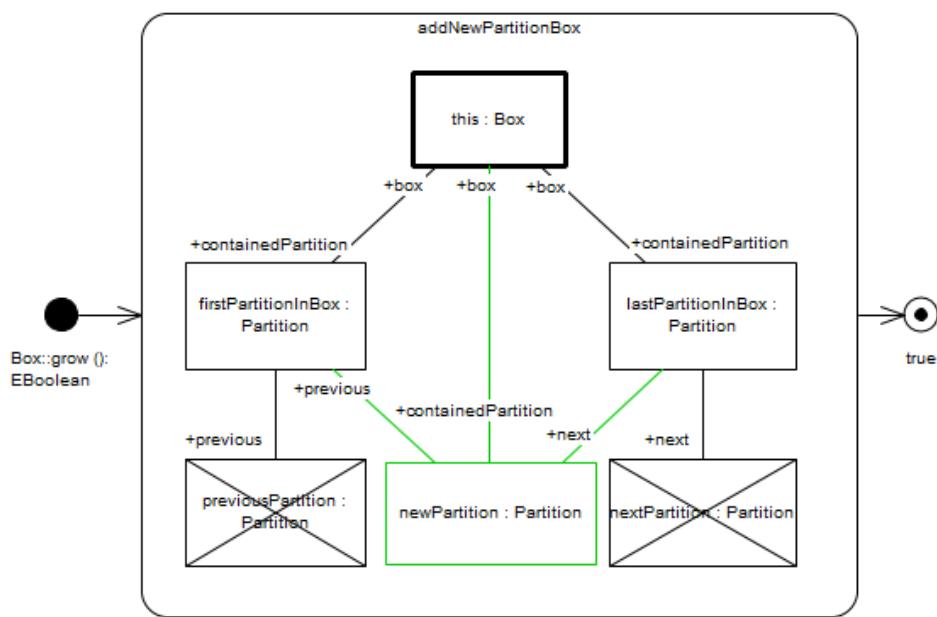


Figure 8.5: Determining the first and last partitions with NACs

- ▶ Notice how the created partition `newPartition` is ‘hung’ into the box. It becomes the next partition of the current `last` partition, and its previous partition is automatically set to the first partition in the box (as dictated by the rules set in Fig. 1.1). In other words, the new partition is appended onto the current set of partitions.
- ▶ In order to complete `grow`, we need to set the size of the `newPartition`. Given that the new size is calculated via the helper function `determineNextSize`, we need to use a *MethodCallExpression*. Go ahead and invoke the corresponding dialogue, activate the assignment (`:=`) operator, and match your values to Fig. 8.6.

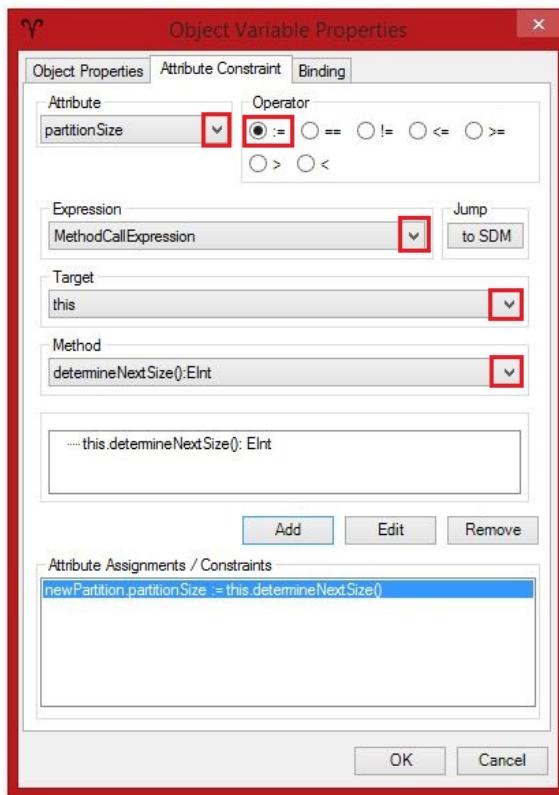


Figure 8.6: Invoking a method via a `MethodCallExpression`

- ▶ Since `determineNextSize` doesn’t require any parameters, you can ignore the `Parameter Values` field this time.
- ▶ If you’ve done everything right, your SDM should now closely resemble Fig. 8.7. As usual, try to export, generate code, and inspect the method implementation in Eclipse.

- That's it – the `grow` SDM is complete! This was probably the most challenging SDM to build so give yourself a solid pat on the back. If you found it easy, well then ... I guess I'm doing my job correctly. To see how this is done in the textual syntax, review Fig. 8.13.

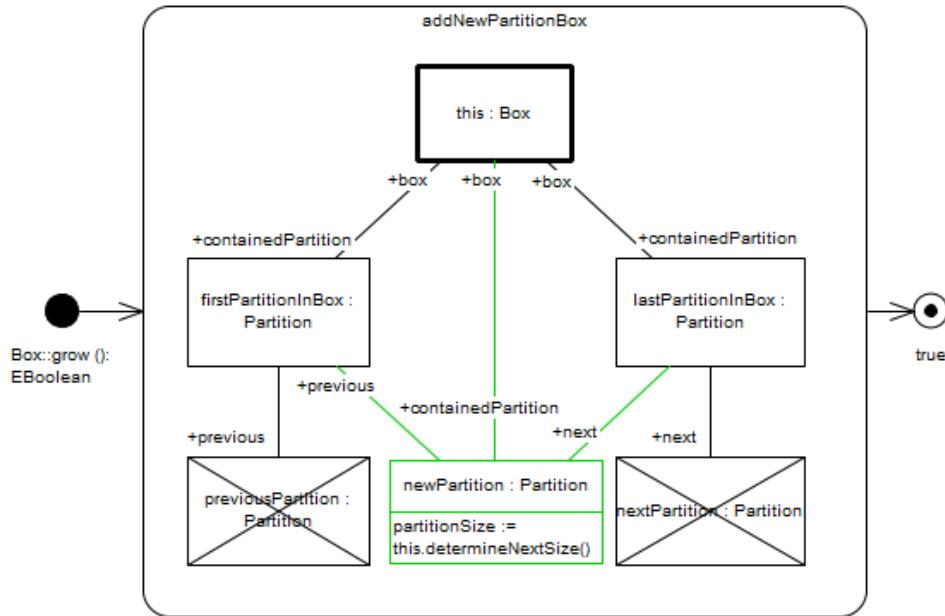


Figure 8.7: Complete SDM for `Box::grow`

## 8.2 Implementing grow

- In `box.grow()`, create a simple control flow with one story pattern (Fig. 8.8).

```

11 grow() : EBoolean {
12 [addNewPartitionBox]
13 return true
14 }

```

Figure 8.8: Basic control flow to grow box

- Create and open the new pattern. You'll want it to match the invoking box with *any* two partitions, so create a bound `this` box, and the free variables `firstPartitionInBox` and `lastPartitionInBox`. You'll also need an object variable (set to create) to represent the new partition. The skeleton of your pattern should now resemble (Fig. 8.9).

```

1 pattern addNewPartitionBox {
2
3 @this : Box
4
5 firstPartitionInBox : Partition
6
7 lastPartitionInBox : Partition
8
9 ++ newPartition : Partition
10
11 }

```

Figure 8.9: The `addNewPartitionBox` skeleton

- Next, we need to create an appropriate *NAC* which will constrain the possible choices for `lastPartitionInBox`. Create a negative `nextPartition` object variable by using the negation ‘!’ operator.
- Now add ‘`-next-> nextPartition`’ to `lastPartition`'s scope. This attempts to establish a `next` link from the last partition. Next, add ‘`++ -next-> newPartition`’ (Fig. 8.10). This constraint will only be fulfilled if the NAC fails, and it establishes the `newPartition` as the final partition.

```

C Box.eclass | addNewPartitionBox.pattern |
1 pattern addNewPartitionBox {
2
3 @this : Box
4
5 firstPartitionInBox : Partition
6
7 ! previousPartition : Partition
8
9 lastPartitionInBox : Partition {
10 -next-> nextPartition
11 ++ -next-> newPartition
12 }
13
14 ! nextPartition : Partition
15
16 ++ newPartition : Partition
17 }
```

Figure 8.10: Creating the first NAC

- ▶ In a similar fashion, create a second NAC, `previousPartition`, for `firstPartitionInBox`. No new references have to be created here, so all you need to establish is the link connecting `firstPartitionInBox` to the negative element, `previousPartition` (Fig. 8.11).

```

C Box.eclass | addNewPartitionBox.pattern |
1 pattern addNewPartitionBox {
2
3 @this : Box
4
5 firstPartitionInBox : Partition {
6 -previous-> previousPartition
7 }
8
9 ! previousPartition : Partition
10
11 lastPartitionInBox : Partition {
12 -next-> nextPartition
13 ++ -next-> newPartition
14 }
15
16 ! nextPartition : Partition
17
18 ++ newPartition : Partition
19 }
```

Figure 8.11: Pattern with both NACs

- ▶ Now edit `@this` with appropriate link variables to the first and last partitions. Try using auto-completion here for the reference names!

- The next step is to establish the `box` and `previous` references in our newly created object variable, `newPartition`. While you could of course write ‘`++ -box-> this`’, any link variables established here are automatically set to ‘green,’ and do not need to be explicitly set with an `++` operator. This is because you cannot connect a ‘black’ link to a ‘green’ node.<sup>23</sup> In other words, a ‘green’ object variable has a global effect on all link variables declared in its scope.
- Your pattern should now resemble Fig 8.12.

```

1 pattern addNewPartitionBox {
2
3 @this : Box {
4 -containedPartition-> lastPartitionInBox
5 -containedPartition-> firstPartitionInBox
6 }
7
8 firstPartitionInBox : Partition {
9 -previous-> previousPartition
10 }
11
12 ! previousPartition : Partition
13
14 lastPartitionInBox : Partition {
15 -next-> nextPartition
16 ++ -next-> newPartition
17 }
18
19 ! nextPartition : Partition
20
21 ++ newPartition : Partition {
22 -box-> this
23 -previous-> firstPartitionInBox
24 }
25 }

```

Figure 8.12: Pattern with deterministic choice of first and last partitions

- We’re not *quite* done yet - our newest partition doesn’t yet have a size. This means that not only do we need to make another attribute constraint to set its value, but `newPartition` needs to directly invoke a method in order to get the correct value. You can do this via a *MethodCallExpression*. The structure of this expression is similar to Java where:

```
MethodCallExpression := (object_variable_expression | parameter_expression) '.' ID '(' argument_list ')'
```

- We’ve encountered both of these expression types already – ‘@’ in

---

<sup>23</sup>Remember that a rule actually consists of two graphs:  $r = (L, R)$ . A green node only belongs to  $R$  and not to  $L$ , while a black link is in  $L$  and in  $R$ . If the target of the link is only in  $R$ , however,  $L$  would have a link with an undefined target. This is not allowed ( $L$  is not a graph).

`removeCard` and ‘\$’ in `check`. The latter doesn’t apply to this pattern, so write:

```
@this.determineNextSize()
```

- Your workspace should now resemble Fig. 8.13.



```

C Box.eclasse addNewPartitionBox.pattern X
1 pattern addNewPartitionBox {
2
3 @this : Box {
4 -containedPartition-> lastPartitionInBox
5 -containedPartition-> firstPartitionInBox
6 }
7
8 firstPartitionInBox : Partition {
9 -previous-> previousPartition
10 }
11
12 ! previousPartition : Partition
13
14 lastPartitionInBox : Partition {
15 -next-> nextPartition
16 ++ -next-> newPartition
17 }
18
19 ! nextPartition : Partition
20
21 ++ newPartition : Partition {
22 newPartition.partitionSize := @this.determineNextSize()
23
24 -box-> this
25 -previous-> firstPartitionInBox
26 }
27 }
```

Figure 8.13: Complete pattern for adding a new partition to Box

- Now we’re done! While NACs may be difficult to understand at first, as you can see, they’re not hard to implement, and can be used in a wide variety of applications. To see how this method is implemented in the visual syntax, check out Fig. 8.7 in the previous section.

## 9 Conditional branching

When working with SDMs, you'll often find yourself needing to decide which statement(s) to execute based on the return value of an arbitrary (black box) operation, as we saw in `check`. In our example so far, we have implemented these constructs via SDM *pattern matching*.

With eMoflon however, there is an alternate way to construct these black boxes. In fact, this feature is yet another way of integrating handwritten Java code with your SDM. We can invoke methods directly from an *if* statement. The only “rule” of this feature is that the method must return an `EBoolean` to indicate `Success` or `Failure`, corresponding to `true` or `false`, respectively. Any other types imply `Failure` if the return value of the method is `null`. It follows that void methods cannot be used for branching – an exception will be thrown during code generation (if you ignored the validation error).

Unfortunately, you can't simply invoke a method from a standard activity node. Instead, you must use a new type of activity node, a *statement node*. Statement nodes can be used to invoke methods and provide a means of invoking libraries and arbitrary Java code from SDMs. Please note that we do not differentiate at this point between methods that are implemented by hand or via an SDM. Thus, statement nodes can of course be used to invoke other SDMs via a *MethodCallExpression*. Most importantly, statement nodes enable *recursion*, as the current SDM can be invoked on `this` with appropriate new arguments. In essence, this type of node is only used to guarantee a specific action between *activity nodes*, and does not extend the current set of matched variables. They can however, be used as a conditional by branching on whatever value the method returns.

*Statement Node*

Let's reconsider `grow`, the method we just completed that adds a new partition to our box. Reviewing either Fig. 8.7 (Visual) or Fig. 8.13 (Textual), the current pattern assumes there are already at least two partitions in `box` (the `firstPartitionInBox` and `lastPartitionInBox`). What would happen if `box` had only one, or even no partitions at all? The pattern would *never* find a match!

To fix this problem, let's modify `grow` so that if the original match fails, we initialize two new partitions (the first and last), but *only* if it failed due to the `box` being completely empty. In other words, if `box` has e.g., only one partition (an invalid state that cannot be reached by growing from zero partitions), it is considered invalid and no longer be grown.

---

### 9.1 A short note on the initializeBox method

Pretend you've just updated the control flow in your `grow` SDM, and haven't specified `initializeBox` yet. After saving and building, you will be able to see the changes in `BoxImpl.java`, the source file containing the generated code. In fact, open this file now and navigate to `grow`, which starts at (approximately) line 207 (Fig. 9.1). This is the generated *statement node* code and, as you can see, all it does is invoke your method and branch based on its result.

```

} else {
 // initialize
 if (BoxImpl.pattern_Box_2_3_expressionFB(this)) {
 return BoxImpl.pattern_Box_2_4_expressionF();
 } else {
 return BoxImpl.pattern_Box_2_5_expressionF();
 }
}

```

Figure 9.1: Code generated for branching with a statement node

Go to `initializeBox`, it should look like Fig. 9.2.

```

△256@ public boolean initializeBox() {
257 // [user code injected with eMoflon]
258
259 // TODO: implement this method here but do not remove the injection marker
260 throw new UnsupportedOperationException();
261
262 ...

```

Figure 9.2: The `initializeBox` declaration

You have the choice of either implementing the method by hand here in Java as an injection, or you can return to the metamodel and implement it there as an SDM. The statement node will work just fine in both cases.

Using Java and injections makes sense if the method is non-structural, but seeing as we must check to see if there is a single partition, then create the first two partitions of the box if it succeeds, `initializeBox` is actually quite structural and can be described beautifully as a pattern. This is why we opted to specify it as an SDM.

## 9.2 Branching with statement nodes

- ▶ Currently, there is no method to help us initialize `box` from its pristine state (no partitions). Create one by editing your metamodel (the `LearningBoxLanguage` diagram) and invoking the `Operations` dialogue by first selecting `Box`, then pressing `F10`.<sup>24</sup>
- ▶ Name the new method `initializeBox` and, recalling the one rule of conditional branching, set its return type to `EBoolean`.
- ▶ Save and close the dialogue, then re-open the `grow` SDM and *Quick Create* a new story node from `addNewPartition`.
- ▶ This will be the node we'll use to invoke our helper method. Double click the node to invoke its properties editor and switch the `Type` to a `StatementNode`. Name it `initialize` (Fig. 9.3a).
- ▶ Before closing the dialogue, switch to the `Statement` tab, and create a `MethodCallExpression` to invoke your newest method (Fig. 9.3b). We want to access the `Box` object (`this`) and its `initializeBox` method. It doesn't require any parameters, so leave the values field empty.

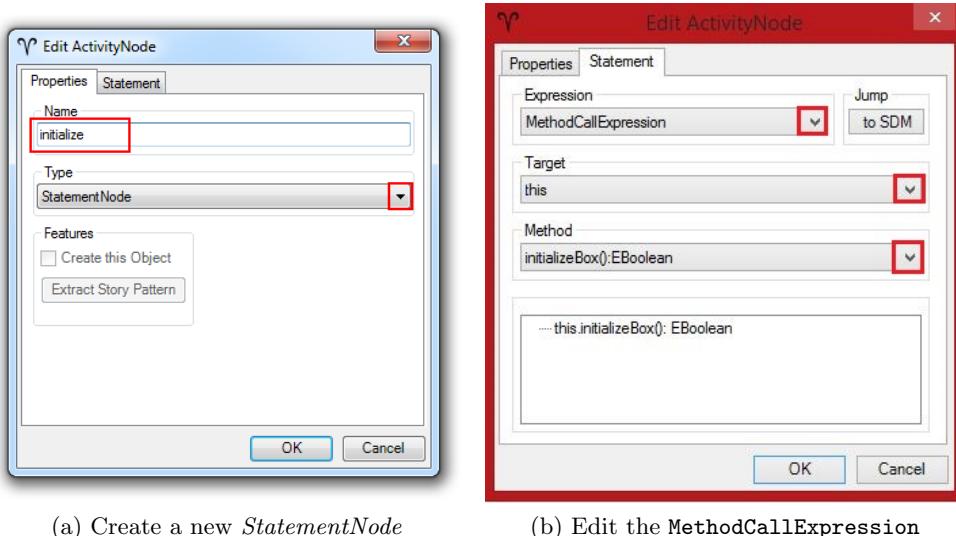


Figure 9.3

<sup>24</sup>To review creating new operations, review Section 2.6 of Part II

- ▶ Now we need to update the edge guards stemming from `addNewPartitionInBox`. Given that we only want to call `initializeBox` if the pattern fails, change the edge guard leading to your statement node to `Failure`. Similarly, update the edge guard returning `true` to `Success`.
- ▶ Finally, attach two stop nodes – `true` and `false` – along with their appropriate edge guards from `initialize`. These indicate that if the method execution worked, the box could be initialized. If it failed however, `box` was in an invalid state (by e.g., having only one partition) and returns `false`. Overall, the new additions to `box.grow()` should resemble Fig. 9.4.

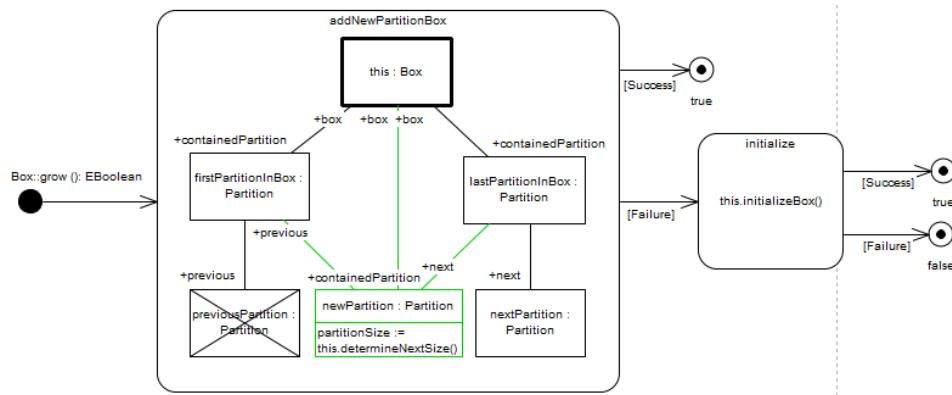


Figure 9.4: Extending `grow` with a *MethodCallExpression*

- ▶ To review our work up to this point, we have declared `initializeBox` and invoked it from a statement node. We have yet to actually specify the method however. Double-click the anchor to return to the main diagram and create a new SDM for `initializeBox`.
- ▶ Create a normal activity node named `buildPartitions` with the pattern depicted in Fig. 9.5.

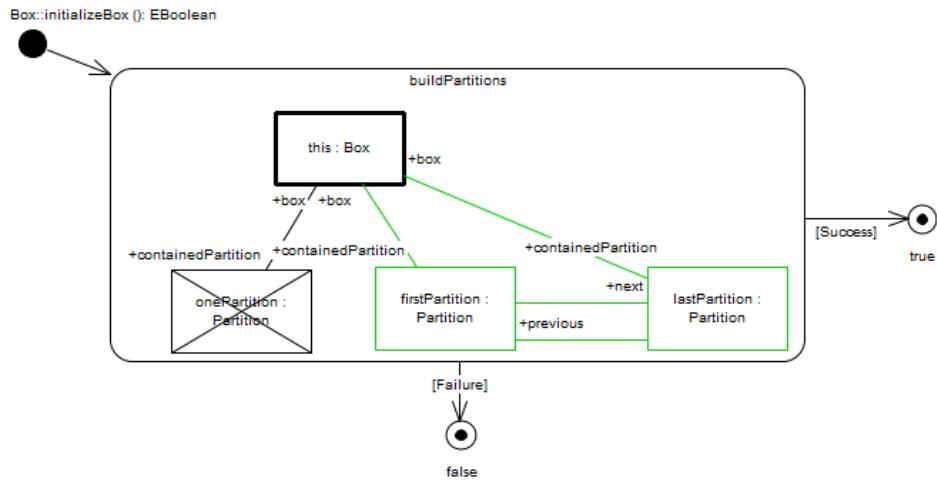


Figure 9.5: Complete SDM

- ▶ The NAC used here is only fulfilled if the box has absolutely no partitions, i.e., is in a pristine state and can be initialized. In other words, if `grow` is used for an empty box, it initializes the box for the first time and grows it after that, ensuring that the box is always in a valid state.
- ▶ You're finished! Save, validate, and build your metamodel, then check out how this is done in the textual syntax in Fig. 9.6 and Fig. 9.7.

▷ [Next](#)

### 9.3 Branching with statement nodes

- Before doing anything else, let's declare the method that will insert two new partitions into `box` when the pattern in `grow` fails. Open `Box.eclass` and add the following signature:

```
initializeBox() : EBoolean
```

- Now modify `Box.grow()` by adding a nested *if/else* construct, with `[addNewPartitionToBox]` as the first conditional, and a *statement node* to invoke `initializeBox` if it fails. *Statement nodes* are specified via:

```
'<' method_call '>'
```

`grow` should now resemble Fig. 9.6.

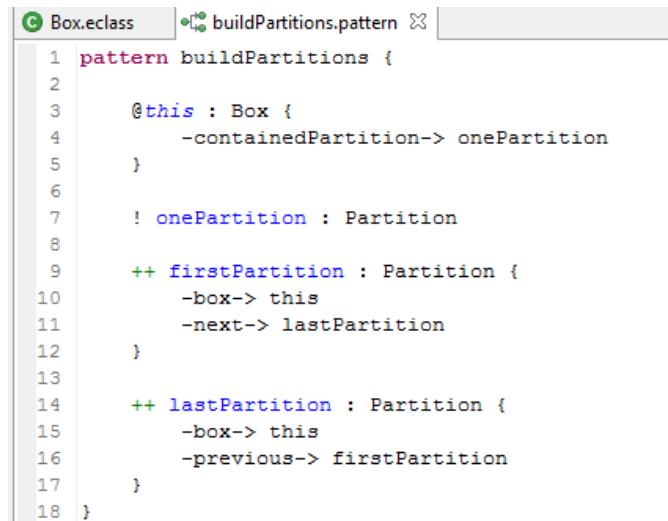
```

10 grow() : EBoolean {
11 if [addNewPartitionToBox] {
12 return true
13 } else {
14 if <@this.initializeBox()> {
15 return true
16 }
17 else {
18 return false
19 }
20 }
21 }
22
23 initializeBox() : EBoolean {
24 if [buildPartitions] {
25 return true
26 }
27 else {
28 return false
29 }
30 }

```

Figure 9.6: Extending `grow` with a *statement node*

- Next we have to specify our newest method. Create a new pattern called `buildPartitions` in its scope. Complete the pattern as illustrated in Fig. 9.7.
- As you can see, we have created a NAC that can only be fulfilled if the box has absolutely no partitions at all. This means that if a box is completely empty, it will be initialized for the first time with two partitions (according to `buildPartitions`), and is guaranteed to remain in a valid state via `grow`.



The screenshot shows a code editor window with the title bar "Box.eclass" and a tab labeled "buildPartitions.pattern". The main content of the editor is a textual representation of a Node Accepting Condition (NAC). The code is as follows:

```
1 pattern buildPartitions {
2
3 @this : Box {
4 -containedPartition-> onePartition
5 }
6
7 ! onePartition : Partition
8
9 ++ firstPartition : Partition {
10 -box-> this
11 -next-> lastPartition
12 }
13
14 ++ lastPartition : Partition {
15 -box-> this
16 -previous-> firstPartition
17 }
18 }
```

Figure 9.7: NAC initializing an empty box

- That's it! Save and build your metamodel to make sure no errors exist. To see how this is depicted in the visual syntax, check out Fig. 9.4 and Fig. 9.5.

## 10 A string representation of our learning box

In the next SDM we shall create a string representation for all the contents in a single learning box. To accomplish this, we will have to iterate through every card, in every partition. The concept is similar to `Partition`'s `empty` method, except we'll need to create a nested *for each* loop (Fig. 10.1). Further still, we'll need to call a helper method to accumulate the contents of each card to a single string.

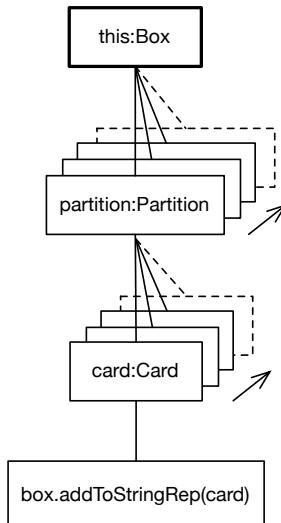


Figure 10.1: Nested *For Each* loops

As you can see, The first loop will match all partitions, while the second matches each card. Finally, a *statement node* is used to invoke the `addToStringRep` method. In contrast to how they were used for conditional branching in `grow`, this statement node will simply invoke a void method.

Unlike `initializeBox` however, this helper method is actually better specified as an injection so, analogously to how you implemented `determineNextSize` for `box.grow()`, quickly edit `BoxImpl.java` by replacing the default code for `addToStringRep` with that in Fig. 10.2. You can use Eclipse's built-in auto-completion to speed up this process. Save, create the injection file, and confirm the contents of `BoxImpl.inject`.

```
public void addToStringRep(Card card) {
 // [user code injected with eMoflon]
 StringBuilder sb = new StringBuilder();
 if (stringRep == null) {
 sb.append("BoxContent: [");
 } else {
 sb.append(stringRep);
 sb.append(", [");
 }
 sb.append(card.getFace());
 sb.append(", ");
 sb.append(card.getBack());
 sb.append("]");
 stringRep = sb.toString();
}
```

Figure 10.2: Implementation of `addToStringRep`

▷ Next [visual]  
▷ Next [textual]

### 10.1 Implementing `toString` for Box

- ▶ Visual SDMs support arbitrary nesting of *for each* story nodes via special guards. In Section 5.1 we used the `[end]` edge guard to terminate a loop. Now we'll use a new guard, the `[each time]` guard, to indicate *[each time]* control flow that is *nested* and executed for each match. Go ahead and create the SDM for `Box::toString` until it closely resembles Fig. 10.3.

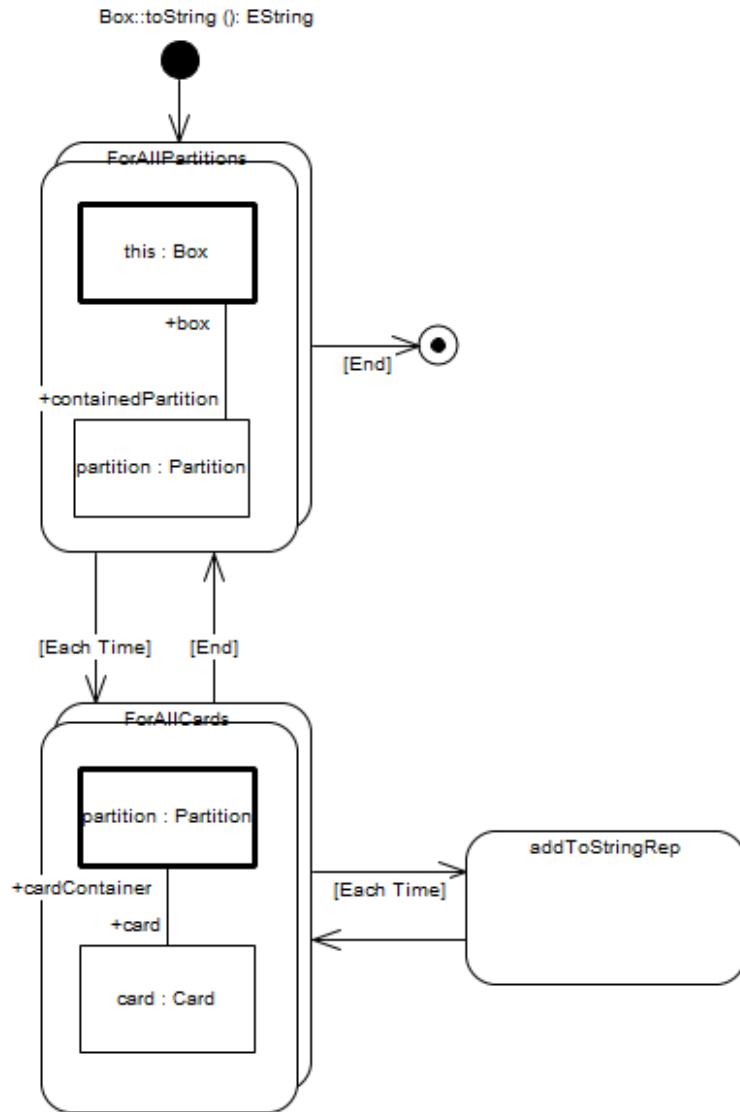


Figure 10.3: Control flow with nested loops

- ▶ Knowing `addToStringRep`'s default node type will not allow it to invoke our helper method, change it into a `StatementNode`. Then, analogously to how you established a `MethodCallExpression` for `grow`, have this node invoke `this.addToStringRep(Card)` (Fig. 10.4).
- ▶ You can see in the `Method` statement that a `Card` parameter is required. For this double click on the field below and enter the values like in Fig. 10.5. Now we included `card` so we may pass the object variable to the method.

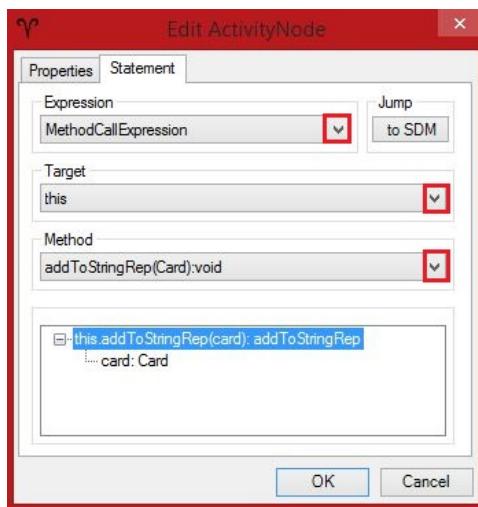


Figure 10.4: Add a *MethodCallExpression*

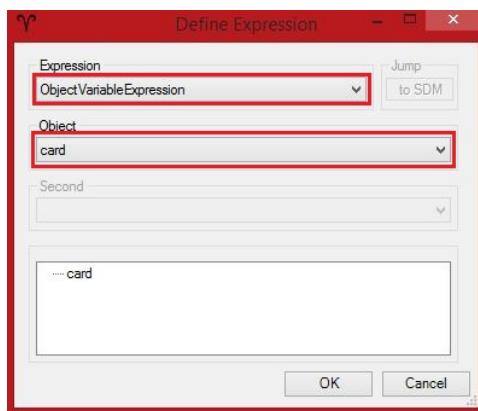


Figure 10.5: Add a parameter to the *MethodCallExpression*

- To complete the SDM, return the final string representation value of the box via an *AttributeValueExpression* in the stop node (Fig. 10.6). This is a new expression type we haven't encountered before. It simply binds the `stringRep` attribute of the box (`this`) to the return value in the stop node

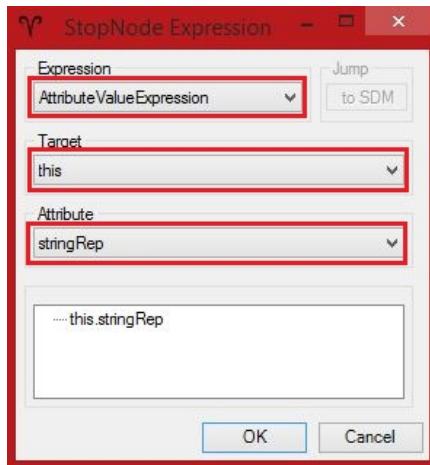


Figure 10.6: Specify a return value as an *AttributeValueExpression*

- Take some time to compare and reflect on the complete SDM as depicted in Fig. 10.7. The idea was to abstract from the actual text representation of the box and model the necessary traversal of the data structure. The helper method `addToStringRep` could, for example, build up something totally different.
- While modelling this SDM, we have seen that *for each* story nodes can be nested, and have used a *MethodCallExpression* to invoke a void helper method only for its side effects (building up the string representation of the box).
- As always, save, validate, and build your metamodel in Eclipse. To see how this is done in the textual syntax, check out the nested loops in Fig. 10.9 and each pattern in Fig. 10.10.

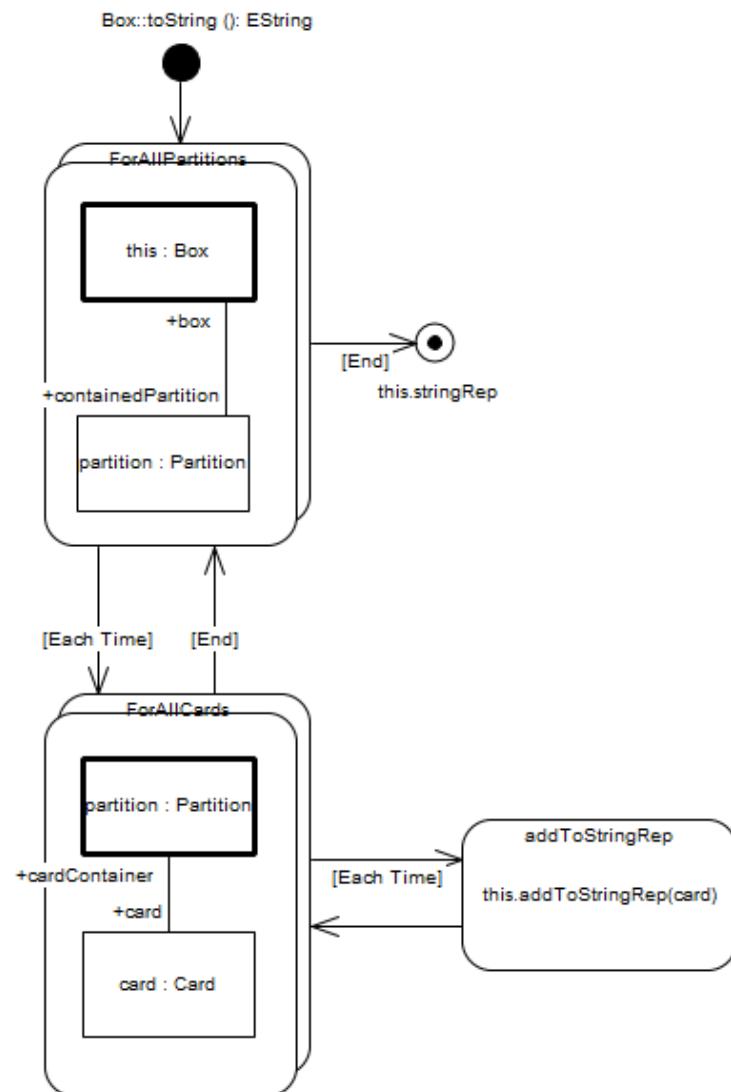


Figure 10.7: The complete SDM for `Box::toString`

▷ Next

## 10.2 Implementing `toString` for Box

- ▶ Closely following Fig. 10.1, create a nested `forEach` loop in `Box.toString()`. Don't worry about invoking `addToStringRep` yet – just make sure you return the `this.stringRep` attribute. It should resemble Fig. 10.8. `this.stringRep` is referred to as an *AttributeValueExpression* as it accesses an attribute value of an object variable (`this`) via a ‘.’ (dot) operator.

```

26 toString() : EString {
27 forEach [forAllPartitions] {
28 forEach [forAllCards] {
29 }
30 }
31 }
32 return this.stringRep
33 }

```

Figure 10.8: Control flow for `toString` with nested *for each* loops

- ▶ In order to invoke `addToStringRep`, we need a *statement node*. Remembering that statement nodes are enclosed in `< >`, write inside the `sSecond` loop:

```
<@this.addToStringRepCard(card)>
```

The correct `card` parameter will be matched by the `forAllCards` pattern. We'll establish this in a moment.

- ▶ The completed `toString` activity should now resemble Fig. 10.9.

```

15 toString() : EString {
16 forEach [forAllPartitions] {
17 forEach [forAllCards] {
18 <@this.addToStringRepCard(card)>
19 }
20 }
21 return this.stringRep
22 }

```

Figure 10.9: Using a *statement node* to invoke a void helper method

- ▶ Don't forget that the aim of the activity is to access every card in `Box`. The control flow terminates when the helper method has been called for all existing cards. This means the `forAllPartitions` and `forAllCards` patterns must match all cards in all partitions.

- Complete each pattern as depicted in Fig. 10.10. The `partition` in `forAllCards` is bound to the one matched in the current (first) loop iteration, but `card` is *not* bound as it must be newly matched each time.

The screenshot shows a code editor with two tabs. The top tab is titled 'Box.eclass' and contains the following code:

```
1 pattern forAllPartitions {
2
3 @this : Box {
4 -containedPartition-> partition
5 }
6
7 partition : Partition
8 }
```

The bottom tab is titled 'forAllCards.pattern' and contains the following code:

```
1 pattern forAllCards {
2
3 @partition : Partition {
4 -card-> card
5 }
6
7 card : Card
8 }
```

Figure 10.10: Box traversal patterns

- As crazy as it may seem, that's it! To see how this SDM is represented visually, check out Fig. 10.7.

## 11 Fast cards!

Congratulations, you're almost there! This is the last SDM needed before your Leitner's learning box is fully functional.

For very simple cards (i.e., words in a different language that are quite similar), it might be a bit annoying to have to answer these cards again and again in successive partitions. Such *fast* cards should somehow be marked as such and handled differently. If a fast card is correctly answered once, it should be immediately moved to the final partition in the box. This way, the card is practiced once, and only tested once more before finally being ejected from the box.

It makes sense for a **FastCard** to inherit from **Card**, so we'll extend the current object in our metamodel by a new **EClass** for fast cards, depicted below with a marker to show it behaves differently (Fig. 11.1).

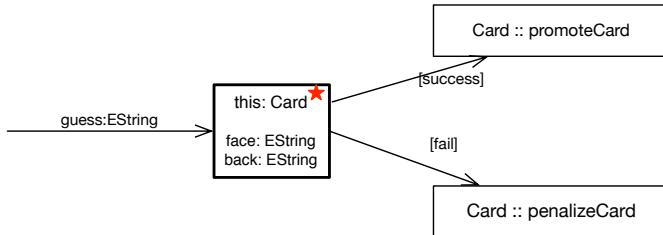


Figure 11.1: Checking a fast card against a guess

In addition to creating a new **EClass**, we also need to extend the existing **check** method to check for this special card type once a guess is determined to be correct. Now **check** needs to decide, based on the dynamic type<sup>25</sup> of **card**, if it needs to handle this special fast card. This can be expressed in SDMs with a *BindingExpression* (or just *Binding*). A binding can be *Binding* specified for a *bound* object variable and is the final case in which an object variable can be marked as being bound.

To refresh your memory, we have already learnt that a bound object variable is either (1) assigned to **this**, (2) a parameter of the method, or (3) a value determined in a preceding activity node. Bindings represent a fourth possibility of giving a manual binding for an object variable.

Finally, this new pattern faces a similar challenge as **grow**. A **FastCard** can't simply progress to the **next** partition. It must skip ahead to the absolute last partition in the box. This means yet another NAC is required to determine the last partition in a **Box**.

<sup>25</sup>In a statically typed language like Java, every object has a static type (determined at compile time) and a dynamic type (that can only be determined at runtime).

## 11.1 Implementing FastCards

- To introduce fast cards into your learning box, return to the metamodel diagram and create a new EClass, **FastCard**. Quick link to Card and choose **Create Inheritance** from the context menu. We only want to check the dynamic type of a tested card at runtime, which means we don't need to override anything. Therefore, when the **Overrides & Implementations** dialogue appears, make sure nothing is selected (Fig. 11.2). Your metamodel should then resemble Fig. 11.3.

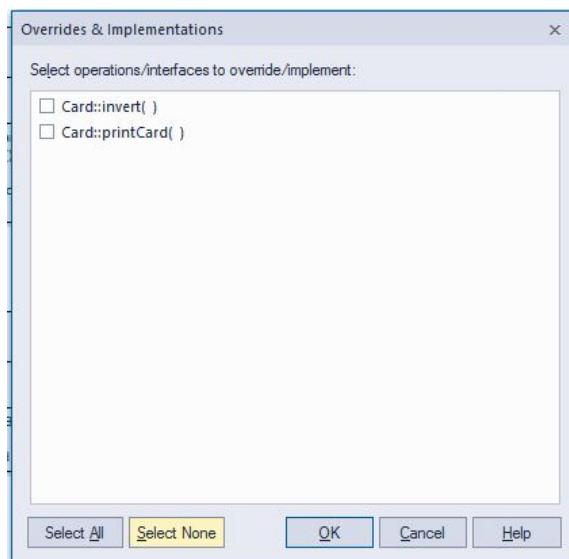


Figure 11.2: Selecting operations to override

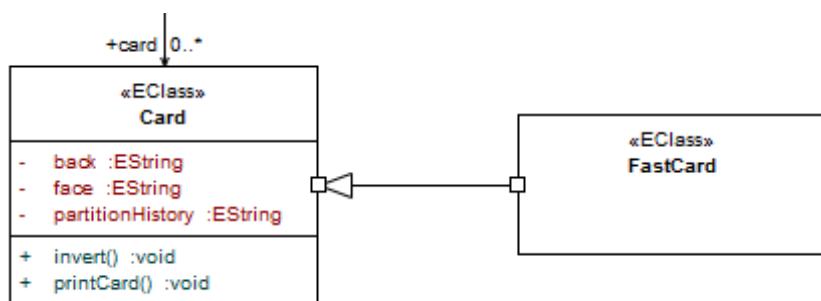


Figure 11.3: Fast cards are a special kind of card

- ▶ Now return to the `check` SDM (in `Partition`) and extend the control flow as depicted in Fig. 11.4.

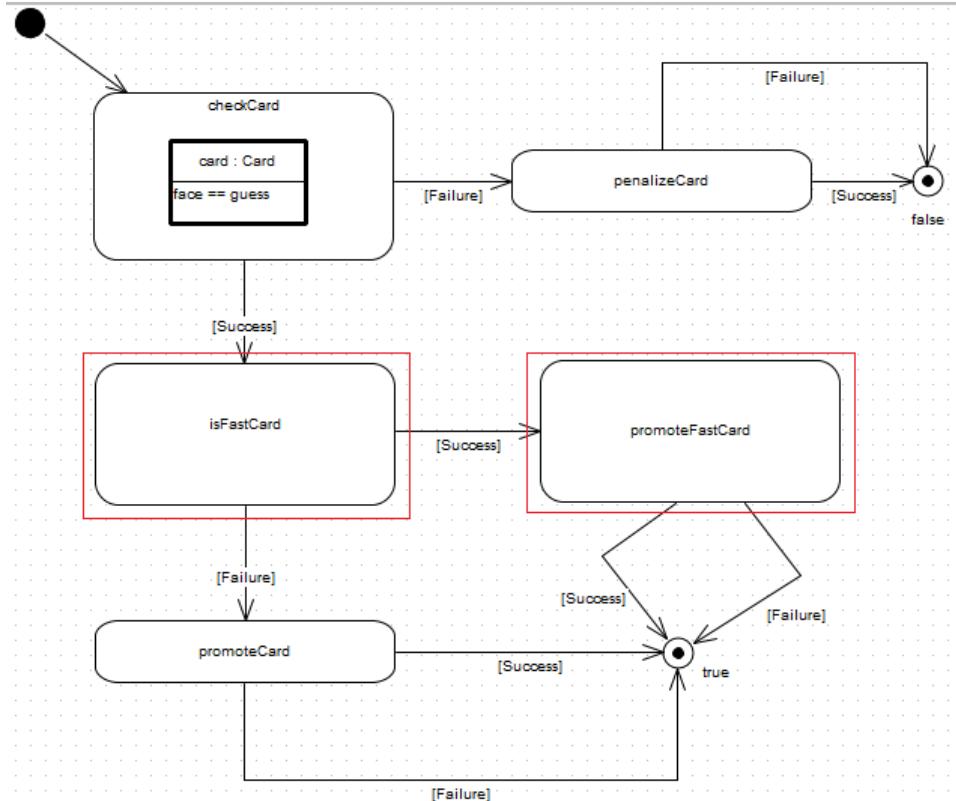


Figure 11.4: Extend check to handle fast cards.

- ▶ As you can see, you have created two new story nodes, `isFastCard`, and `promoteFastCard`.
- ▶ Next, in order to complete the newest conditional, create a bound `FastCard` object variable, named `fastcard` in `isFastCard` (Fig. 11.5).
- ▶ To check the dynamic type, we'll need to create a binding of `card` (of type `Card`) to `fastcard` (of type `FastCard`), so edit the `Binding` tab in the `Object Variable Properties` dialogue (Fig. 11.5). Please note that this tab will not allow any changes unless the `bound` option in `Object Properties` is selected. As you can see, this set up configures the pattern matcher to check for types, rather than `parameters` and `attributes` as we've previously encountered.

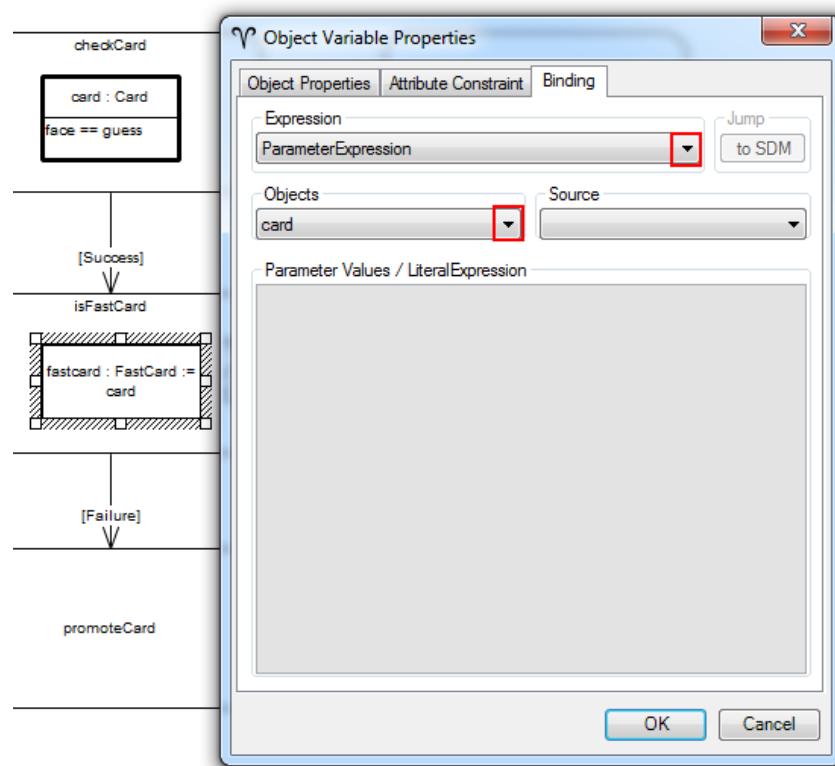


Figure 11.5: Create a binding for `fastcard`

In our case, we could use a *ParameterExpression* or an *ObjectVariableExpression* as `card` is indeed a parameter and has already been used in `checkIfGuessIsCorrect`. We haven't tried the latter yet, so let's use *ObjectVariableExpression*.

- ▶ Update the `fastcard` binding by switching the expression to *ObjectVariableExpression*, with `card` as the target. Note that a binding could also use a *MethodCallExpression* to invoke a method whose return value would be the bound value. This is very useful as it allows invoking helper methods directly in patterns.
- ▶ To finalize the SDM, (i) extract the `promoteFastCard` story pattern and build the pattern according to Fig. 11.6 and (ii) create the parallel `Success` and `Failure` edges from this activity to the stop node returning `true` for the same reason as in `check` earlier. Compare the pattern in Fig. 11.6 to Figs. 4.13 and 4.14, the original promotion and penalizing card movements. As you can see, they're very similar, except `fastCard` is transferred from the current partition (`this`) immediately to the last partition in `box`, identified as having no next `Partition` with an appropriate NAC. Note that a second NAC is used to handle the case where `this` would be the next `Partition`, which is also not what we want.

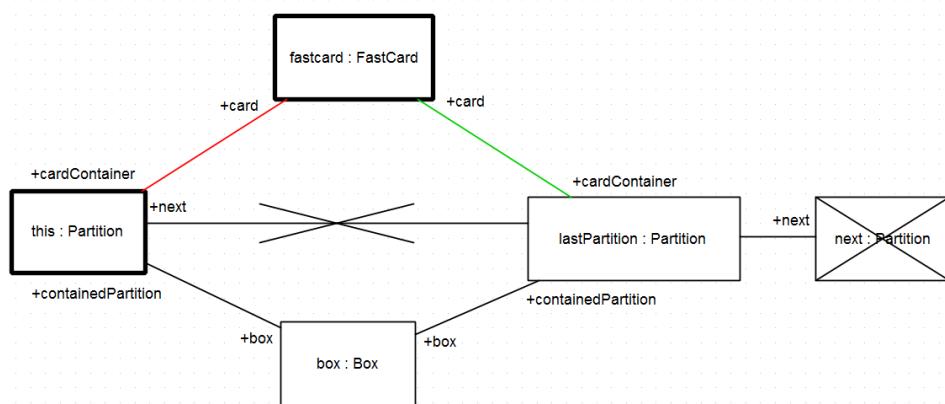


Figure 11.6: Story pattern for handling fast cards.

- ▶ Inspect Fig. 11.11 to see how this is done in the textual syntax.
- ▶ You have now implemented every method using SDMs – fantastic work! Save, validate, and build your metamodel to see some new code. Inspect the implementation for `check`. Can you find the generated type casts for `fastcard`?

- At this point, we encourage you to read each of the textual SDM instructions to try and understand the full scope of eMoflon's features (which start on page 9) but you are of course, free to carry on.

## 11.2 Implementing FastCards

- ▶ Create a new EClass under “LearningBoxLanguage” named **FastCard** which extends **Card**. It doesn’t need any new attributes or methods, so leave its specification empty. It should resemble Fig. 11.7.

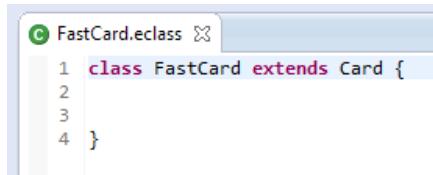


Figure 11.7: FastCards are a special type of Card

- ▶ Open **Partition.eclass** once again, and find `check(card, guess)`. Edit the activity by adding a second `if/else` construct. Call the new assertion pattern `isFastCard`, and the action pattern `promoteFastCard`. Move the original `[promoteCard]` pattern into the `else` branch. The check method should now resemble Fig. 11.8.

```

22 check(card : Card, guess : EString) : EBoolean {
23 if [checkCard]{
24
25 if [isFastCard] {
26 [promoteFastCard]
27 } else {
28 [promoteCard]
29 }
30 return true
31 } else {
32 [penalizeCard]
33 return false
34 }
35 }
36 }
```

Figure 11.8: Checking for FastCards

- ▶ `isFastCard` is a simple, one line pattern. You simply need to create a *binding expression* to bind the object variable `fastCard` (of type **FastCard**), to `card` (of type **Card**) which was passed in as a parameter. Remember, to access parameter values, use a *ParameterExpression* which prefixes the name with a ‘\$’ symbol. Your workspace should now resemble Fig. 11.9.

```

C Partition.eclass isFastCard.pattern ✎
1 pattern isFastCard {
2 @fastCard : FastCard := $card
3 }

```

Figure 11.9: Checking if a card is a FastCard

- To establish `promoteFastCard`, first create four object variables: `@this` for the current partition of the card, `@fastCard` representing the card, `lastPartition` for the last partition in the box, and the `box` containing all partitions. Under `lastPartition`, also create a NAC `next`, which we will use to forbid that the last partition in the box has a next partition. Your pattern should now resemble Fig. 11.10.

```

C Partition.eclass promoteFastCard.pattern ✎
1 pattern promoteFastCard {
2
3 @this : Partition
4
5 @fastCard : FastCard
6
7 lastPartition : Partition
8 ! next : Partition
9
10 box : Box
11 }

```

Figure 11.10: Object variables for `promoteFastCard`

- When creating the necessary link variables remember - this is the pattern that will be invoked when the fast card status has already been confirmed! This means that you'll want to: (1) ensure that all partitions belong to the current box. (2) remove `fastCard` from its current partition, and insert it into `lastPartition` (3) confirm that `lastPartition` does not have a `next` partition, i.e., it really is the last partition in the box.<sup>26</sup>
- Your final pattern should resemble Fig. 11.11. As you can see, this pattern is remarkably similar to the original movement patterns, `promoteCard` and `penalizeCard` (Fig. 4.21). This of course makes sense – the target pattern is only now always the last partition in the box and no longer the current `next`.

---

<sup>26</sup>If you need help remembering how NACs work, review Section 7

```
promoteFastCard.pattern ✘ |
1 pattern promoteFastCard {
2
3 @this : Partition {
4 -box-> box
5 }
6
7 @fastcard : FastCard {
8 --- -cardContainer-> this
9 ++ -cardContainer-> lastPartition
10 }
11
12 lastPartition : Partition {
13 -box-> box
14 -next-> next
15 ! -next-> this
16 }
17
18 ! next : Partition
19
20 box : Box
21 }
```

Figure 11.11: The completed fast card promotion pattern

- ▶ You have now completed *every* method signature using SDMs – fantastic work! Build your project to confirm there aren’t any errors, and review Fig. 11.6 to see how **FastCards** are implemented in the visual syntax.
- ▶ You are encouraged to read the visual SDM sections on each method to understand the full scope of eMoflon’s features (which start on page 9), but you are more than welcome to carry on and complete Part III.

### 11.3 FastCards in the GUI

We hope you haven't forgotten about the GUI! Now that we have a new `card` type, let's quickly try editing our `box` instance so we can experiment with them in our application.

- ▶ To review the details of creating instances, read Part II, Section 3 but for now, open `Box.xmi`, right-click on your first partition, and create a new `FastCard` child element. Open the properties tab below, and edit the `back` and `face` values for testing (Fig. 11.12). As you can see, it has all the same attributes as a standard `card`.

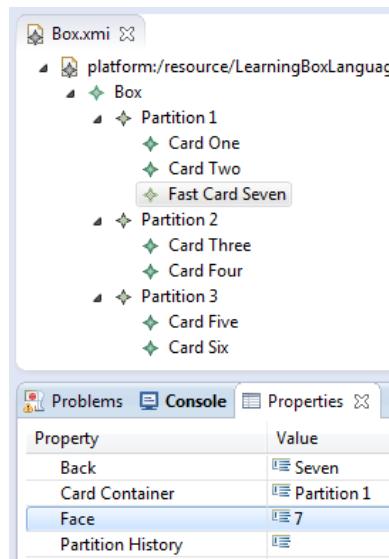


Figure 11.12: Creating and editing a new `FastCard` element

- ▶ Save your file, then open the GUI and try your extended `check` method. Experiment with making wrong and correct guesses for both card types, paying attention their behaviour. If you've done everything right until this point, your newest `FastCard` should act differently than its standard `card` counterparts.

## 12 Reviewing eMoflon's expressions

As you've discovered while making SDMs, eMoflon employs a simple context-sensitive expression language for specifying values. We have intentionally avoided creating a full-blown sub-language, and limit expressions to a few simple types. The philosophy here is to keep things simple and concentrate on what SDMs are good for – expressing structural changes. Our approach is to provide a clear and type-safe interface to a general purpose language (Java) and support a simple *fallback* via calls to injected methods as soon as things get too low-level and difficult to express structurally as a pattern.

The alternative approach to eMoflon would be to support arbitrary expressions, for example, in a script language like JavaScript or in an appropriate DSL<sup>27</sup> designed for this purpose.

We've encountered several different expression types throughout our SDMs so far, and all of them can be used for binding expressions. Since each syntax has used at least three of these once, let's consider what each type would mean:

### **LiteralExpression:**

As usual this can be anything and is literally copied with a surrounding typecast into the generated code. Using *LiteralExpressions* too often is usually a sign for not thinking in a pattern oriented manner and is considered a *bad smell*.

### **MethodCallExpression:**

This would allow invoking a method and binding its return value to the object variable. This is how non-primitive return values of methods can be used safely in SDMs.

### **ParameterExpression:**

This could be used to bind the object variable to a parameter of the method. If the object variable is of a different type than the parameter (i.e., a subtype), this represents basically a successful typecast if the pattern matches.

---

<sup>27</sup>A DSL is a Domain Specific Language: a language designed for a specific task which is usually simpler than a general purpose language like Java and more suitable for the exact task.

**ObjectVariableExpression:**

This can be used to refer to other object variables in preceding story nodes. Just like *ParameterExpressions*, this represents a simple type-cast if the types of the target and the object variable with the binding are different.

## 13 Complex Attribute Manipulation

In Section 4.1 we have modeled the `Partition::check()` method. However, as you might realized, our implementation ignores the `partitionSize`, i.e., the maximal number of cards that fit in a partition. In order to prevent our implementation from “bursting” a partition we gonna use *complex attribute constraints* to extend the SDM for the `Partition::check()` method. In contrast to simple attribute constraints<sup>28</sup>, which provide only binary operations for comparing attribute values (or literals), complex attribute constraints provides a language to specify arbitrary relations on attributes.

### 13.1 Using Complex Attribute Constraints

We start by extending the `penalizeCard` pattern of the `Partition::check()` method in such a way that it only moves a card to the previous partition if the additional card does not exceed the actual `partitionSize` of the previous partition. But before we start we have to extend `Partition` class by an attribute that keeps track of the actual number of card contained in a partition:

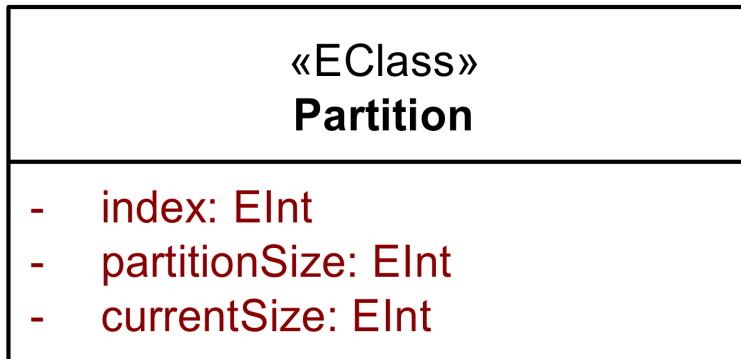


Figure 13.1: adding `currentSize:EInt` to `Partition`

- Add to the class `Partition` the new attribute `currentSize:EInt`.

Now we can start defining complex attribute constraints:

- Navigate to the `penalizeCard` pattern in the `Partition::check()` method and add a `CSP instance`; Following a similar process as cre-

---

<sup>28</sup>First used in Section 4.1 to compare user’s guess against the card’s face value

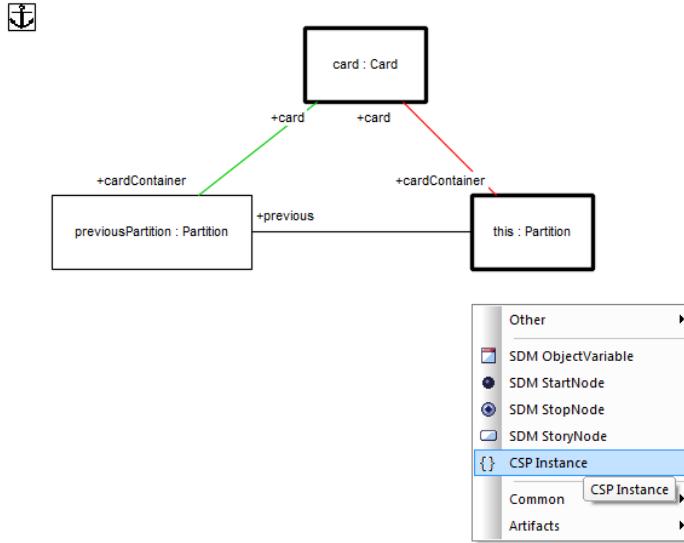


Figure 13.2: Creating a new `CSP instance`

ating a new object variable, i.e., either hit the spacebar or use the toolbox to create a new *CSP instance* (Fig. 13.2).

- ▶ You'll notice a box were you can specify your complex attribute constraints. Enter the attribute constraints as shown in Fig. 13.3 to define that a card is only moved if the additional card does not exceed the actual `partitionSize` of the previous partition.

Basically an attribute constraint is a predicate of the form

`SYMB(param1, ..., paramn)`

consisting of an symbol `SYMB` naming the predicate, followed by a list of parameters. A parameter can be:

- A *local variables* of the form  
`varname:Type`,  
as for example, `newPrevCurrentSize:EInt` (first line Fig. 13.3)  
defines an local variable `newPrevCurrentSize` of type `EInt`.
- A *constant* is of the form  
`value:Type`,  
as for example, `1:EInt` (first line Fig. 13.3) represents constant 1 of type `EInt`.

- An *attribute variable* of the form

`objectVarName.attributeName,`

and refers to the current value of the attribute `attributeName` of object variable `objectVarName`, while an attribute variable of the form `objectVarName.attributeName'` refers to an attribute value after the transformation. This is necessary for, e.g., write the java expression `a.x=a.x+1` as the predicate `+(a.x', a.x, 1:EInt)`.

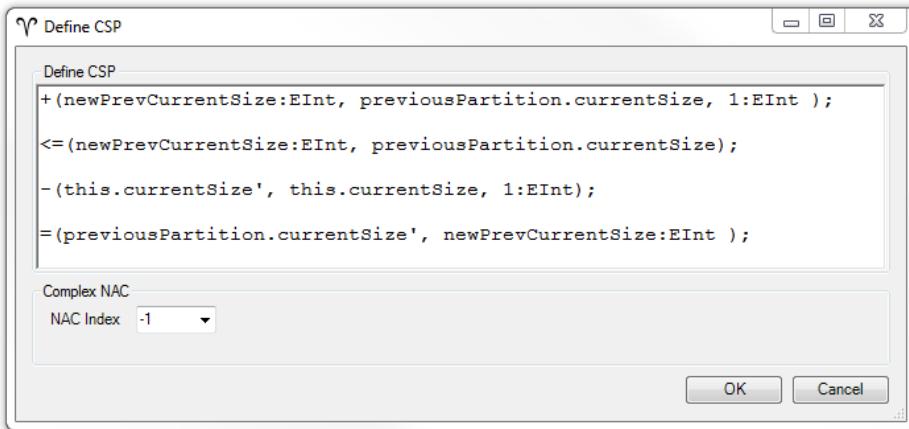


Figure 13.3: Defining complex attribute constraints

The CSP instance (the set of attribute constraints) shown in Fig. 13.3 consist of the constraints with meaning<sup>29</sup>:

`+ (newPrevCurrentSize:EInt, previousPartition.currentSize, 1:EInt),`  
meaning that:

`newPrevCurrentSize == previousPartition.currentSize + 1;`

`<=(newPrevCurrentSize:EInt, previousPartition.currentSize),`  
meaning that:

`newPrevCurrentSize ≤ previousPartition.currentSize;`

`-(this.currentSize', this.currentSize, 1:EInt),`

meaning that:

`this.currentSize' == this.currentSize - 1;`

`=(previousPartition.currentSize', newPrevCurrentSize:EInt),`

meaning that:

`previousPartition.currentSize' == newPrevCurrentSize.`

- Export the project as usual. Before you start generating code you have

---

<sup>29</sup>An complete list of build in constraints is given at the end of this chapter.

to switch the code generation engine. Open the `moflon.properties.xmi` located in your project and change the SDM Codegenerator Method Body Handler to `DEMOCLES_ATTRIBUTES` (Fig. 13.4).

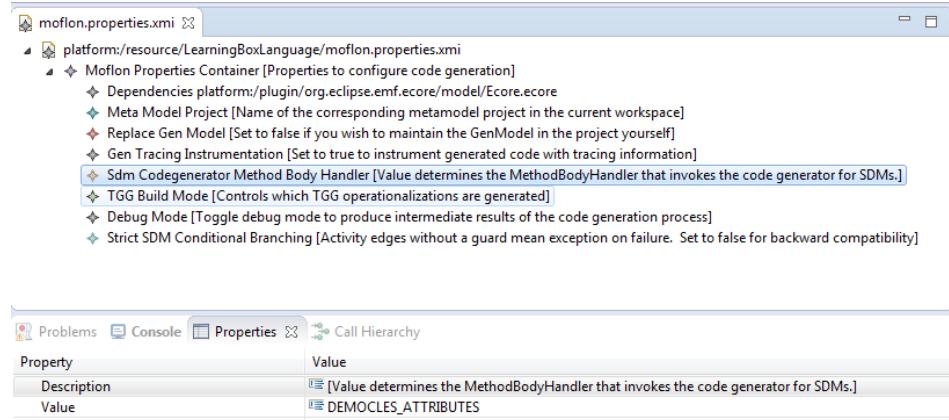


Figure 13.4: Switch SDM Codegenerator Method Body Handler to DEMOCLES\_ATTRIBUTES

The code generator engine is now able to solve the CSP instances, i.e., the constraints are operationalized and put in correct order. The code resulting for the CSP instance is shown in Figs. 13.5 and 13.6 for the LHS and RHS, respectively.

```

int this_currentSize = _this.getCurrentSize();
int this_currentSize_prime;
this_currentSize_prime = this_currentSize - 1;
int previousPartition_currentSize = previousPartition.getCurrentSize();
int newPrevCurrentSize;
newPrevCurrentSize = previousPartition_currentSize + 1;
if (newPrevCurrentSize < previousPartition_currentSize)
{
 int previousPartition_currentSize_prime;
 previousPartition_currentSize_prime = newPrevCurrentSize;
 ...
}

```

Figure 13.5: LHS pattern code generated for the CSP instance

```
this.setCurrentSize(this_currentSize_prime);
previousPartition.setCurrentSize(previousPartition_currentSize_prime);
```

Figure 13.6: RHS pattern code generated for the CSP instance

Notice that the attribute variables `this.currentSize'` and `previousPartition.currentSize'` (represented in the code by `this_currentSize_prime` and `previousPartition_currentSize_prime`, respectively) are treated as local variables on the LHS and then assigned on the RHS to the corresponding attribute values. Hence we might rewrite the CSP instance shown in Fig. 13.3 to the more compact shown in Fig.13.7.

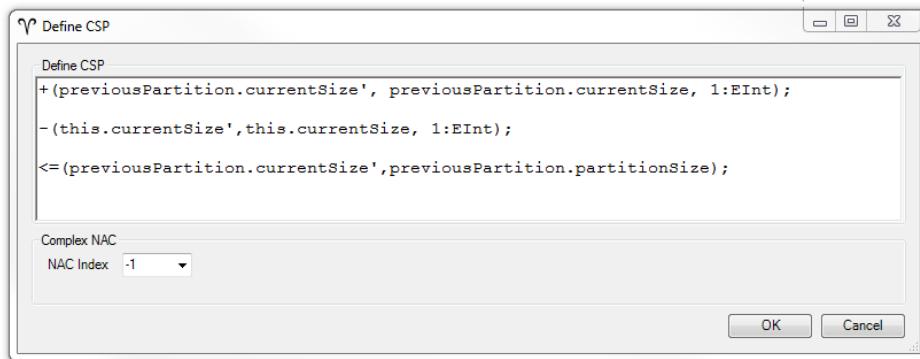


Figure 13.7: Defining complex attribute constraints

## 13.2 Defining Own Complex Attribute Constraints

In Section 10 we have implemented the `Box::toString()` method to obtain a string representation of the box. The method uses internally the method `Box::addToStringRep(Card) : EString` to create a string from a card, that was realized using handwritten code via injections.

In the following the `Box::addToStringRep(Card) : EString` method should be replaced by using complex attribute constraints.

- ▶ Reconsider the SDM shown in Fig. 10.7. First add an attribute constraint for appending the string "PartitionContent:" to the `stringRep` attribute of `Box`. To this end, add to the `ForAllPartitons` pattern the complex attribute constraint:

---

```
+ (this.stringRep', this.stringRep, "Partition Content:" : EString),
```

Note that the “+” predicate is overloaded, while +( :EInt, :EInt, :EInt) (as used in the previous section), means integer addition, +( :EString, :EString, :EString) is concatenation.

In a next step the *ForAllCards* pattern should be extended by an complex attribute constraints such that for each card the string containing the front and back is appended to *this.stringRep*. Instead of using the the concat (+) operation again, we just cand to define our own constraint by

- ▶ just adding the CSP instance:

`myConcat(this.stringRep', this.stringRep, card.face, card.back)` to the *ForAllCards* pattern. The complete SDM should now look as shown in Fig. 13.8.

- ▶ Export as usual. If you are trying to build the metamodel project in eclipse you get an error message (Fig. 13.9) that informs you that the attribute constraint with signature:

`myConcat( :EString, :EString, :EString, :EString)` is unknown. If you look at `LearningBoxLanguage/lib/` there is a new file `LearningBoxLanguageAttributeConstraintsLib.xmi`

- ▶ Open the file. It contains a constraint specification `myConcat` (Fig.13.10 bottom) that represents the signature (`myConcat( :EString, :EString, :EString, :EString)`) which is derived from the information of the CSP-instance in EA. To define the meaning of `myConcat` we have to define operations for the constraint.
- ▶ Right click the operation specification group for `myConcat` (top of Fig. 13.10) and add as new child an **operation specification**.

An operation<sup>30</sup> is specified by an *Adornment String*, and a **Specification** that is a template defining the code to be generated.

- ▶ Complete the operation specification as shown in Fig.13.11. The adornment string FBBB means that before the operation can be executed the values for parameter 2–4 must be known (Bound), and the Free parameter (i.e., the first parameter in this case) is derived from the Bound parameters. The *Specification* field contains the template string that specifies a java expression that defines how the Free parameter is derived from the Bound ones. The template uses the parameter identifier as variables (sounded with \$ \$).

---

<sup>30</sup>Double click on the operation specification to open the properties view.

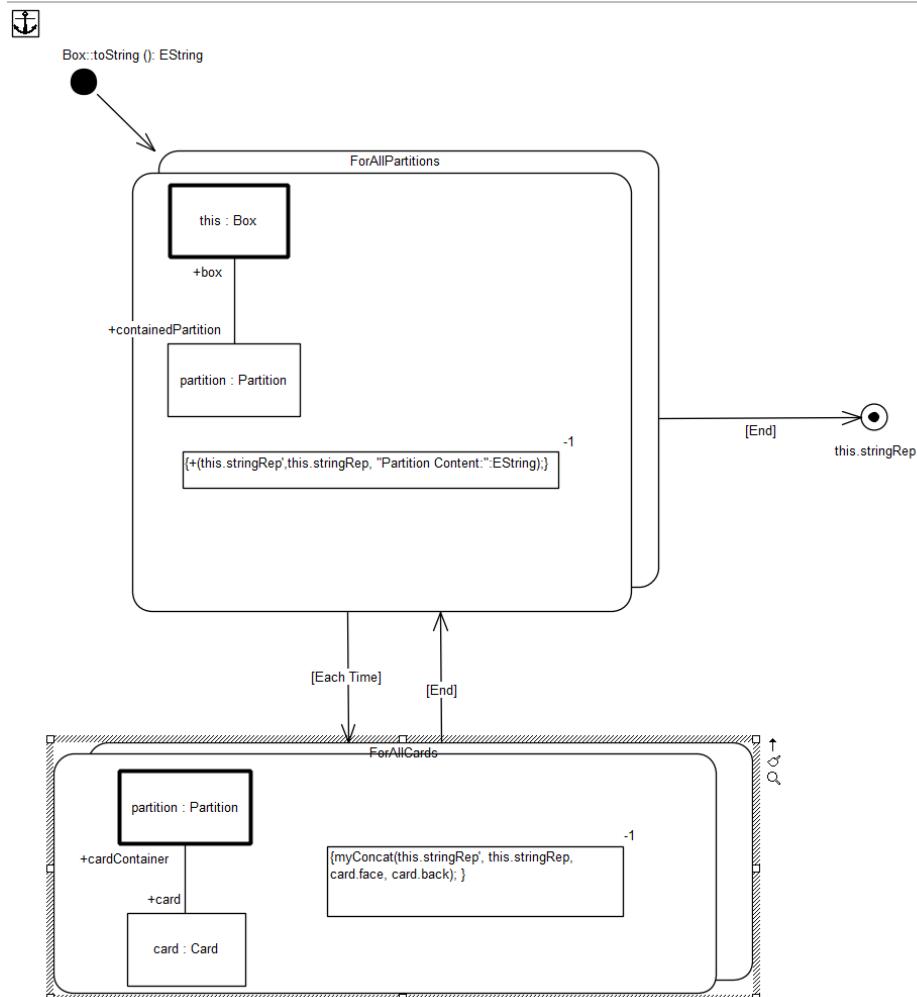
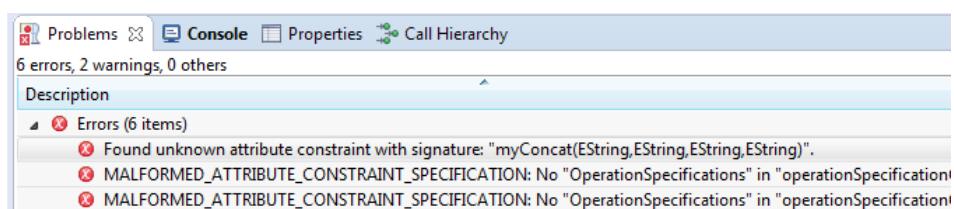
Figure 13.8: Complet SDM for `Box::toString()`

Figure 13.9: Error

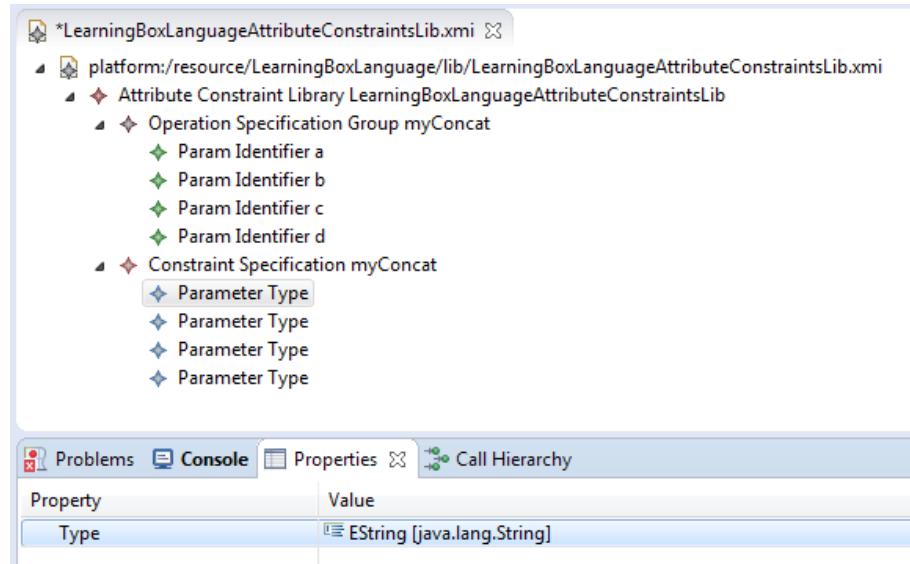


Figure 13.10: The content of LearningBoxLanguageAttributeConstraintsLib.xmi

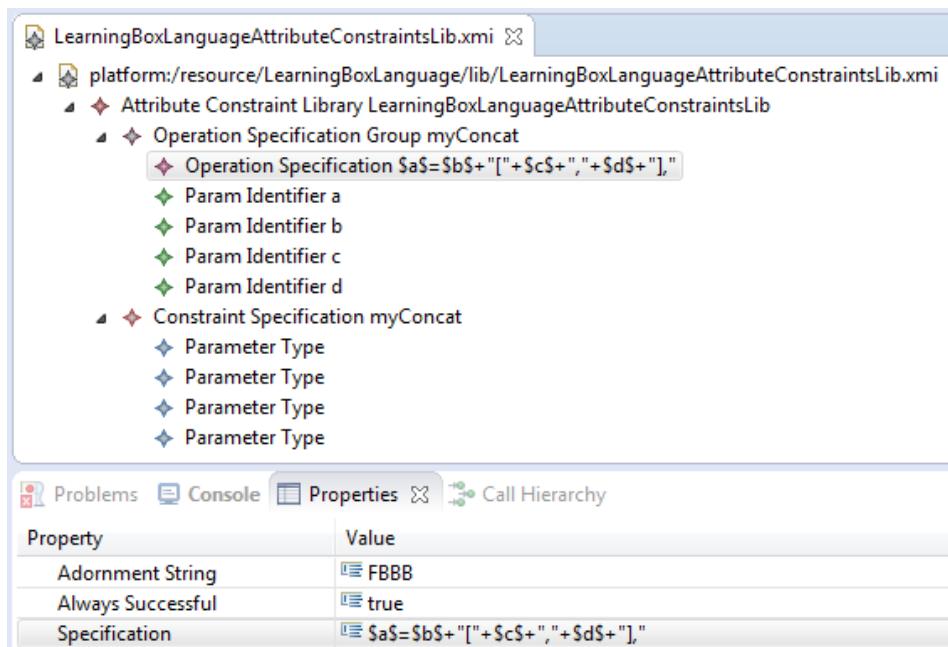


Figure 13.11: The operation specification for myConcat

### 13.3 Build in Complex Attribute Constraints

| Symbol | Signatures                                                                                                                                       | Semantics    |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| =      | (a:EInt , b:EInt)<br>(a:EDouble , b:EDouble)<br>(a:EFloat , b:EFloat)<br>(a:EShort , b:EShort)<br>(a:ELong , b:ELong)<br>(a:EString , b:EString) | $(a = b)$    |
| $\leq$ | (a:EInt , b:EInt)<br>(a:EDouble , b:EDouble)<br>(a:EFloat , b:EFloat)<br>(a:EShort , b:EShort)<br>(a:ELong , b:ELong)                            | $(a \leq b)$ |
| <      | (a:EInt , b:EInt)<br>(a:EDouble , b:EDouble)<br>(a:EFloat , b:EFloat)<br>(a:EShort , b:EShort)<br>(a:ELong , b:ELong)                            | $(a < b)$    |
| $\geq$ | (a:EInt , b:EInt)<br>(a:EDouble , b:EDouble)<br>(a:EFloat , b:EFloat)<br>(a:EShort , b:EShort)<br>(a:ELong , b:ELong)                            | $(a \geq b)$ |
| >      | (a:EInt , b:EInt)<br>(a:EDouble , b:EDouble)<br>(a:EFloat , b:EFloat)<br>(a:EShort , b:EShort)<br>(a:ELong , b:ELong)                            | $(a > b)$    |

| Symbol | Signatures                                                                                                                                                                                                                   | Semantics         |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|
| +      | (a:Eint , b:EInt, c:EInt)<br>(a:EDouble , b:EDouble, c:EDouble)<br>(a:EFloat , b:EFloat, c:EFloat)<br>(a:EShort , b:EShort, c:EShort)<br>(a: , b:, c:)<br>(a:ELong , b:ELong, c:ELong)<br>(a:EString , b:EString, c:EString) | $(a = b + c)$     |
| -      | (a:Eint , b:EInt, c:EInt)<br>(a:EDouble , b:EDouble, c:EDouble)<br>(a:EFloat , b:EFloat, c:EFloat)<br>(a:EShort , b:EShort, c:EShort)<br>(a:ELong , b:ELong, c:ELong)                                                        | $(a = b - c)$     |
| /      | (a:Eint , b:EInt, c:EInt)<br>(a:EDouble , b:EDouble, c:EDouble)<br>(a:EFloat , b:EFloat, c:EFloat)<br>(a:EShort , b:EShort, c:EShort)<br>(a:ELong , b:ELong, c:ELong)                                                        | $(a = b/c)$       |
| *      | (a:Eint , b:EInt, c:EInt)<br>(a:EDouble , b:EDouble, c:EDouble)<br>(a:EFloat , b:EFloat, c:EFloat)<br>(a:EShort , b:EShort, c:EShort)<br>(a:ELong , b:ELong, c:ELong)                                                        | $(a = b \cdot c)$ |

## 14 Conclusion and next steps

Congratulations – you’ve reached the end of eMoflon’s introduction to unidirectional model transformations! You’ve learnt that SDMs are declared as *activities*, which consist of *activity nodes*, which are either *story patterns* or *statement nodes* (for method calls). *Patterns* are made up of *object* and *link variables* with appropriate attribute constraints. Each of these variables can be given different *binding states*, binding operators, and can be marked as negative (for expressing NACs).

To further test your amazing story driven modelling skills, challenge yourself by:

- Adjusting `check` to eject the card from the box if it is guessed correctly and contained in the last partition (to signal it’s been learnt). Do you know how `check` currently handles this?
- Editing `Partition.empty()` to include a method call to `removeCard`, thus reusing this previous SDM
- Modifying the GUI source files to execute all methods

If you have any comments, suggestions, or concerns for this part, feel free to drop us a line at [contact@moflon.org](mailto:contact@moflon.org). Otherwise, if you enjoyed this section, continue to Part IV to learn about Triple Graph Grammars, or Part V for Model-to-Text Transformations. The final part of this handbook – Part VI: Miscellaneous – contains a full glossary, eMoflon hotkeys, and tips and tricks in EA which you might find useful when creating SDMs in the future.

For more detailed information on each part, please refer to Part 0, which can be downloaded at <http://tiny.cc/emoflon-rel-handbook/part0.pdf>.

Cheers!

# Glossary

**Activity** Top-most element of an SDM.

**Activity Edge** A directed connection between activity nodes describing the control flow within an activity.

**Activity Node** Represents atomic steps in the control flow of an SDM. Can be either a story node or statement node.

**Assignments** Used to set attributes of object variables.

**Attribute Constraint** A non-structural constraint that must be satisfied for a story pattern to match. Can be either an assertion or assignment.

**Binding State** Can be either *bound* or *unbound/free*. See *Bound vs Unbound*.

**Binding Operator** Determines whether a variable is to be *checked*, *created*, or *destroyed* during pattern matching.

**Binding Semantics** Determines if an object variable *must* exist (*mandatory*), may not exist (*negative*; see *NAC*), or is *optional* during *pattern matching*.

**Bound vs Unbound** Bound variables are completely determined by the current context, whereas unbound (free) variables have to be determined by the *pattern matcher*. `this` and parameter values are always bound.

**Dangling Edges** An edge with no target or source. Graphs with dangling edges are invalid, which is why dangling edges are avoided and automatically deleted by the pattern matching engine.

**EA** Enterprise Architect; The UML visual modeling tool used as our visual frontend.

**Edge Guards** Refine the control flow in an activity by guarding activity edges with a condition that must be satisfied for the activity edge to be taken.

**Link Variable** Placeholders for links between matched objects.

**Literal Expression** Represents literals such as true, false, 7, or “foo.” See Section 12.

**MethodCallExpression** Used to invoke any method. See Section 12.

**NAC** Negative Application Condition; Used to specify structures that must not be present for a rule to be applied.

**Object Variable** Place holders for actual objects in the current model to be determined during pattern matching.

**ObjectVariableExpression** Used to reference other object variables. See Section 12.

**Parameter Expression** Used to refer to method parameters. See Section 12.

**(Graph) Pattern Matching** Process of assigning objects and links in a model to the object and link variables in a pattern in a type conform manner. This is also referred to as finding a match for the pattern in the given model.

**Statement Node** Used to invoke methods as part of the control flow in an activity.

**Story Node** *Activity node* that contains a *story pattern*.

**Story Pattern** Specifies a structural change of the model.

**Unification** An extension of the Object Oriented “Everything is an object” principle, where everything is regarded as a *model*, even the metamodel which defines other models.

# An Introduction to Metamodelling and Graph Transformations

---

*with eMoflon*



---

## Part IV: Triple Graph Grammars

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

For further information contact us at [contact@emoflon.org](mailto:contact@emoflon.org).

*The eMoflon team*  
Darmstadt, Germany (August 2015)

# Contents

|   |                                               |    |
|---|-----------------------------------------------|----|
| 1 | Triple Graph Grammars in a nutshell . . . . . | 3  |
| 2 | Setting up your workspace . . . . .           | 6  |
| 3 | Creating your TGG schema . . . . .            | 16 |
| 4 | Specifying TGG rules . . . . .                | 23 |
| 5 | TGGs in action . . . . .                      | 47 |
| 6 | Extending your transformation . . . . .       | 52 |
| 7 | Model Synchronization . . . . .               | 56 |
| 8 | Model Generation . . . . .                    | 59 |
| 9 | Conclusion and next steps . . . . .           | 64 |
|   | Glossary . . . . .                            | 65 |

## 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<sup>1</sup> 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 straightforward 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

---

<sup>1</sup>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,<sup>2</sup> into a `Dictionary`, a single flat container able to hold an unlimited number of entries classified by difficulty (Fig. 0.1).

---

<sup>2</sup>For a detailed review on Leitner's Learning Box, see Part II, Section 1

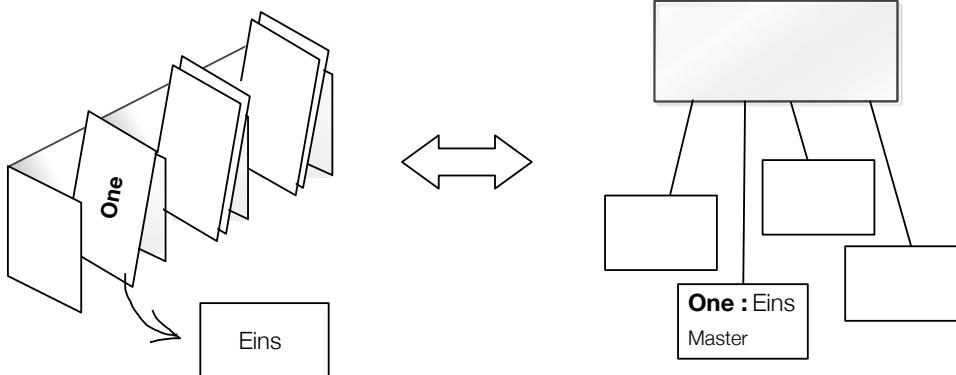


Figure 0.1: Transforming Leitner’s learning box into a dictionary

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 [?],<sup>3</sup> 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

---

<sup>3</sup>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 *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.<sup>4</sup>

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!

---

<sup>4</sup>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.

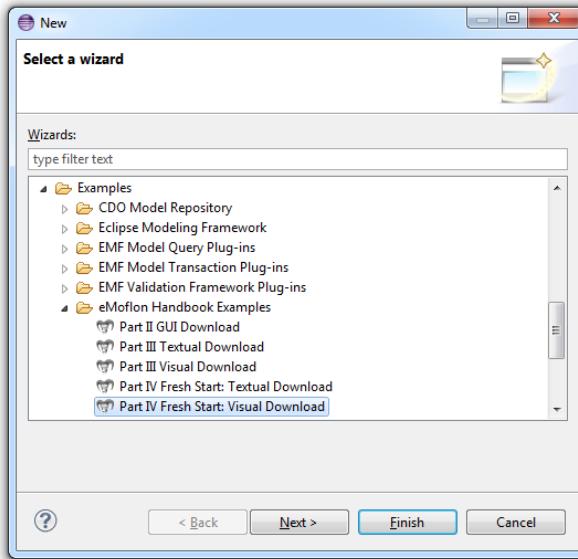


Figure 2.1: Initialize your workspace with your preferred syntax

- After loading, if your workspace does not resemble ours in Fig. 2.2, with eMoflon’s “Build” icon on the toolbar 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.

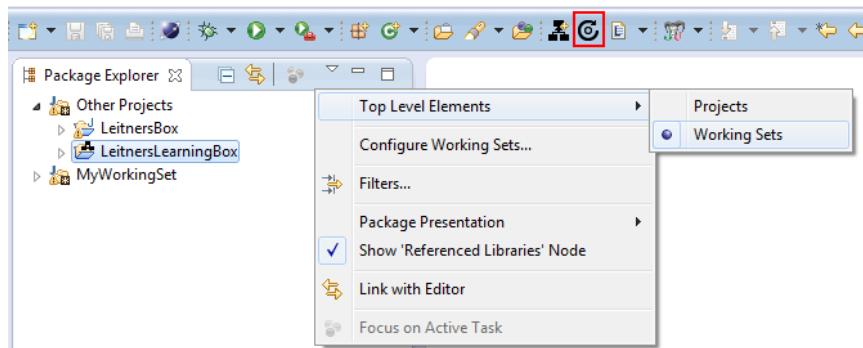


Figure 2.2: Setting your Package Explorer

- 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.

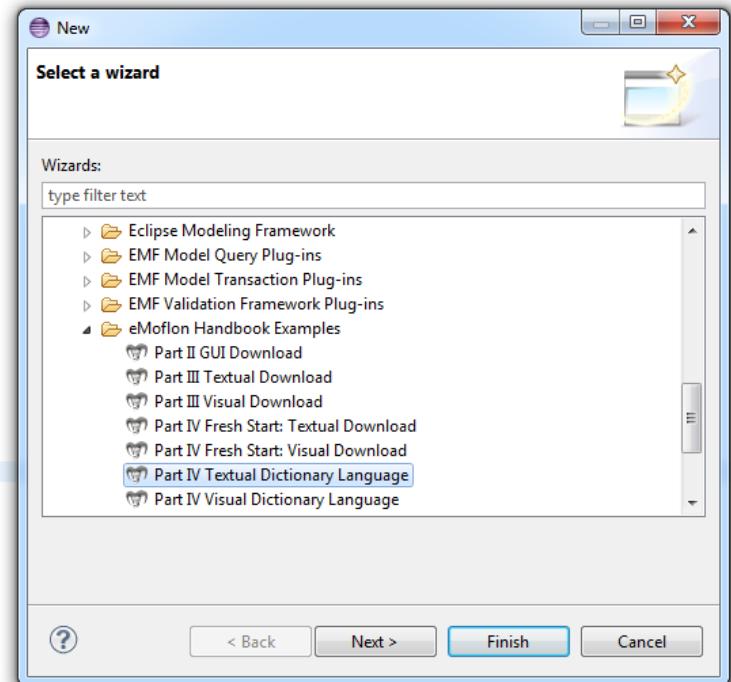


Figure 2.3: Get the visual **DictionaryLanguage** metamodel

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

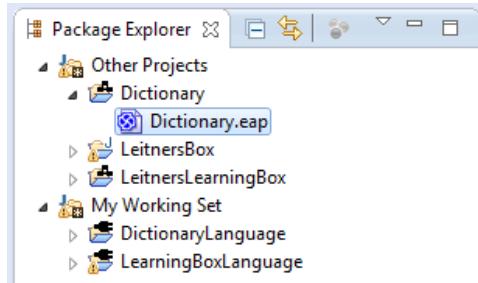


Figure 2.4: Dictionary metamodel successfully copied into the workspace

- 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.

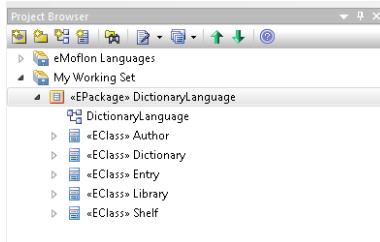


Figure 2.5: The `DictionaryLanguage` metamodel structure

- 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**.

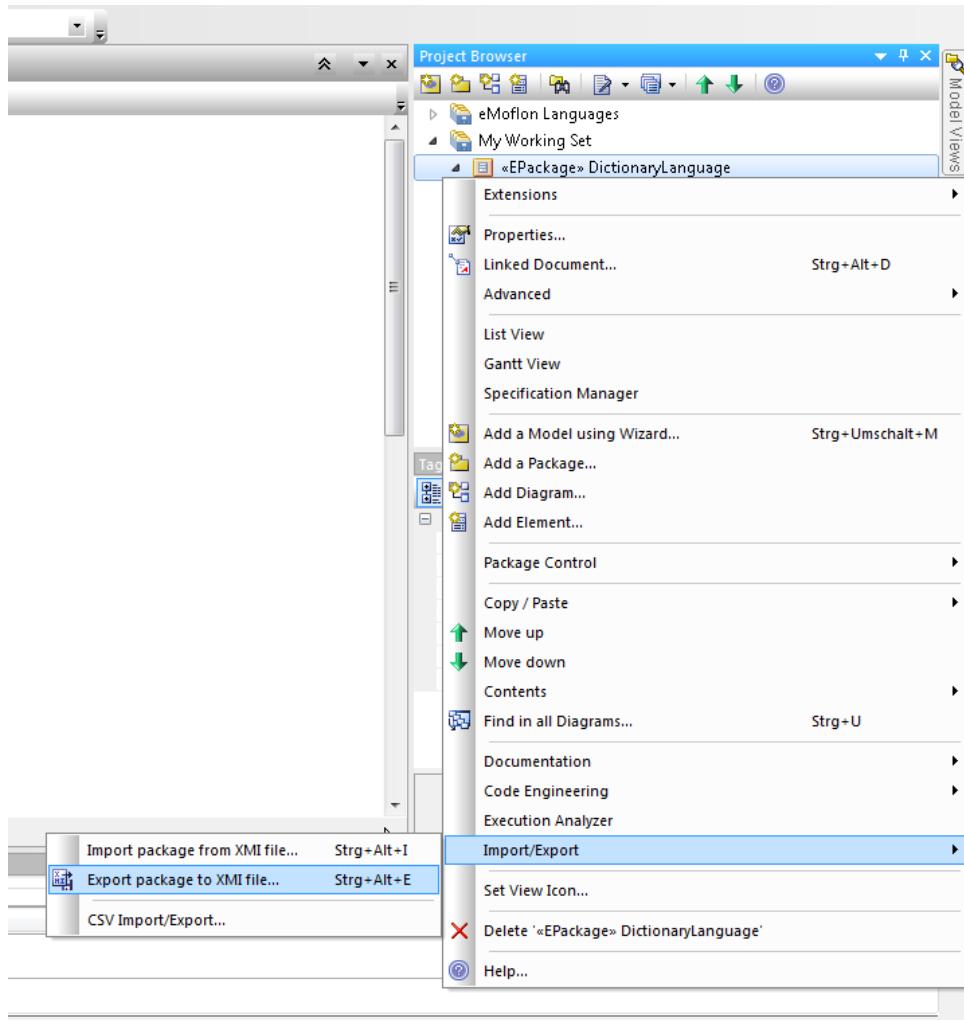


Figure 2.6: Starting the export process in EA

- ▶ 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...”

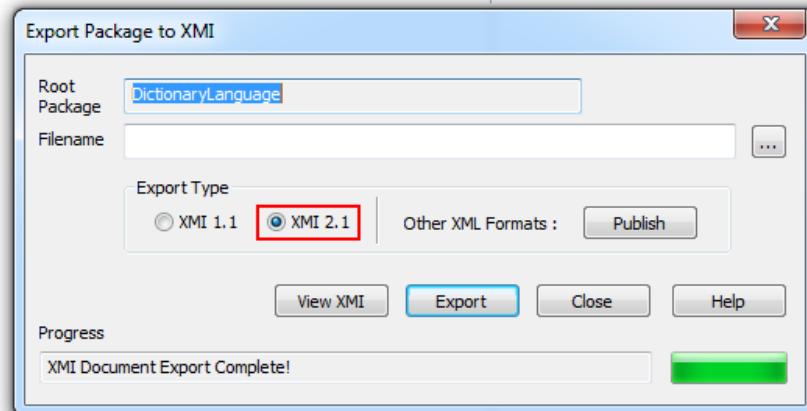


Figure 2.7: Exporting the metamodel to a file

- ▶ 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.

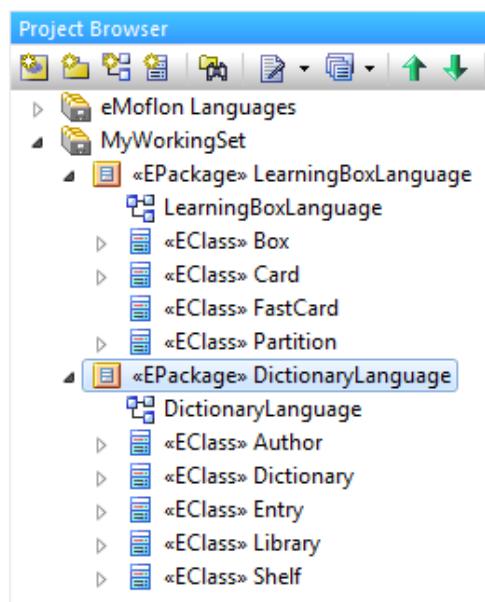


Figure 2.8: The TGG metamodels successfully included in one EAP

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

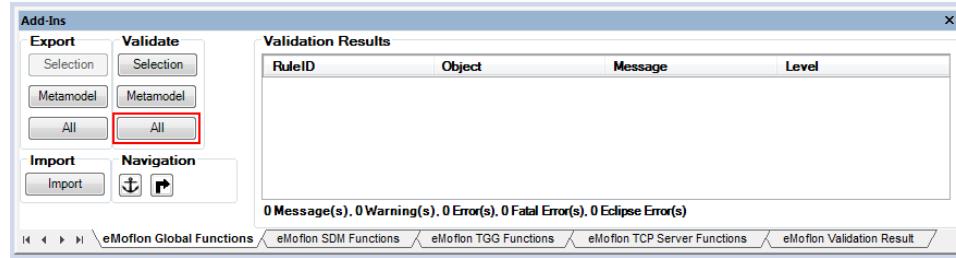


Figure 2.9: No validation errors for `LeitnersLearningBox`

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

<sup>5</sup>To review the details of how to use the eMoflon control panel, read Section 2.8 from Part II

### 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.

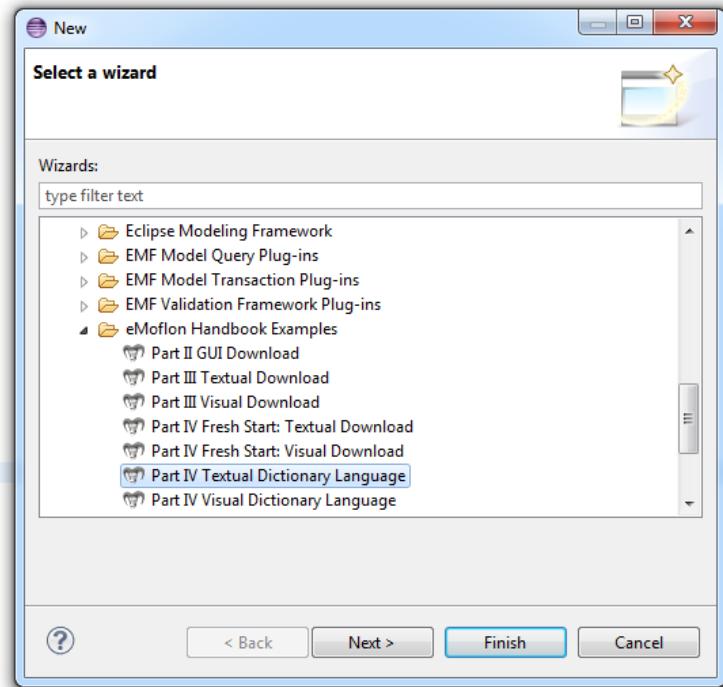


Figure 2.10: Get the textual Dictionary project

- ▶ 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.

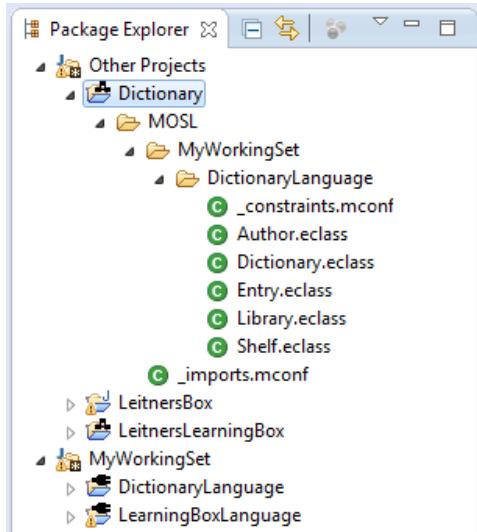


Figure 2.11: DictionaryLanguage’s metamodel structure

- ▶ 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).

```

_c _imports.mconf ✘
1 import MocaTree
2 import Ecore
3 import SDMLanguage
4 import Dictionary

```

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.

### 3.1 Visual TGG Schema

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

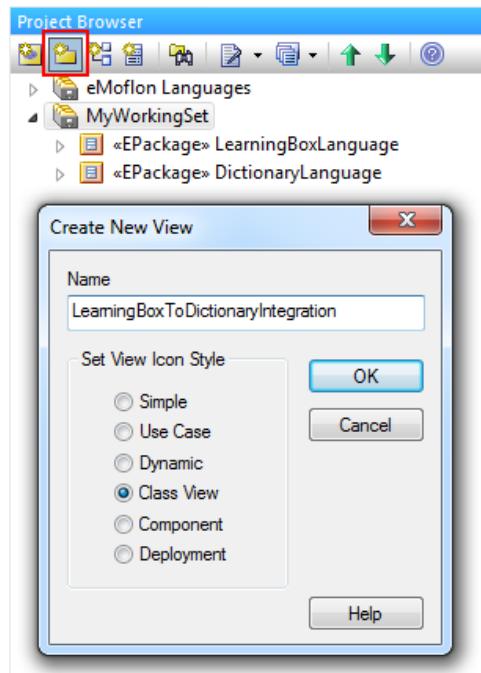


Figure 3.1: Create a new TGG integration package

- 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 `DictionaryLanguage` 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.

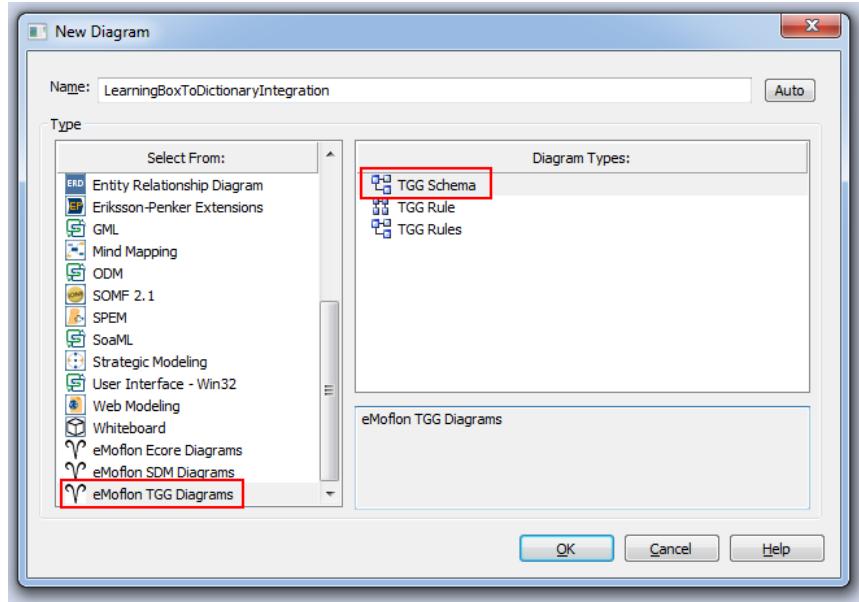


Figure 3.2: Choose TGG Schema as your diagram type

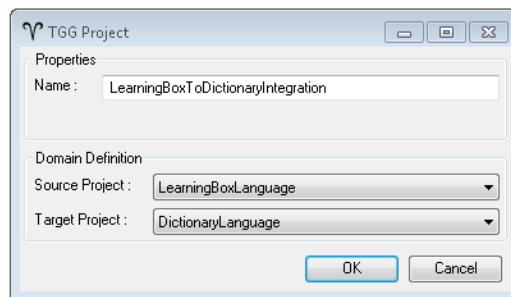


Figure 3.3: Set the source and target projects for the TGG project

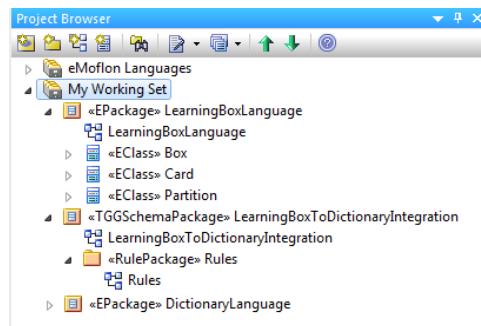


Figure 3.4: Initial structure of a new TGG project

- ▶ 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.<sup>6</sup>

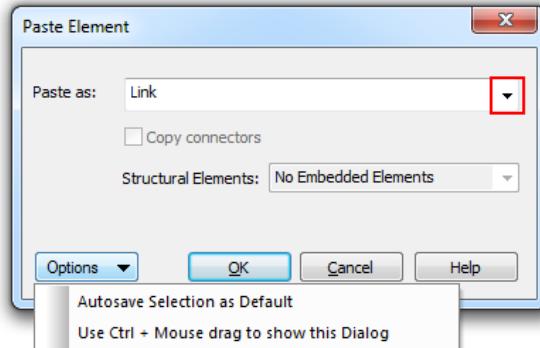


Figure 3.5: Copying an element as a simple link

- ▶ 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).

---

<sup>6</sup>Take caution: If you press **Ctrl + Delete** to delete the element in this diagram, you will also delete it from its original metamodel package!

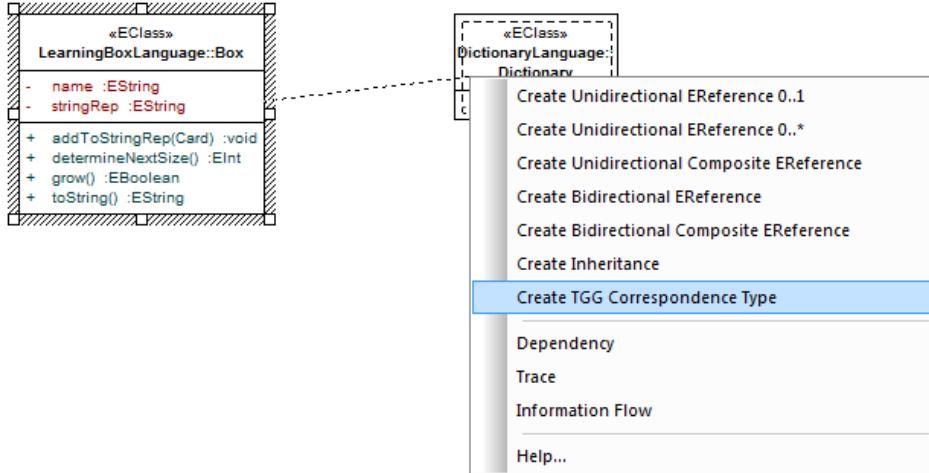


Figure 3.6: Quick-link to create a correspondence type

- 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.

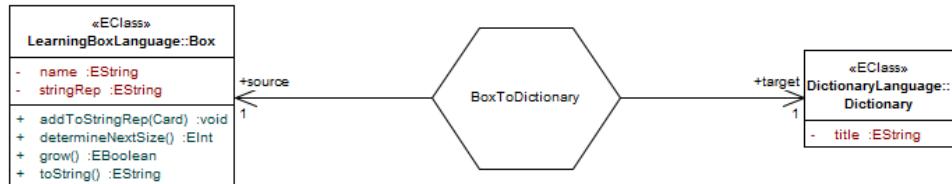


Figure 3.7: An established correspondence type

- You've just finished initializing 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).

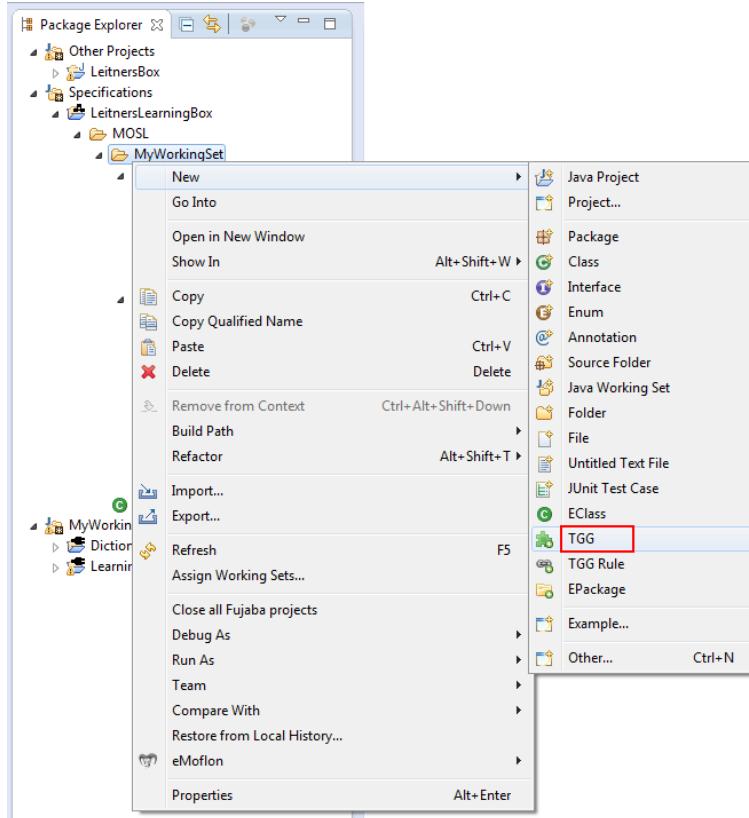


Figure 3.8: Creating a new TGG schema

- ▶ 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.

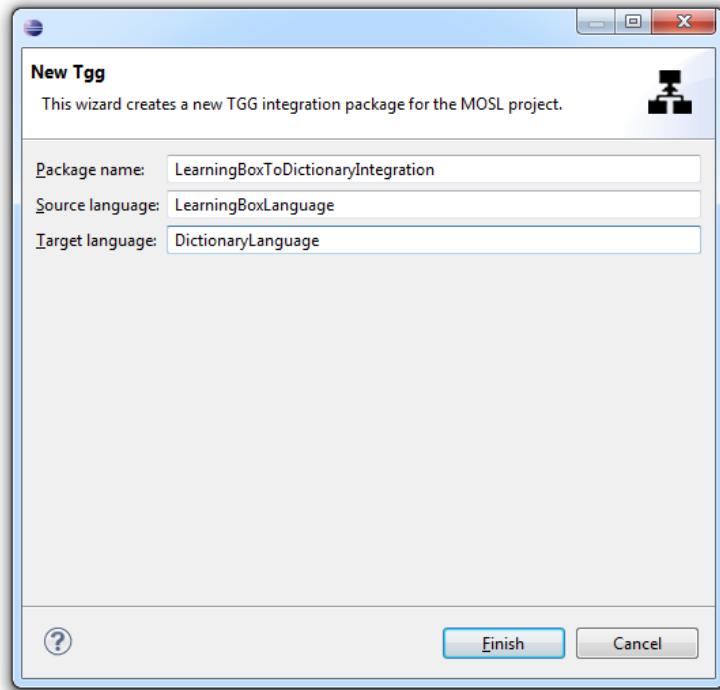


Figure 3.9: Setting your `source` and `target` metamodels

- ▶ 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).

```
schema.sch
source /LearningBoxLanguage
target /DictionaryLanguage

class BoxToDictionary {
 source -> Box
 target -> Dictionary
}
```

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**.

## 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**.

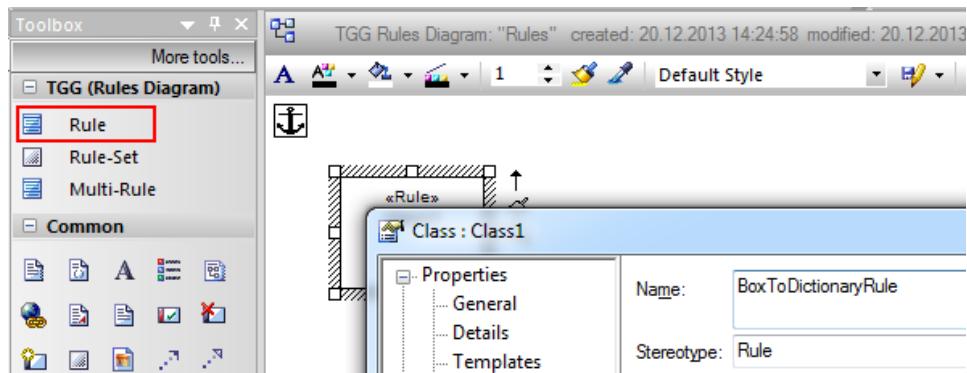


Figure 4.1: Creating a TGG rule

- 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**.<sup>7</sup> The **name** and **binding operator** should already be set to **box** and **create**. Repeat the action to create an instance of **Dictionary**.

<sup>7</sup>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*.<sup>8</sup> 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` *CSP* and `dictionary.title` remain consistent.

- 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.

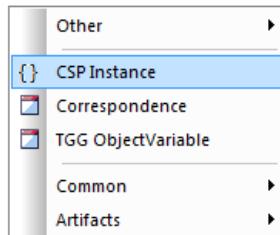


Figure 4.2: CSP instance from the toolbox

- 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.

---

<sup>8</sup>First defined in Part III, Section 4

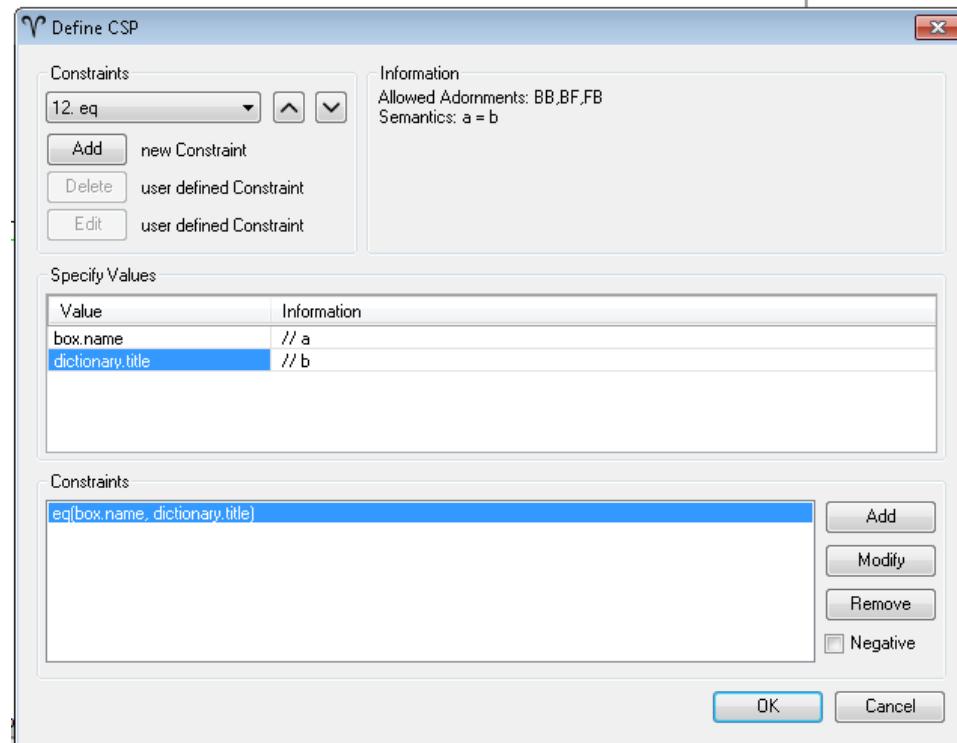


Figure 4.3: Completing the constraint

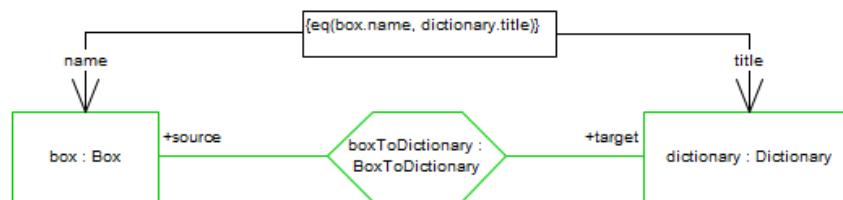


Figure 4.4: A TGG rule with an attribute constraint

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.<sup>9</sup>

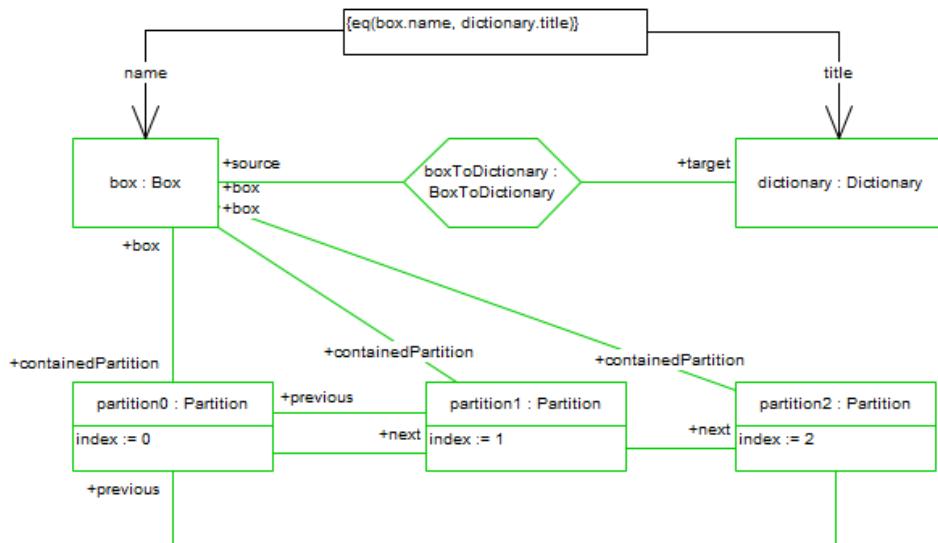


Figure 4.5: Complete TGG rule diagram for **BoxToDictionaryRule**

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.

---

<sup>9</sup>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.

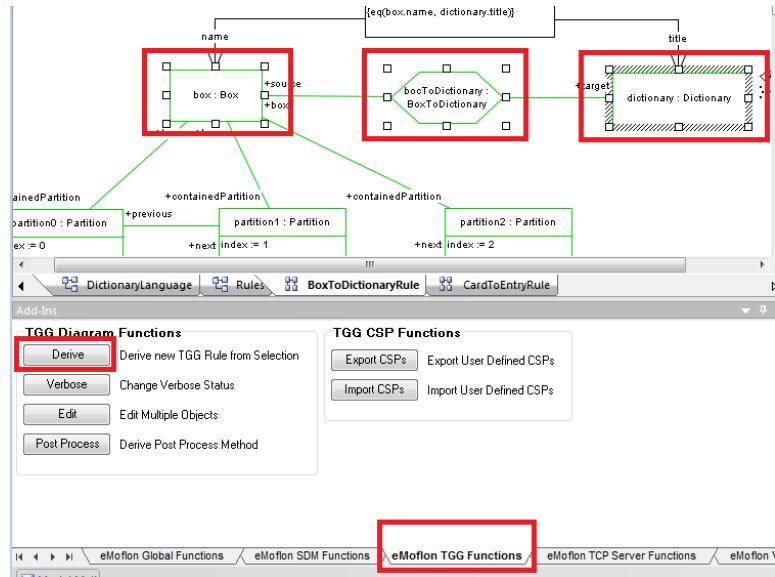


Figure 4.6: Derive a new rule from an existing one

- ▶ 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` dropdown 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).

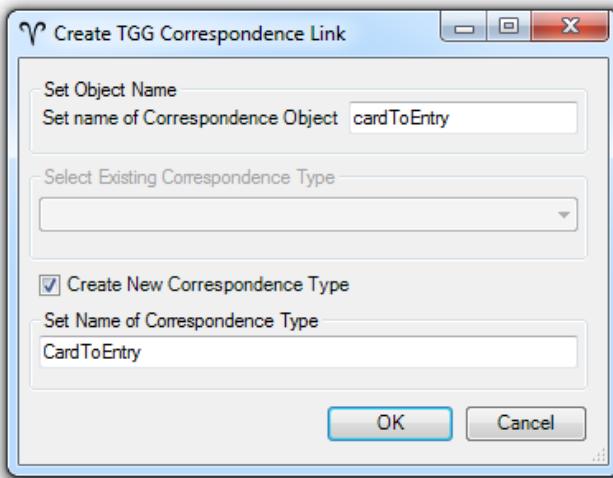


Figure 4.7: Create a new correspondence type on-the-fly

- ▶ 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>`.

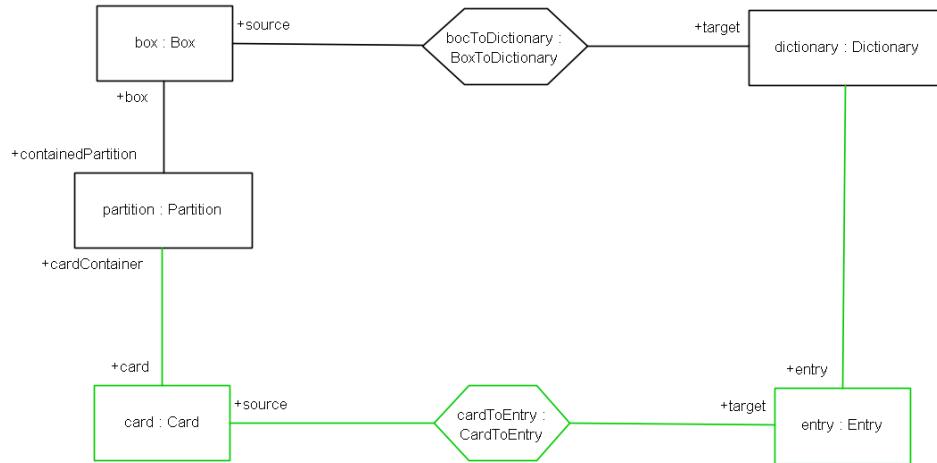


Figure 4.8: `CardToEntryRule` without attribute manipulation

- ▶ 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).

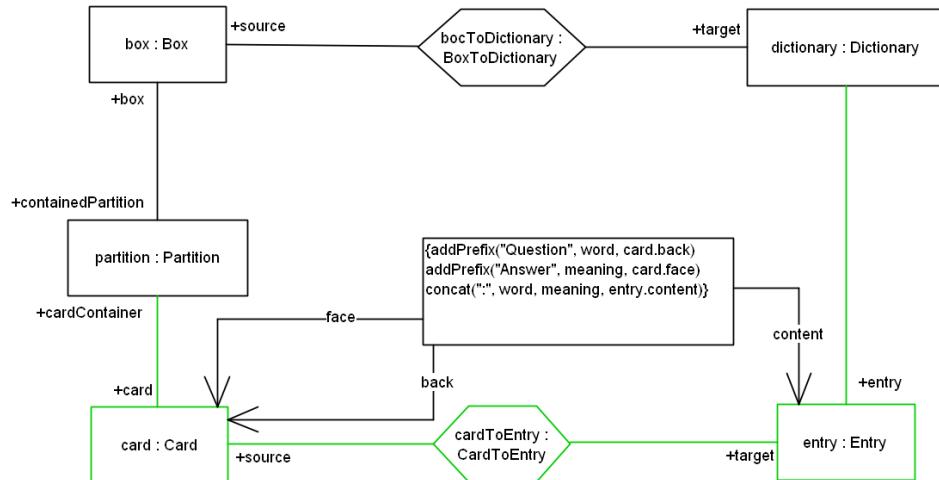


Figure 4.9: Attribute manipulation for `card` and `entry`

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.

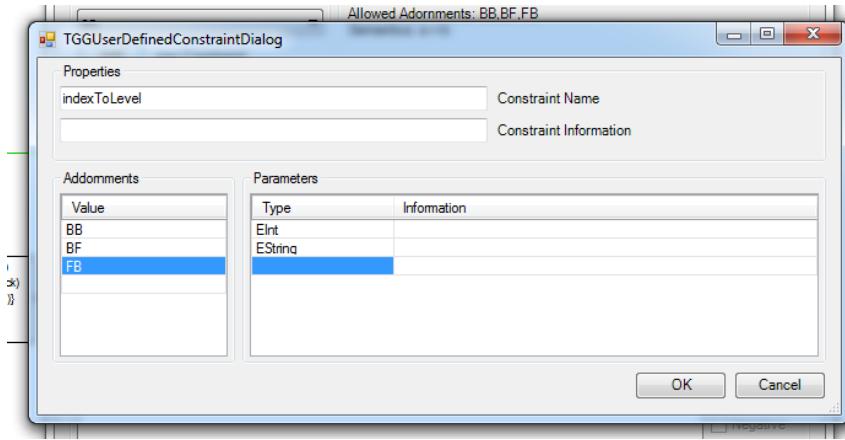


Figure 4.10: Creating an unique constraint

- ▶ 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`.

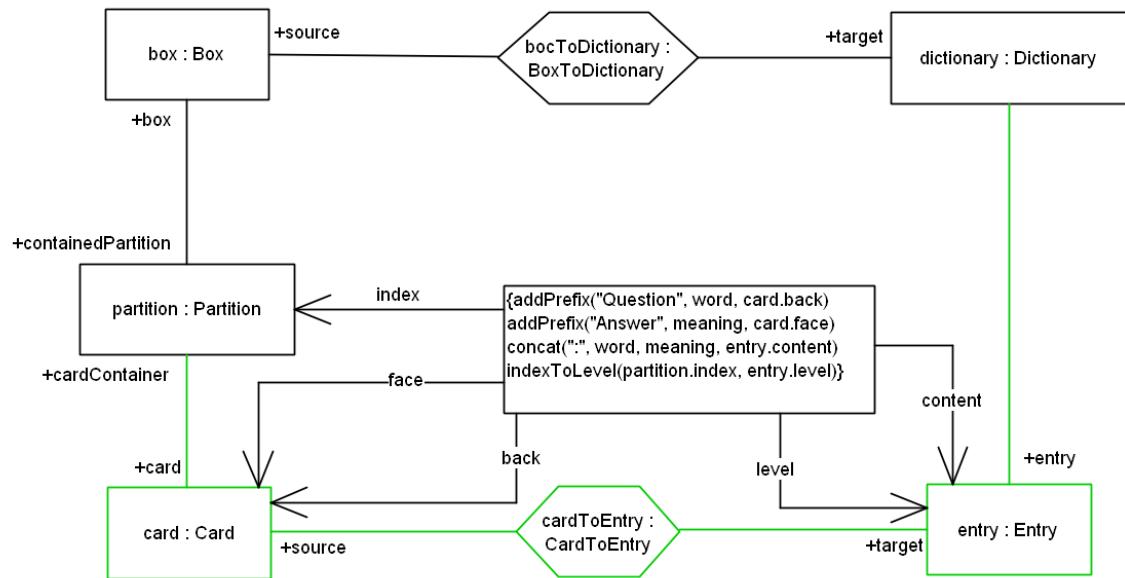


Figure 4.11: `CardToEntryRule` with complete attribute manipulation

## 4.4 Textual TGG Rules

Rules in the textual 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.

```

schema.sch BoToDictionaryRule.tgg
rule BoxToDictionaryRule {
 source {
 ++ box : Box
 }

 correspondence {}

 target {
 ++ dictionary : Dictionary
 }

 constraints {}
}

```

Figure 4.12: Creating source and target objects

- ▶ Now we can create our first TGG correspondence link! In the **correspondence** 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.

```
schema.sch BoxToDictionaryRule.tgg
rule BoxToDictionaryRule {
 source {
 + box : Box
 }

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

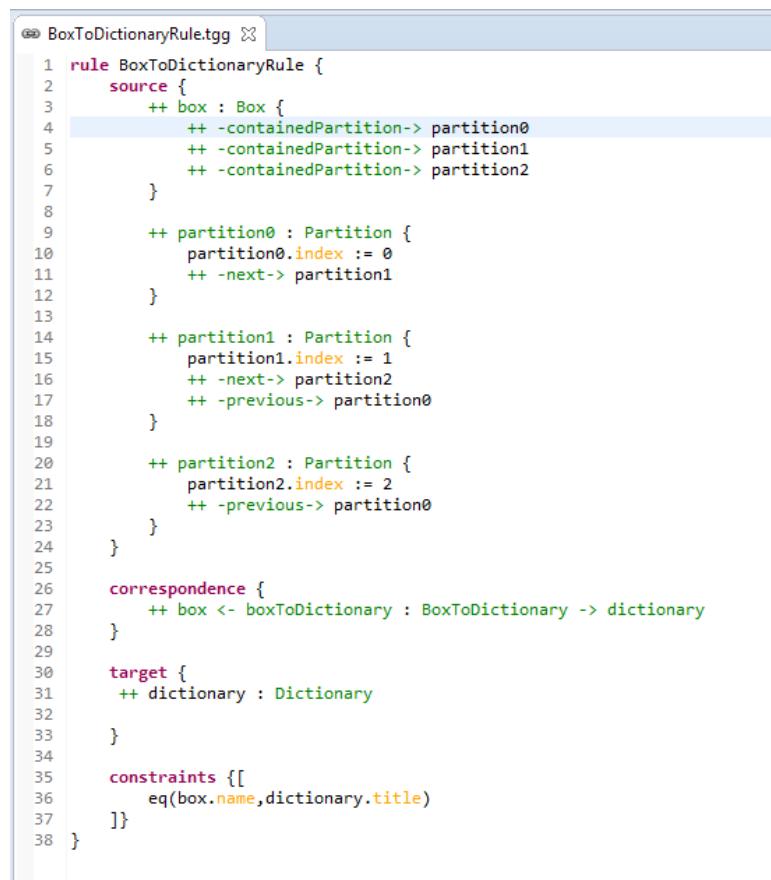
 target {
 + dictionary : Dictionary
 }

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

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.



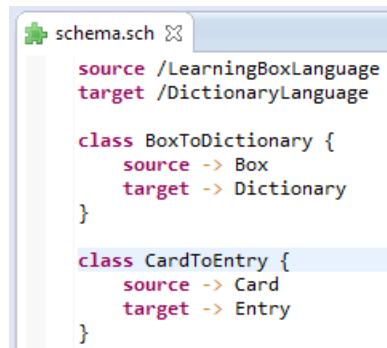
```
BoxToDictionaryRule.tgg
1 rule BoxToDictionaryRule {
2 source {
3 ++ box : Box {
4 ++ -containedPartition-> partition0
5 ++ -containedPartition-> partition1
6 ++ -containedPartition-> partition2
7 }
8
9 ++ partition0 : Partition {
10 partition0.index := 0
11 ++ -next-> partition1
12 }
13
14 ++ partition1 : Partition {
15 partition1.index := 1
16 ++ -next-> partition2
17 ++ -previous-> partition0
18 }
19
20 ++ partition2 : Partition {
21 partition2.index := 2
22 ++ -previous-> partition0
23 }
24
25 correspondence {
26 ++ box <- boxToDictionary : BoxToDictionary -> dictionary
27 }
28
29 target {
30 ++ dictionary : Dictionary
31 }
32
33 constraints {[
34 eq(box.name,dictionary.title)
35]}
```

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.



```

schema.sch ✘
source /LearningBoxLanguage
target /DictionaryLanguage

class BoxToDictionary {
 source -> Box
 target -> Dictionary
}

class CardToEntry {
 source -> Card
 target -> Entry
}

```

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.

```

rule CardToEntryRule {
 source {
 box : Box
 partition0 : Partition
 ++ card : Card
 }
 correspondence {
 }
 target {
 }
 constraints {[
}

```

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.<sup>10</sup>

---

<sup>10</sup>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`.

```

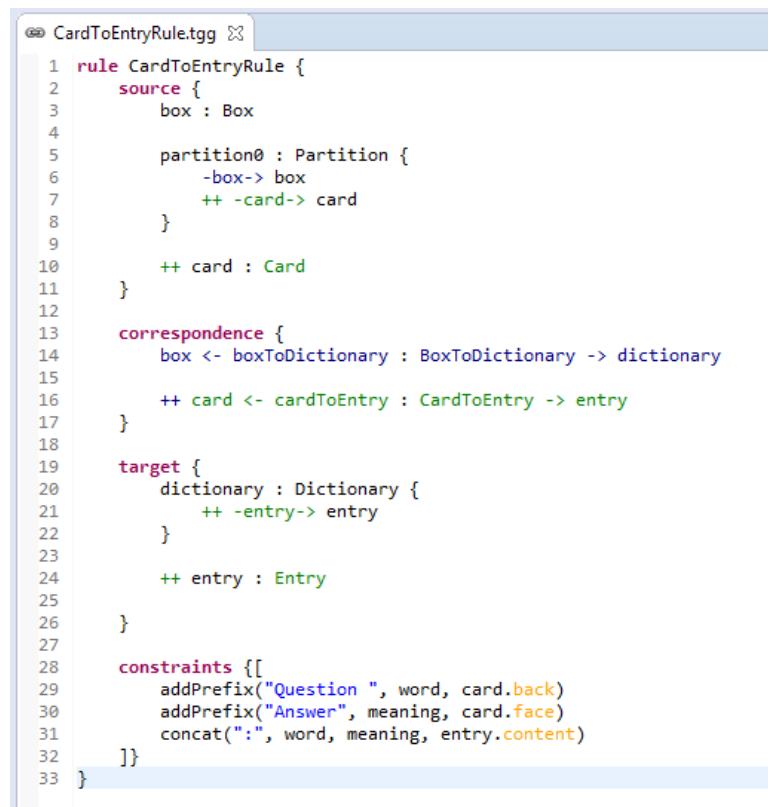
CardToEntryRule.tgg
1 rule CardToEntryRule {
2 source {
3 box : Box
4
5 partition0 : Partition {
6 -box-> box
7 ++ -card-> card
8 }
9
10 ++ card : Card
11 }
12
13 correspondence {
14 box <- boxToDictionary : BoxToDictionary -> dictionary
15
16 ++ card <- cardToEntry : CardToEntry -> entry
17 }
18
19 target {
20 dictionary : Dictionary {
21 ++ -entry-> entry
22 }
23
24 ++ entry : Entry
25 }
26
27
28 constraints {[]
29
30]}
31 }

```

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.



The screenshot shows a code editor window titled "CardToEntryRule.tgg". The code is written in a textual transformation language (TGG) and defines a rule named "CardToEntryRule". The rule has three sections: source, correspondence, and target. The source section defines a "box" element. The correspondence section maps "box" to "dictionary" via "boxToDictionary" and "card" to "entry" via "cardToEntry". The target section defines a "dictionary" element. The constraints section adds prefixes for "Question" and "Answer" and concatenates them with "word" and "meaning" to form "entry.content".

```
@@ CardToEntryRule.tgg
1 rule CardToEntryRule {
2 source {
3 box : Box
4
5 partition0 : Partition {
6 -box-> box
7 ++ -card-> card
8 }
9
10 ++ card : Card
11 }
12
13 correspondence {
14 box <- boxToDictionary : BoxToDictionary -> dictionary
15
16 ++ card <- cardToEntry : CardToEntry -> entry
17 }
18
19 target {
20 dictionary : Dictionary {
21 ++ -entry-> entry
22 }
23
24 ++ entry : Entry
25 }
26
27 constraints {[[
28 addPrefix("Question ", word, card.back)
29 addPrefix("Answer", meaning, card.face)
30 concat(":", word, meaning, entry.content)
31]]}
32
33 }
```

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 ✘
1 rule CardToEntryRule {
2 source {
3 box : Box
4
5 partition0 : Partition {
6 -box-> box
7 ++ -card-> card
8 }
9
10 ++ card : Card
11 }
12
13 correspondence {
14 box <- boxToDictionary : BoxToDictionary -> dictionary
15
16 ++ card <- cardToEntry : CardToEntry -> entry
17 }
18
19 target {
20 dictionary : Dictionary {
21 ++ -entry-> entry
22 }
23
24 ++ entry : Entry
25
26 }
27
28 constraints {
29 addPrefix("Question ", word, card.back)
30 addPrefix("Answer", meaning, card.face)
31 concat(":", word, meaning, entry.content)
32
33 indexToLevel[BB,BF,FB](EInt,EString)
34 indexToLevel(partition0.index,entry.level)
35 }
36 }
```

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.

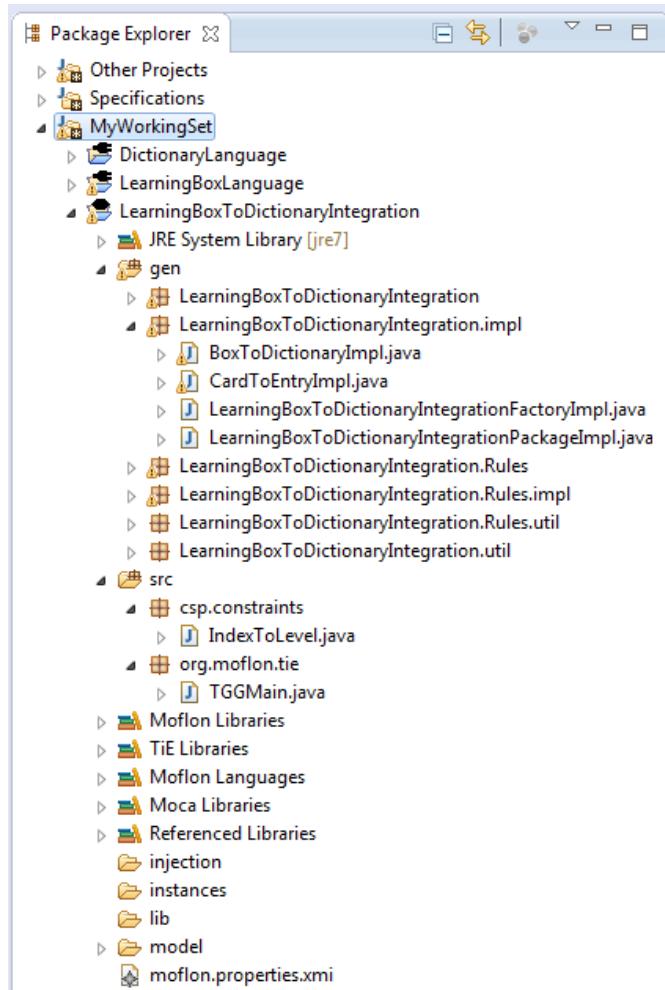


Figure 4.20: Files generated from your TGG

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.<sup>11</sup>

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.

---

<sup>11</sup> 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.

---

```

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<sup>12</sup> 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).

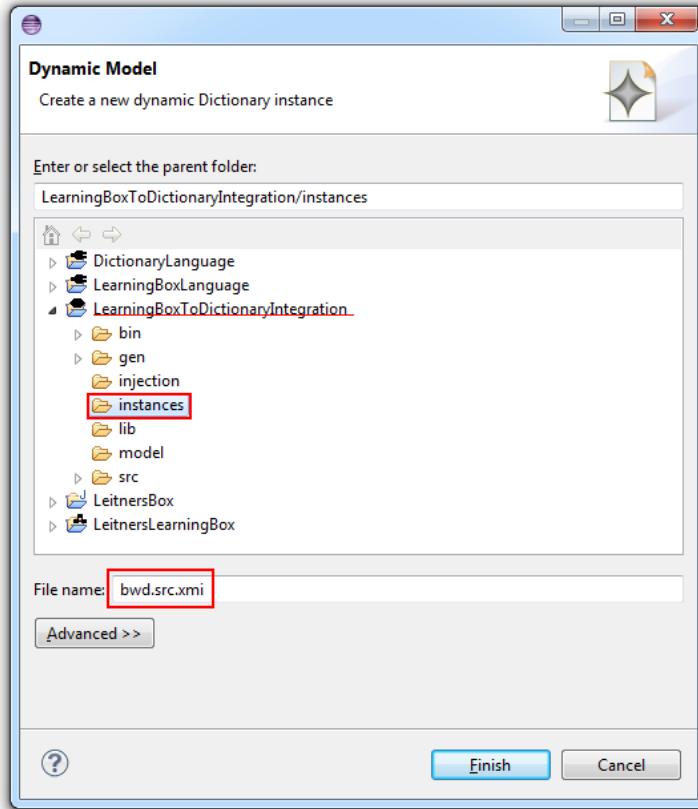


Figure 5.1: Create a dynamic instance of `Dictionary`

---

<sup>12</sup>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 different difficulty level, e.g., **beginner** for **One:Eins**, **advanced** for **Two:Zwei**, and **master** for **Three:Drei**. Your instance should resemble Fig. 5.2.

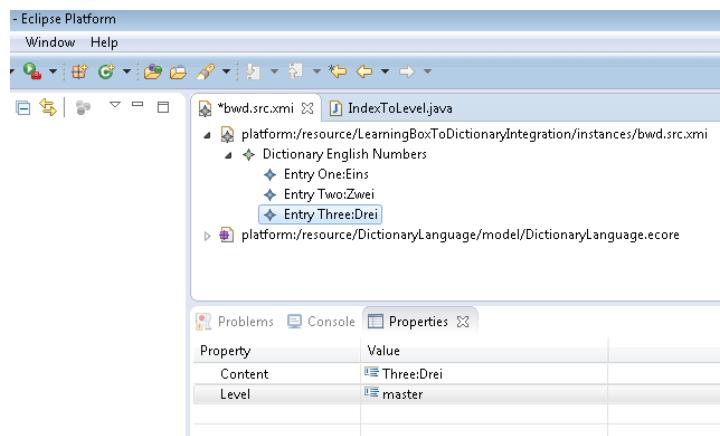


Figure 5.2: Fill a **Dictionary** for the transformation

- ▶ Let's check out the file that will actually execute our transformation. Navigate to “LearningBoxToDictionaryIntegration/src/org.moflon.tie” and click to open **LearningBoxToDictionaryIntegration-Trafo.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.

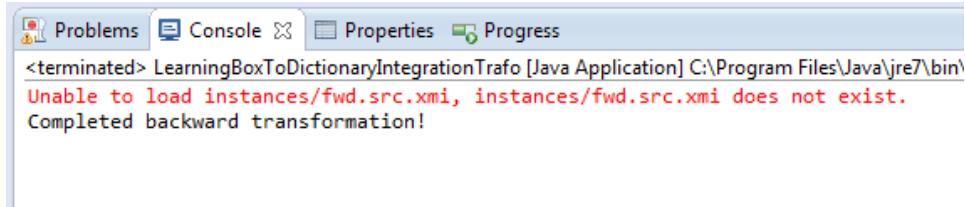


Figure 5.3: Running the backward transformation

- ▶ 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.

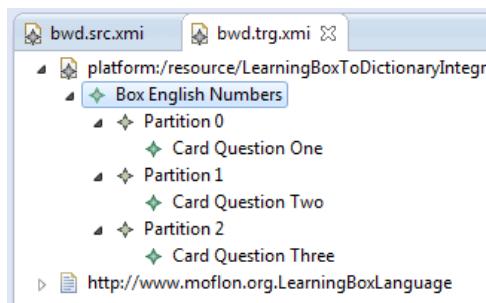


Figure 5.4: Result of the *backwards* transformation

- ▶ 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.<sup>13</sup> 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.

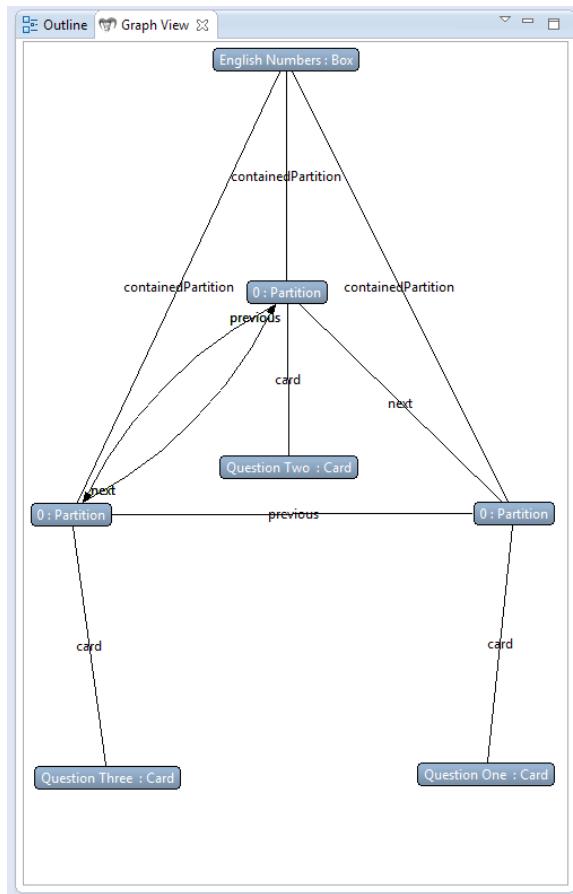


Figure 5.5: Confirm the transformation with the Graph Viewer

- 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`.

---

<sup>13</sup>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,<sup>14</sup> 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.

While building this so-called *ignore rule*, keep in mind that the goal is to *ignore rule* 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.

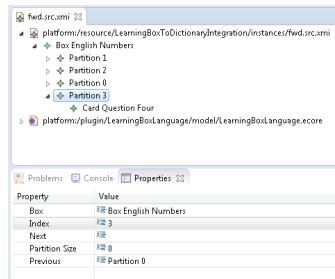


Figure 6.1: Extended **fwd/src.xmi**

<sup>14</sup>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.

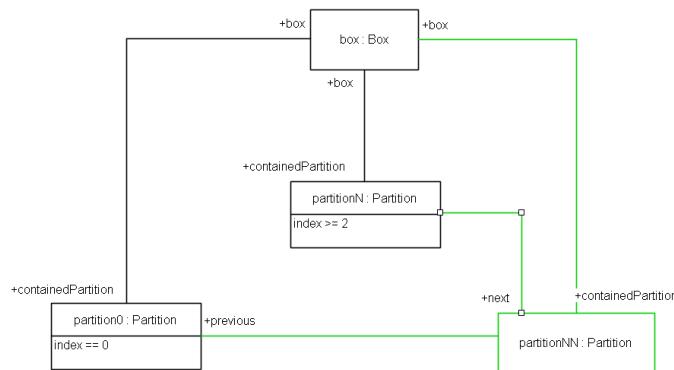
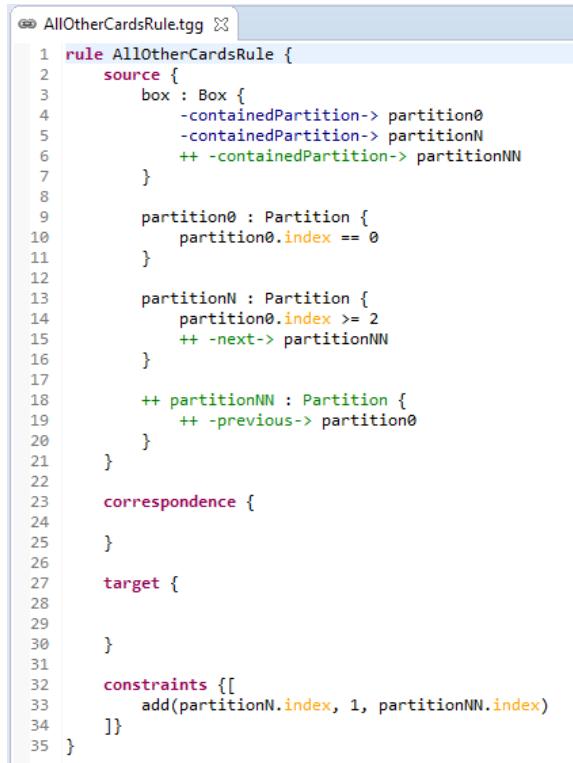


Figure 6.2: The completed `AllOtherPartitionsRule`

- ▶ 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
1 rule AllOtherCardsRule {
2 source {
3 box : Box {
4 -containedPartition-> partition0
5 -containedPartition-> partitionN
6 ++ -containedPartition-> partitionNN
7 }
8
9 partition0 : Partition {
10 partition0.index == 0
11 }
12
13 partitionN : Partition {
14 partition0.index >= 2
15 ++ -next-> partitionNN
16 }
17
18 ++ partitionNN : Partition {
19 ++ -previous-> partition0
20 }
21 }
22
23 correspondence {
24 }
25
26
27 target {
28
29 }
30
31 constraints {
32 add(partitionN.index, 1, partitionNN.index)
33 }
34 }
35 }
```

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 **n**th, 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 **Entries** (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 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`.<sup>15</sup>
- ▶ Run `LearningBoxToDictionaryIntegrationTrafo.java` and inspect the resulting `bwd.trg.xmi` (Fig. 7.1). Unfortunately, the newest `partition3` is missing!

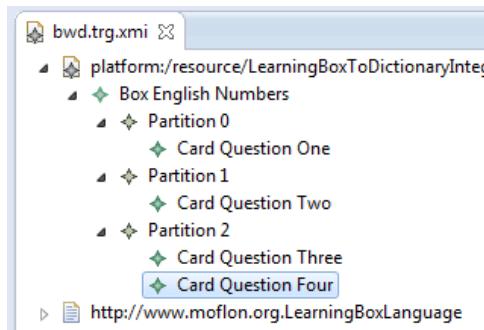


Figure 7.1: The transformation loses data for any index greater than 2

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

---

<sup>15</sup>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.

```
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.

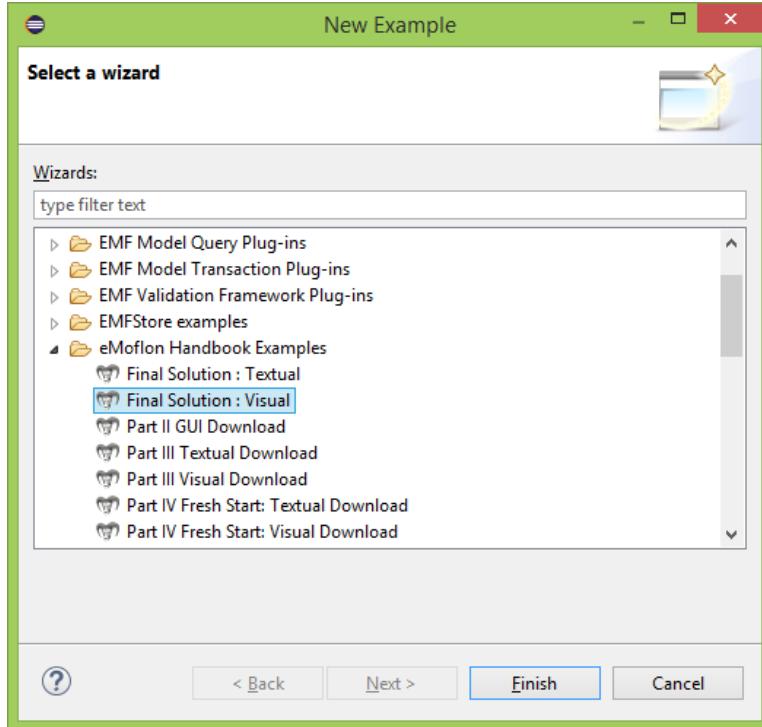


Figure 8.1: Get the final visual solution

- ▶ 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-**

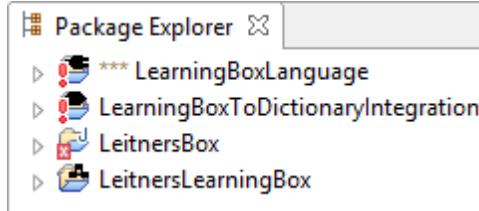


Figure 8.2: Example projects from the wizard

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.

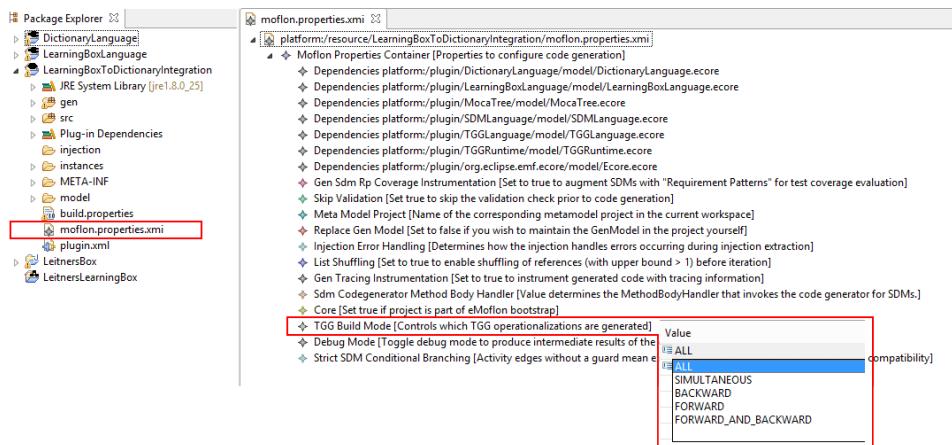


Figure 8.3: Screenshot of the **moflon.properties.xmi**

- 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 **MaxModelErrorSizeController** class to terminate the generation process if a specific model size has been reached. The **RuleSelector** controls

---

```

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 **generated-Models** 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 - 20/6 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.

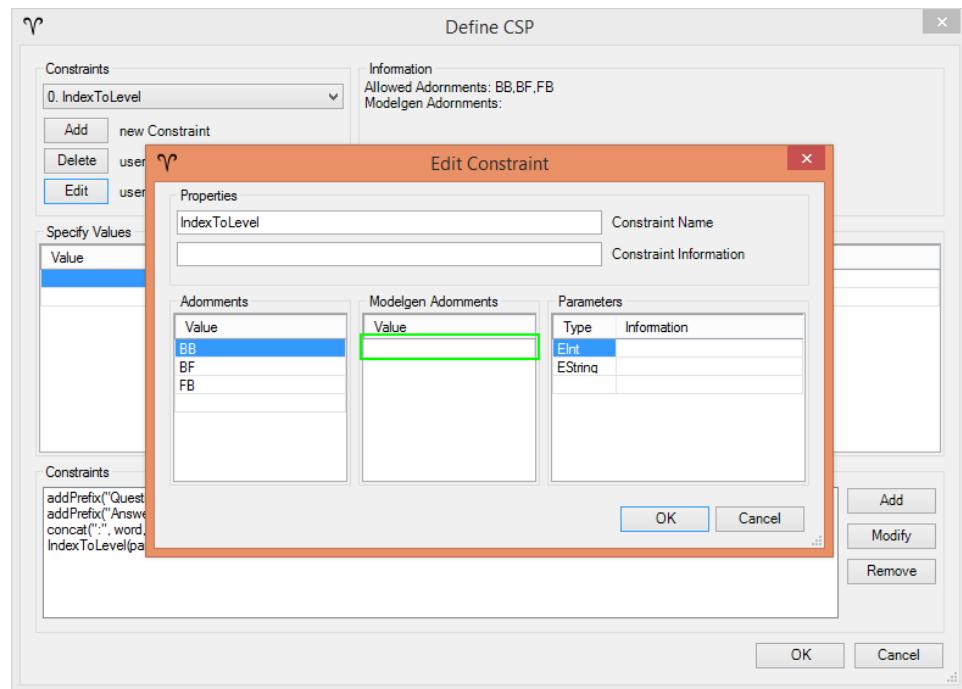


Figure 8.6: Custom constraint dialog

```
// 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.

# An Introduction to Metamodelling and Graph Transformations

---

*with eMoflon*



---

## Part V: Model-to-text Transformations

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

For further information contact us at [contact@emoflon.org](mailto:contact@emoflon.org).

*The eMoflon team*  
Darmstadt, Germany (August 2015)

# Contents

|   |                                                  |    |
|---|--------------------------------------------------|----|
| 1 | Setting up your workspace . . . . .              | 4  |
| 2 | Text-to-tree transformation . . . . .            | 15 |
| 3 | Tree-to-model transformation with TGGs . . . . . | 24 |
| 4 | Tree-to-text transformation . . . . .            | 46 |
| 5 | Conclusion and next steps . . . . .              | 50 |
|   | Glossary . . . . .                               | 51 |

## Part V:

# Model-to-Text Transformations

Approximate time to complete: 1h 30min

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

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

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

*Bidirectional Transformation*

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

checking or guaranteeing consistency, adding or abstracting from platform details, . . . ).

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

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

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

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

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

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

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

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

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

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

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

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

## 1 Setting up your workspace

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

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

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

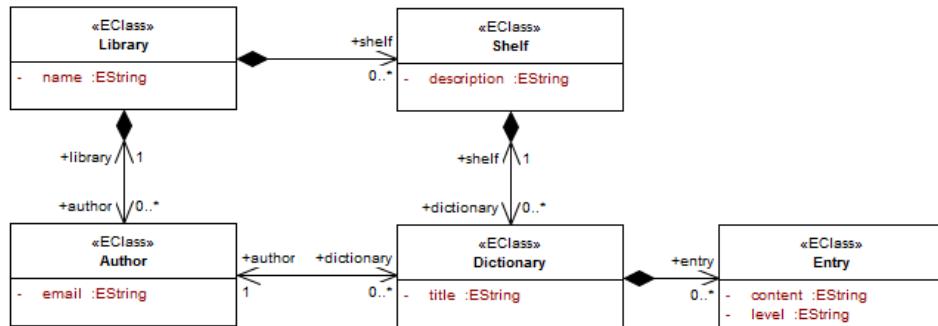


Figure 1.1: Metamodel for dictionaries (visual concrete syntax)

---

<sup>1</sup>For simple cases, *regular expressions* can also be used

```

C Library.eclass ✎
1 class Library {
2
3 name : EString
4
5 <>-> author(0..*) : Author
6 <>-> shelf(0..*) : Shelf
7 }

C Shelf.eclass ✎
1 class Shelf {
2
3 description : EString
4
5 <>-> dictionary(0..*) : Dictionary
6 }

C Author.eclass ✎
1 class Author {
2
3 email : EString
4
5 -> library(1..1) : Library
6 -> dictionary(0..*) : Dictionary
7 }

C Dictionary.eclass ✎
1 class Dictionary {
2
3 title : EString
4
5 <>-> entry(0..*) : Entry
6
7 -> shelf(1..1) : Shelf
8 -> author(1..1) : Author
9 }

C Entry.eclass ✎
1 class Entry {
2
3 content : EString
4 level : EString
5 }

C _constraints.mconf ✎
1 opposites {
2
3 dictionary : Shelf <-> shelf : Dictionary
4
5 author : Library <-> library : Author
6
7 dictionary : Author <-> author : Dictionary
8 }

```

Figure 1.2: Metamodel for dictionaries (textual concrete syntax)

### Option 1: Import a complete cheat package

- ▶ Import the Part V ‘cheat package’ by selecting “New” in the toolbar, and the cheat package in the concrete syntax of your choice (Fig. 1.3).

### Option 2: Continue with the workspace from Part IV

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

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

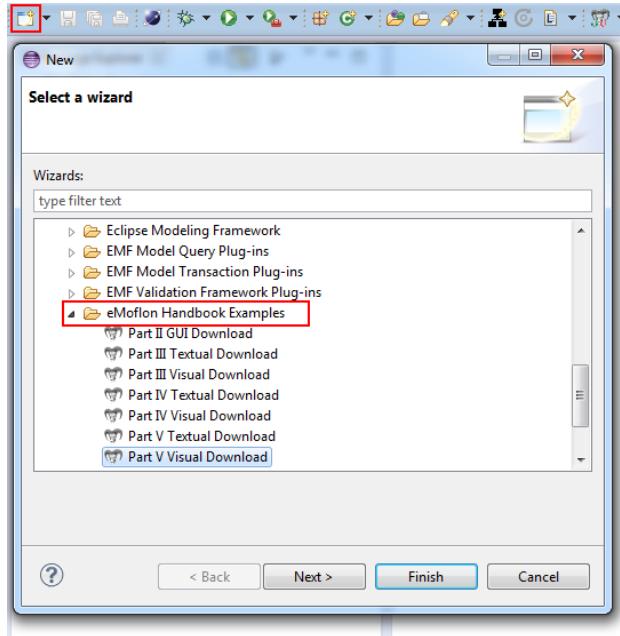


Figure 1.3: Load the cheat package for Part V into your workspace

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

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

---

<sup>2</sup>MOCA stands for Moflon Code Adapter (not coffee, sorry.)

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

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

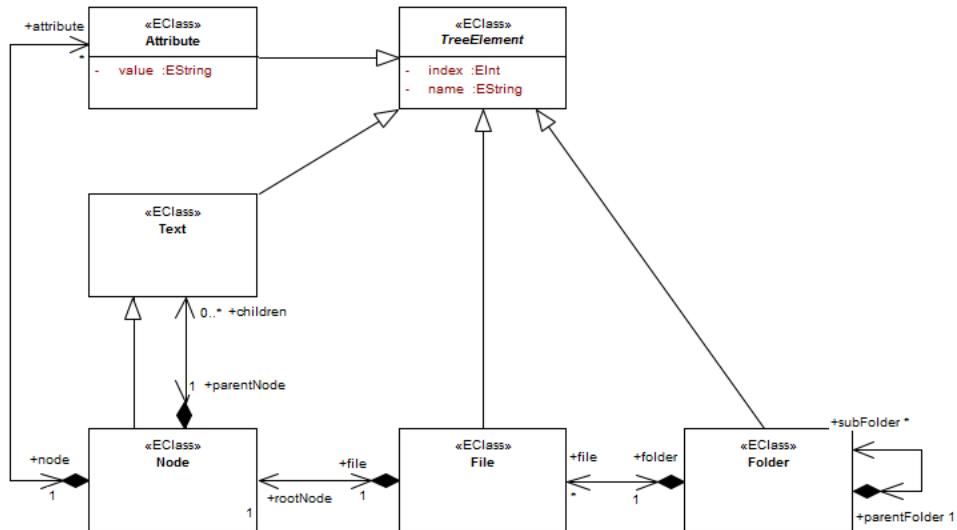


Figure 1.4: Visual depiction of the MocaTree metamodel

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

## 1.1 First steps

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

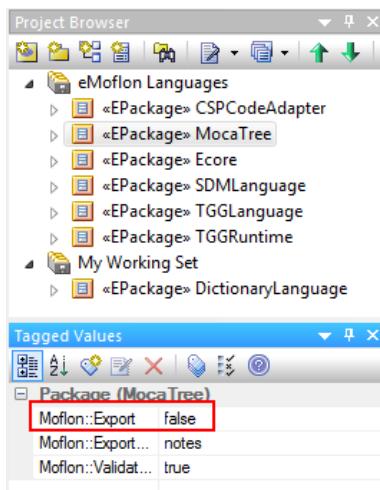


Figure 1.5: `MocaTree` is one of eMoflon’s internal metamodels

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

- ▶ Despite `DictionaryLanguage` being contained in a different working set than `MocaTree`, the two metamodels are contained within the same EA project (EAP) which means you are able to create a new TGG using them both. Add a new package to `My Working Set` named `DictionaryCodeAdapter`.
- ▶ Select the package and add a new TGG schema diagram as depicted in Fig. 1.6. In the next dialogue window, set the source project as `MocaTree`, and the target project as `DictionaryLanguage`.

---

<sup>4</sup>The “Tagged Values” window can be opened by going to “View/Tagged Values” or by hovering over the `Tagged Values` tab immediately to the right of the project browser.

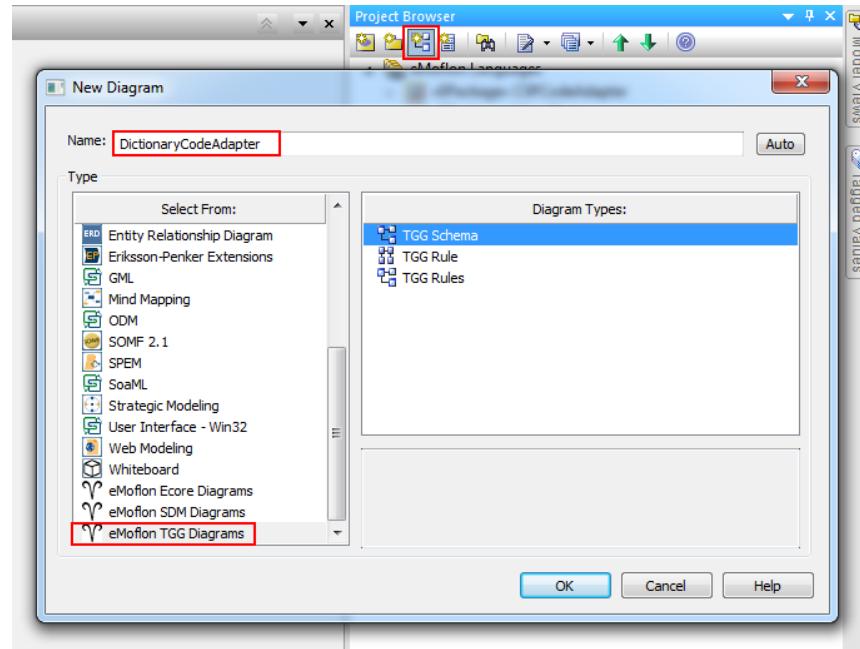


Figure 1.6: Create a new TGG schema diagram

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

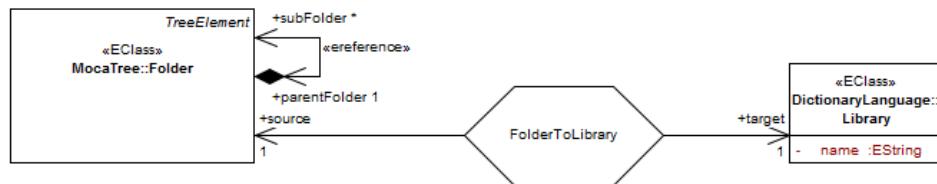


Figure 1.7: The first correspondence type for the transformation

---

<sup>5</sup>For details on the correspondence metamodel and how to create types, refer to Part IV, Section 3.

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

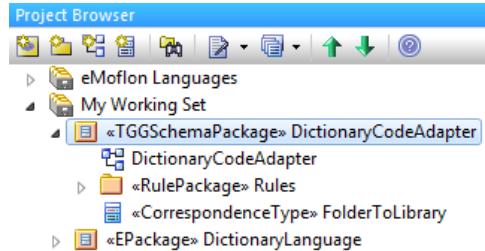


Figure 1.8: A fully prepared TGG project

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

---

<sup>6</sup>Activate via “Extensions/Add-in Windows”

## 1.2 First steps

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

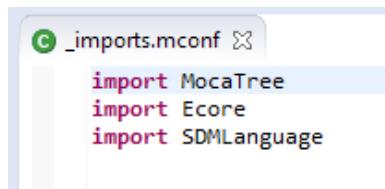


Figure 1.9: Dictionary's imports file

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

---

<sup>7</sup>For details on this correspondence metamodel and how to create types, refer to Part IV, Section 3.

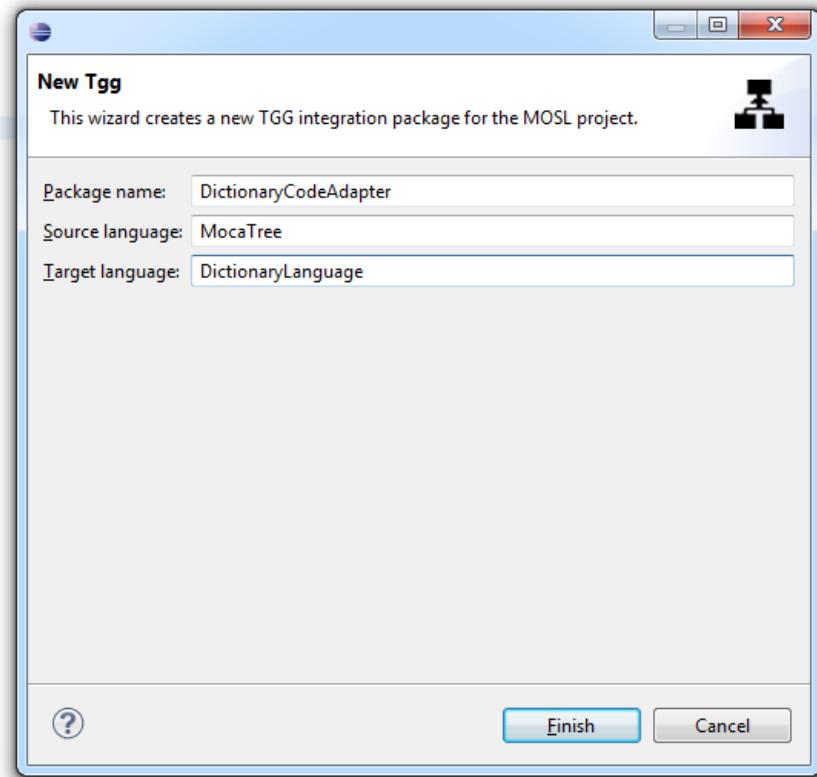


Figure 1.10: Settings for our TGG

The screenshot shows a code editor window with the file name "schema.sch". The content of the editor is a correspondence type definition:

```
1 source /MocaTree
2 target /DictionaryLanguage
3
4 class FolderToLibrary {
5 source -> Folder
6 target -> Library
7 }
8
```

Figure 1.11: Our first correspondence type

### 1.3 Setting up the parser

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

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

- ▶ Right-click on the generated `DictionaryCodeAdapter` folder and navigate to “eMoflon/ Add Parser/Unparser” (Fig 1.12).<sup>8</sup>

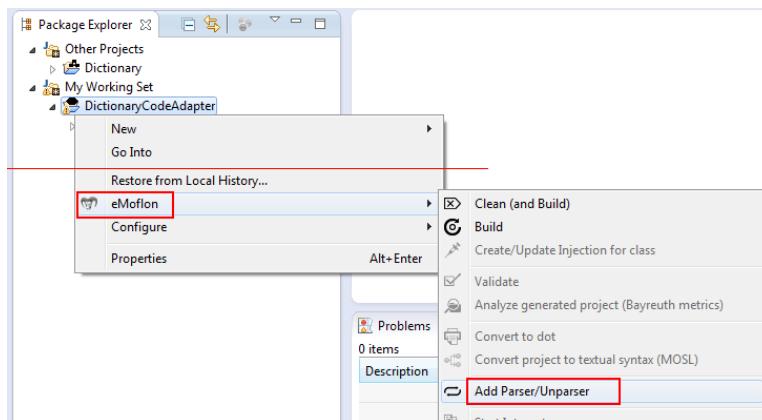


Figure 1.12: Adding a new parser/unparser to a project

- ▶ In the parser settings window, enter `dictionary` as the `File extension`, and confirm that the `Create Parser`, `Create Unparser`, and `ANTLR` options are selected as the corresponding technology for each case (Fig 1.13). Affirm by pressing `Finish`.
- ▶ If everything executed without error, parser and unparser stubs should be generated in the `src` package (Fig. 1.14). In addition, a new `in` folder should appear under `instances`.

---

<sup>8</sup>For presentation purposes, this context menu screenshot has been edited

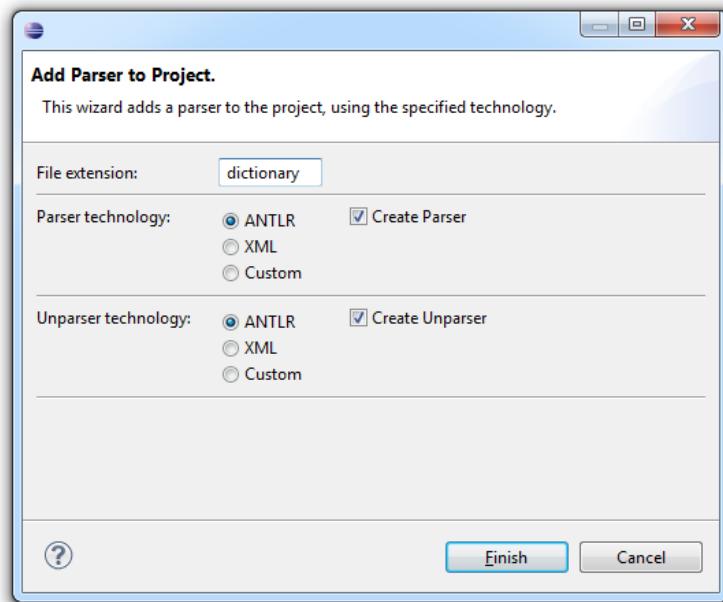


Figure 1.13: Parser/unparser settings

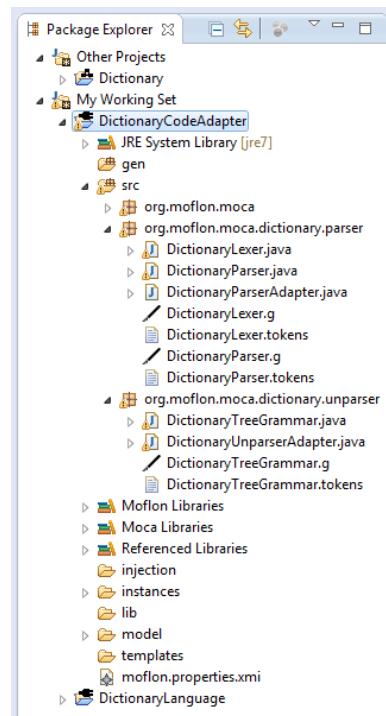


Figure 1.14: Generated stubs and derived files

## 2 Text-to-tree transformation

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

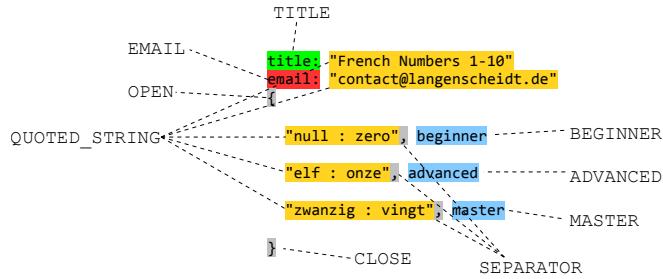
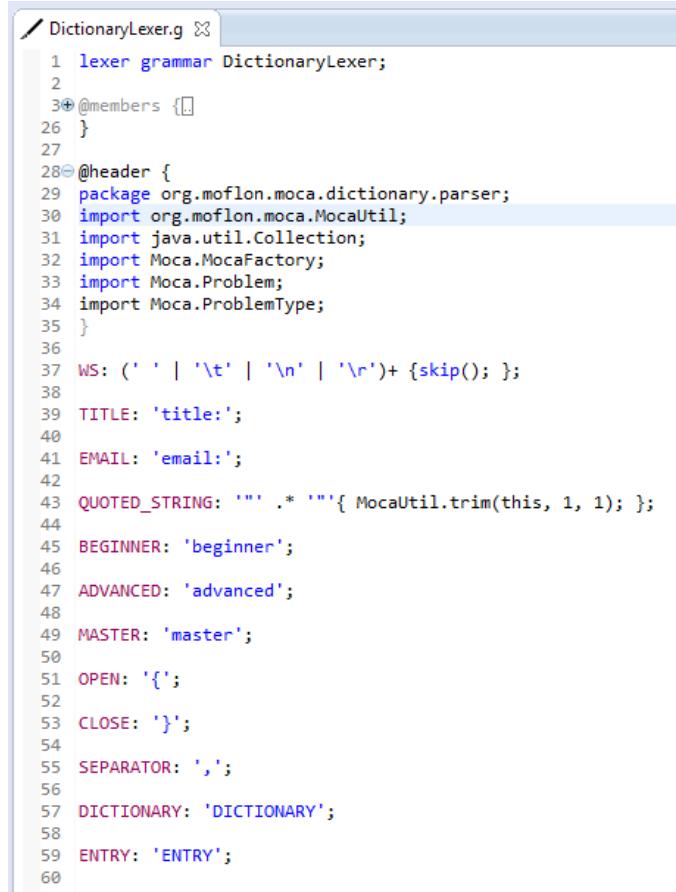


Figure 2.1: Identified tokens in a dictionary file

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

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

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



```

1 lexer grammar DictionaryLexer;
2
3@members {..}
26 }
27
28@header {
29 package org.moflon.moca.dictionary.parser;
30 import org.moflon.moca.MocaUtil;
31 import java.util.Collection;
32 import Moca.MocaFactory;
33 import Moca.Problem;
34 import Moca.ProblemType;
35 }
36
37 WS: (' ' | '\t' | '\n' | '\r')+ {skip(); };
38
39 TITLE: 'title:';
40
41 EMAIL: 'email:';
42
43 QUOTED_STRING: """ .* """{ MocaUtil.trim(this, 1, 1); };
44
45 BEGINNER: 'beginner';
46
47 ADVANCED: 'advanced';
48
49 MASTER: 'master';
50
51 OPEN: '{';
52
53 CLOSE: '}';
54
55 SEPARATOR: ',';
56
57 DICTIONARY: 'DICTIONARY';
58
59 ENTRY: 'ENTRY';
60

```

Figure 2.2: Lexer grammar

- ▶ Don't forget to add `import org.moflon.moca.MocaUtil` to `@header`. Be vigilant to avoid any typos and mistakes!
- ▶ Save to compile the file, and ensure no errors persist before proceeding.

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

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

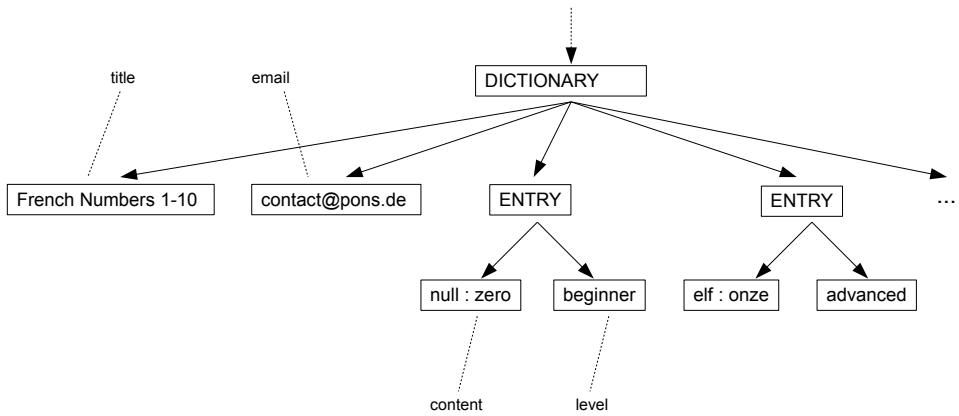


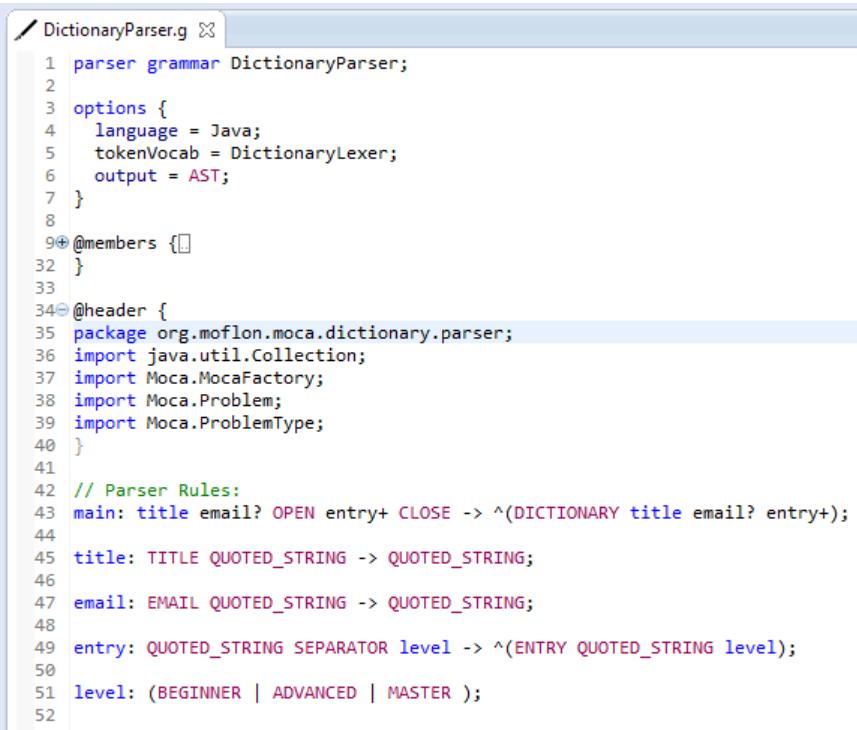
Figure 2.3: Abstract Syntax Tree (AST) of an input token stream

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

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

---

<sup>9</sup>Irrelevant content for our model



The screenshot shows a code editor window with the title bar "DictionaryParser.g". The content of the file is a parser grammar in EBNF-like syntax. The code includes options for Java language, DictionaryLexer token vocab, and AST output. It defines @members, @header, and several parser rules for title, email, entry, and level.

```
1 parser grammar DictionaryParser;
2
3 options {
4 language = Java;
5 tokenVocab = DictionaryLexer;
6 output = AST;
7 }
8
9+ @members {..}
32 }
33
34+ @header {
35 package org.moflon.moca.dictionary.parser;
36 import java.util.Collection;
37 import Moca.MocaFactory;
38 import Moca.Problem;
39 import Moca.ProblemType;
40 }
41
42 // Parser Rules:
43 main: title email? OPEN entry+ CLOSE -> ^(DICTIONARY title email? entry+);
44
45 title: TITLE QUOTED_STRING -> QUOTED_STRING;
46
47 email: EMAIL QUOTED_STRING -> QUOTED_STRING;
48
49 entry: QUOTED_STRING SEPARATOR level -> ^(ENTRY QUOTED_STRING level);
50
51 level: (BEGINNER | ADVANCED | MASTER);
52
```

Figure 2.4: Parser grammar

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

```

14 import DictionaryCodeAdapter.DictionaryCodeAdapterPackage;
15 import DictionaryLanguage.DictionaryLanguagePackage;
16
17 import MocaTree.Folder;
18 import java.io.File;
19
20 public class DictionaryCodeAdapterTrafo extends IntegratorHelper {
21
22 public DictionaryCodeAdapterTrafo() throws IOException {}
23
24 public static void main(String[] args) throws IOException {
25 // Set up logging
26 BasicConfigurator.configure();
27
28 // Text to tree
29 Folder folder = MocaMain.getCodeAdapter().parse(new File("instances/in/myLibrary"));
30 eMoflonEMFUtil.saveModel(folder, "instances/fwd.src.xmi");
31
32 // Forward Transformation
33 DictionaryCodeAdapterTrafo helper = new DictionaryCodeAdapterTrafo();
34 helper.performForward("instances/fwd.src.xmi");
35
36 // Backward Transformation
37 helper = new DictionaryCodeAdapterTrafo();
38 helper.performBackward("instances/bwd.src.xmi");
39
40 // Tree to text
41 MocaMain.getCodeAdapter().unparse("instances/out", (Folder) helper.getSrc());
42 }
43
44 public void performForward(String source) {}
45
46 public void performBackward(String target) {}
47
48 }
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89 }
```

Figure 2.5: Edit `DictionaryCodeAdapterTrafo` to run the transformation

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

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

---

<sup>10</sup>TIE stands for *Tool Integration Environment*

---

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

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

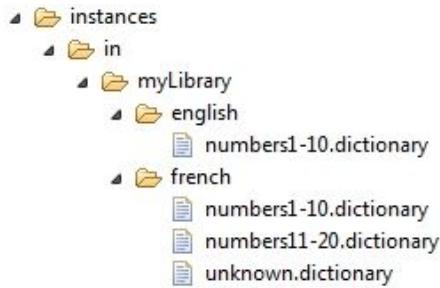


Figure 2.6: Input directory structure

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

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

---

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

```
english/numbers1-10.dictionary:

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

```
french/numbers1-10.dictionary:

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

```
french/numbers11-20.dictionary:

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

```
french/unknown.dictionary:

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

Table 1: Four input .dictionary files

- ▶ Once you have saved each file, right click on `DictionaryCodeAdapterTrafo.java` and navigate to “Run As/Java Application” to run the transformation. Don’t worry about the error messages – they’re related to the unparser which we haven’t implemented yet; You should have received at least one success message indicating your transformation worked.
- ▶ Refresh the `instances` folder. Despite being (mostly) unimplemented, the transformation still partly completed, generating several files in the process (Fig. 2.7).

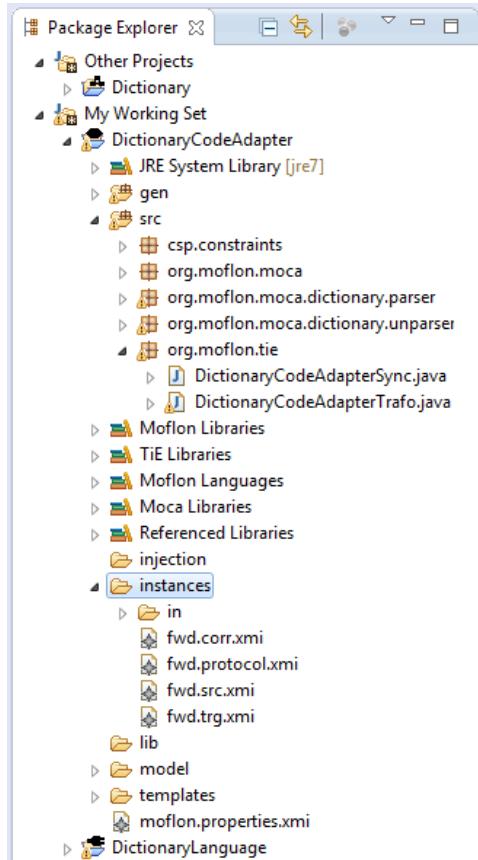


Figure 2.7: Result of the first TGG execution

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

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

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

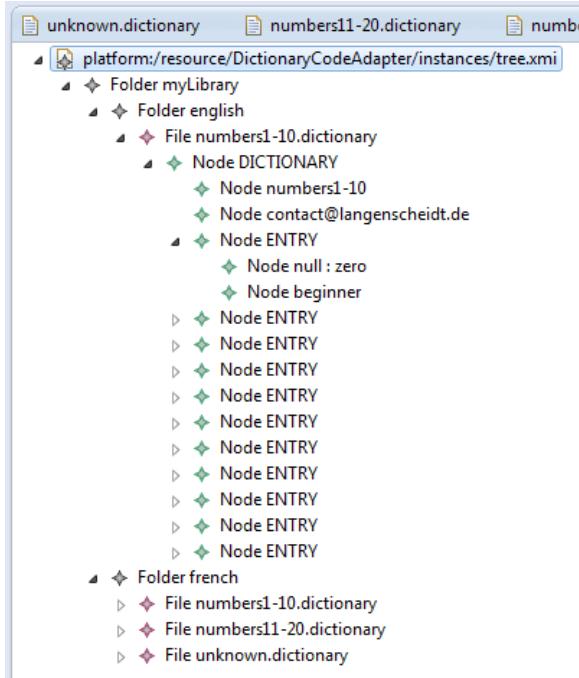


Figure 2.8: A MocaTree created by the framework using our parser

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

---

<sup>12</sup>Refer to Fig 1.4 for the metamodel of this structure

### 3 Tree-to-model transformation with TGGs

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

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

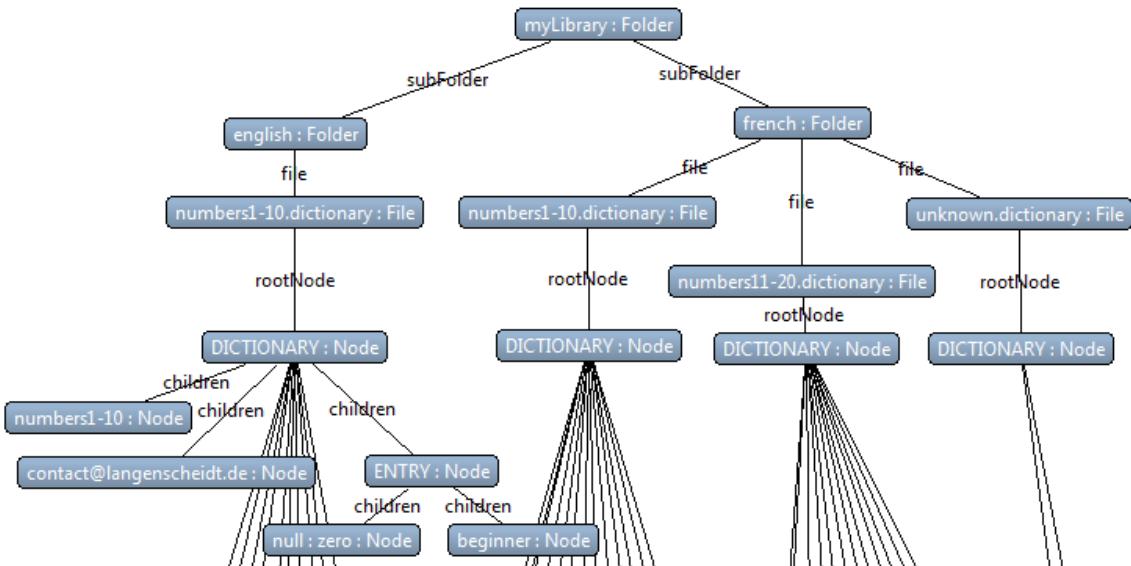


Figure 3.1: The `MocaTree` model of our input directory

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

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

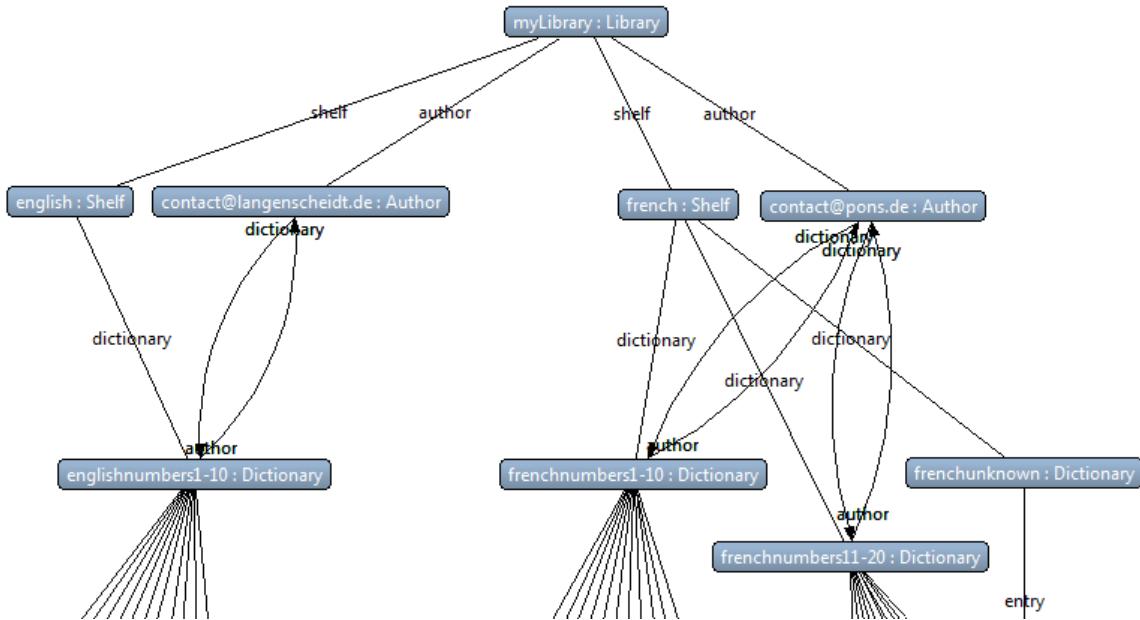


Figure 3.2: The final **Dictionary** target model

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

### 3.1 The visual transformation rules

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

#### FolderToLibraryRule

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

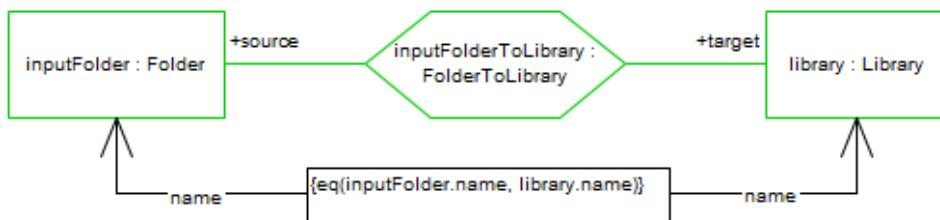


Figure 3.3: FolderToLibraryRule

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

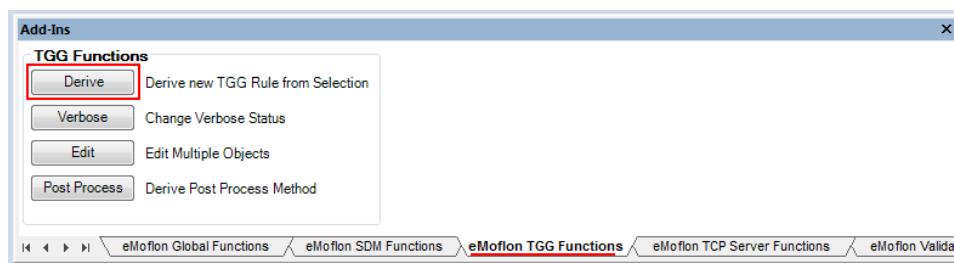


Figure 3.4: Deriving a new rule with eMoflon's control panel

### ForAllShelfRule

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

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

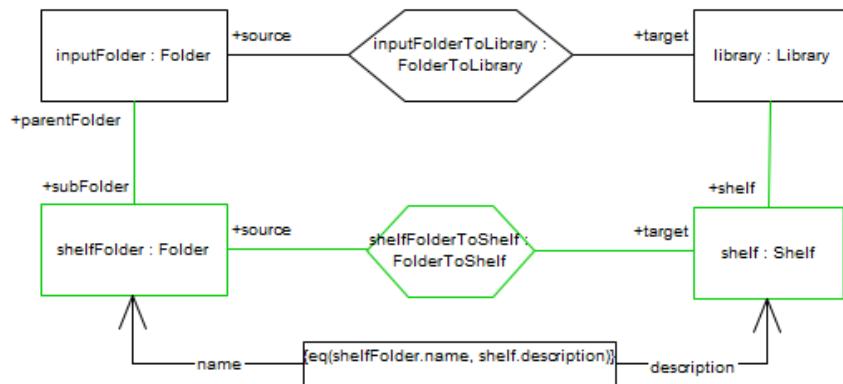


Figure 3.5: ForAllShelfRule

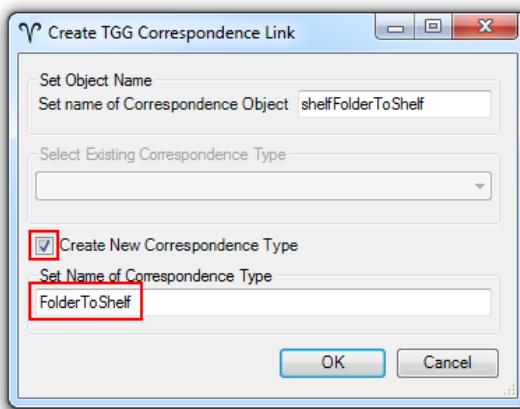


Figure 3.6: Creating a `FolderToShelf` correspondence type on-the-fly

### NodeToDictionaryRule

- With the container `shelf` now assumed to exist, we are ready to handle dictionary `File` elements. Analogously to how you began the previous rule, select `shelfFolder`, `shelfFolderToShelf`, and `shelf`, and derive `NodeToDictionaryRule` with this context.
- Complete it as depicted in Fig. 3.7. As you can see, this rule creates a `dictionaryNode` and its equivalent `dictionary`, and handles the first node in the tree structure. Nearly every element is used to correctly set the `dictionary` and `dictionaryFile` names in two constraints.

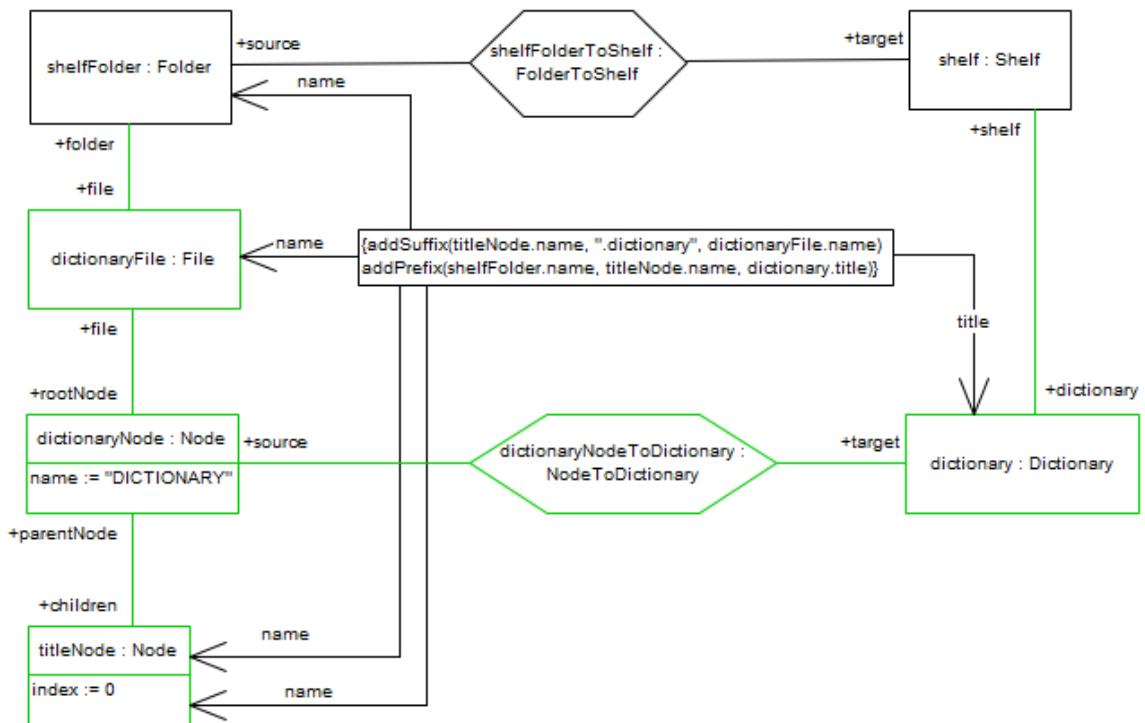


Figure 3.7: NodeToDictionaryRule

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

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

### ForAllEntryRule

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

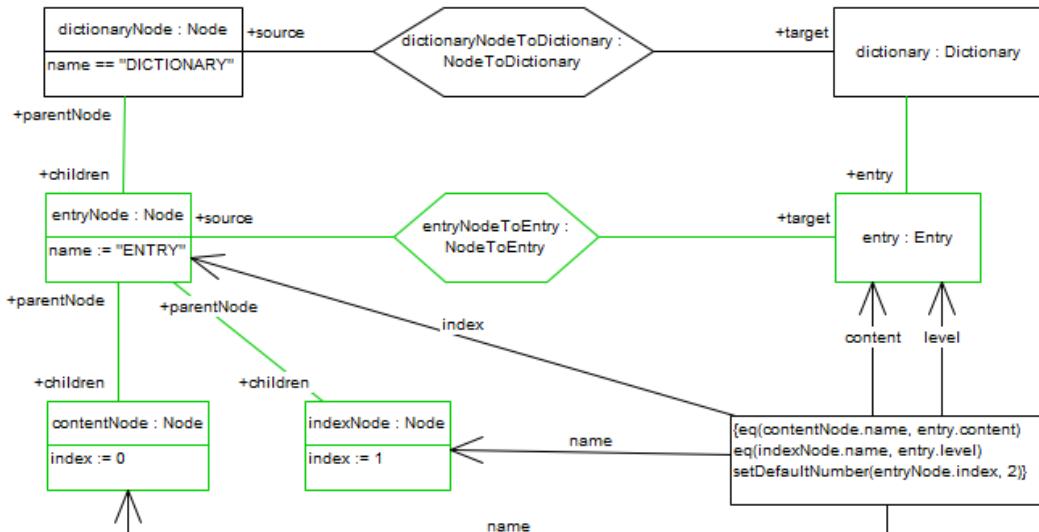


Figure 3.8: `ForAllEntryRule`

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

The final constraint however, is one we haven't used before. If you re-examine your source `tree.xmi` model, you'll notice that every entry has a different `index` value. This prevented us from setting an attribute constraint on `entryNode` but, as long as its index wasn't 0 indicating a `titleNode` (as constrained in the previous `NodeToDictionaryRule`), it didn't matter. Unfortunately, this missing infor-

mation means any new `entryNodes` created in the backward transformation have a default 0 index value, and *could* be mistaken by the rule. By using `setDefaultNumber`, we have declared that any created default `index` attributes must be set to 2.

### AuthorRule

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

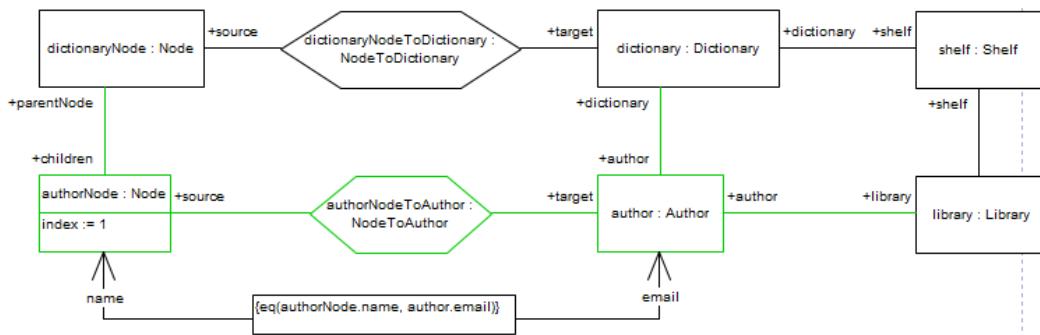


Figure 3.9: AuthorRule

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

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

eMoflon's visual syntax has a cool *refinement* feature which enables you to adjust specific elements in a rule, without having to redraw an entire

diagram exactly as before, save for one or two minor differences. Given that we want rules to handle either *always* creating an `author`, or checking for an existing one first, (both which will be identical to `AuthorRule` except for the binding and reference links on the matched `authorNode`), this feature is exactly what we need.

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

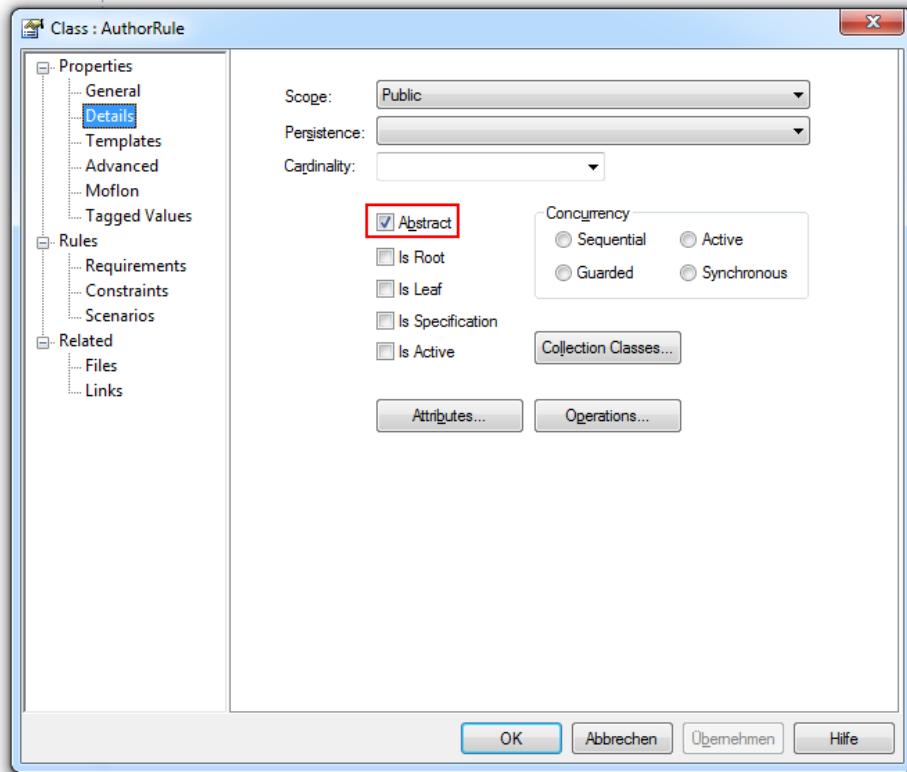


Figure 3.10: Declaring `AuthorRule` as abstract

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

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

### ExistingAuthorRule

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



Figure 3.11: Deriving a refinement rule

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



Figure 3.12: ExistingAuthorRule

- If you’re having difficulty visualising the entire rule with this minor modification, note that EA allows you to drag and drop all elements

from the basis rule as links so they're represented in the diagram. Figure 3.13 represents the entire `ExistingAuthorRule`. This how the rule would have looked without using rule refinement.

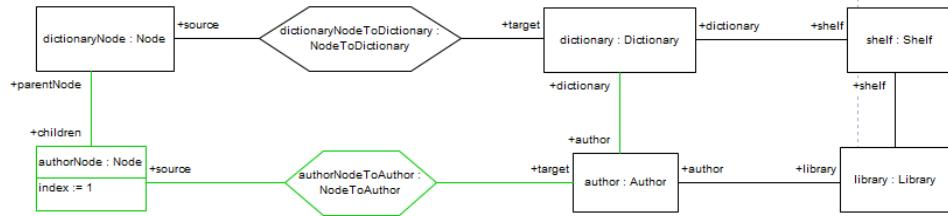


Figure 3.13: `ExistingAuthorRule` (flattened)

### NewAuthorRule

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



Figure 3.14: Completed `NewAuthorRule`

- Of course, we could have left `AuthorRule` concrete and used just two rules, but having an explicit abstract rule, with two concrete implementations of the possibilities, is clearer.
- Return to the `Rules` diagram one last time. In order to ensure the new rules refine `AuthorRule`, quick-link from each to the root rule, choosing `Create Refinement Link` from the context menu. Your diagram should now resemble Fig. 3.15.
- You're nearly done! Make sure everything is saved, and validate your TGG.

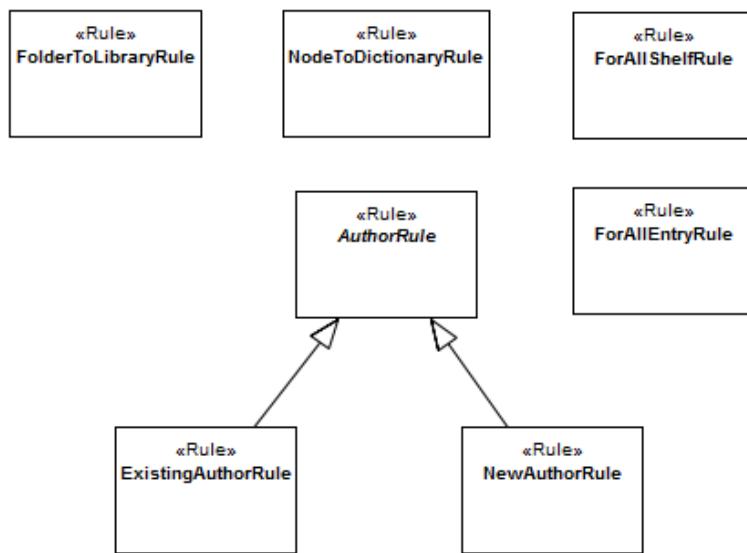


Figure 3.15: Final Rules diagram

▷ Next

### 3.2 The textual transformation rules

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

#### FolderToLibraryRule

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



```

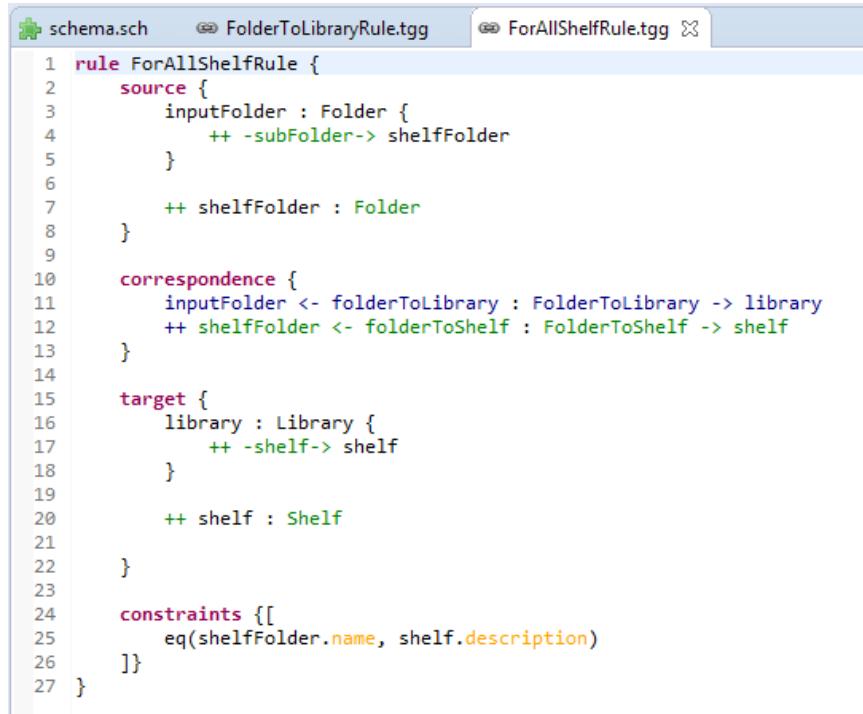
schema.sch FolderToLibraryRule.tgg

1 rule FolderToLibraryRule {
2 source {
3 ++ inputFolder : Folder
4 }
5
6 correspondence {
7 ++ inputFolder <- folderToLibrary : FolderToLibrary -> library
8 }
9
10 target {
11 ++ library : Library
12 }
13
14 constraints {[
15 eq(inputFolder.name, library.name)
16]}
17 }
18 }
```

Figure 3.16: The TGG transformation begins with this rule.

#### ForAllShelfRule

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

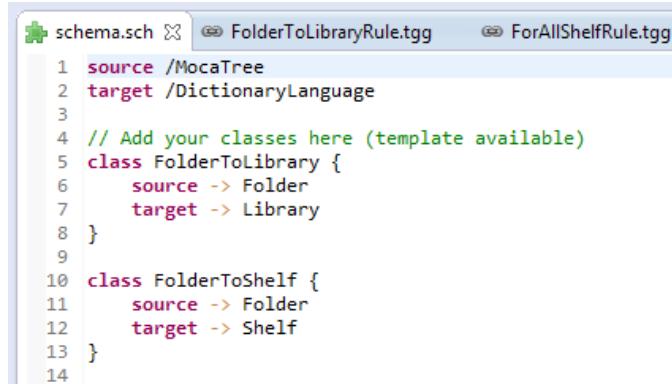


```

1 rule ForAllShelfRule {
2 source {
3 inputFolder : Folder {
4 ++ -subFolder-> shelfFolder
5 }
6 ++ shelfFolder : Folder
7 }
8
9 correspondence {
10 inputFolder <- folderToLibrary : FolderToLibrary -> library
11 ++ shelfFolder <- folderToShelf : FolderToShelf -> shelf
12 }
13
14 target {
15 library : Library {
16 ++ -shelf-> shelf
17 }
18
19 ++ shelf : Shelf
20 }
21
22 }
23
24 constraints {[[
25 eq(shelfFolder.name, shelf.description)
26]]}
27 }
```

Figure 3.17: ForAllShelfRule

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



```

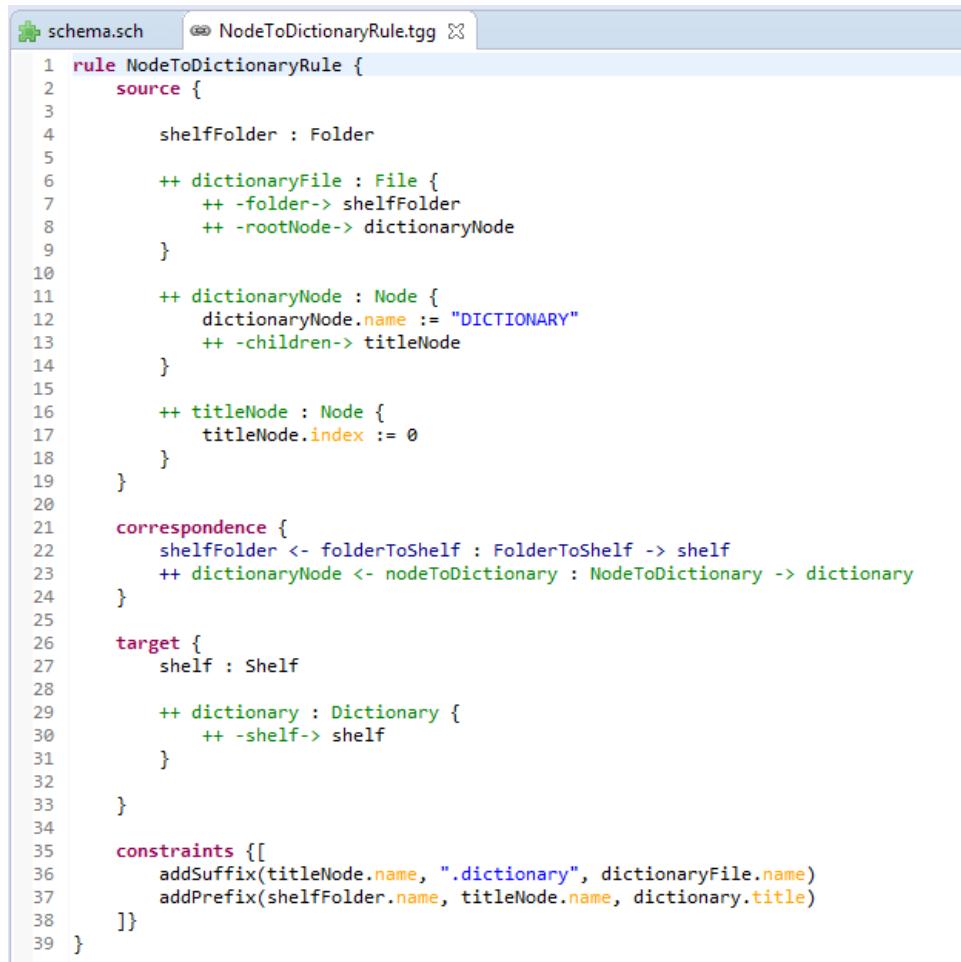
1 source /MocaTree
2 target /DictionaryLanguage
3
4 // Add your classes here (template available)
5 class FolderToLibrary {
6 source -> Folder
7 target -> Library
8 }
9
10 class FolderToShelf {
11 source -> Folder
12 target -> Shelf
13 }
14
```

Figure 3.18: Updated TGG schema

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

### NodeToDictionaryRule

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



```

1 rule NodeToDictionaryRule {
2 source {
3
4 shelfFolder : Folder
5
6 ++ dictionaryFile : File {
7 ++ -folder-> shelfFolder
8 ++ -rootNode-> dictionaryNode
9 }
10
11 ++ dictionaryNode : Node {
12 dictionaryNode.name := "DICTIONARY"
13 ++ -children-> titleNode
14 }
15
16 ++ titleNode : Node {
17 titleNode.index := 0
18 }
19 }
20
21 correspondence {
22 shelfFolder <- folderToShelf : FolderToShelf -> shelf
23 ++ dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
24 }
25
26 target {
27 shelf : Shelf
28
29 ++ dictionary : Dictionary {
30 ++ -shelf-> shelf
31 }
32 }
33
34 constraints {
35 addSuffix(titleNode.name, ".dictionary", dictionaryFile.name)
36 addPrefix(shelfFolder.name, titleNode.name, dictionary.title)
37 }
38 }
39 }
```

Figure 3.19: `NodeToDictionaryRule` handling only `titleNodes`

- ▶ As you can see, this rule demands that a `shelfFolder` and `shelf` already exist before executing, implying that this rule can only be called after executing `ForAllShelfRule`. An attribute constraint is used with `titleNode` to ensure that the correct child `Node` is matched from `dictionaryNode`, and not accidentally to an author or entry node, which will have different indices.
- ▶ This rule also imposes two constraints for attribute manipulation. We need to add the name of the shelf as a prefix to the title node's name to get the dictionary's title (i.e., "english" + "numbers1-10"). Similarly, the second constraint appends `.dictionary` to `titleNode.name` to get the file name of the dictionary.

### **ForAllEntryRule**

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

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

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

```

1 rule ForAllEntryRule {
2 source {
3 dictionaryNode : Node {
4 ++ -children-> entryNode
5 }
6
7 ++ entryNode : Node {
8 entryNode.name := "ENTRY"
9 ++ -children-> contentNode
10 ++ -children-> indexNode
11 }
12
13 ++ contentNode : Node {
14 contentNode.index := 0
15 }
16
17 ++ indexNode : Node {
18 indexNode.index := 1
19 }
20 }
21
22 correspondence {
23 dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
24 ++ entryNode <- nodeToEntry : NodeToEntry -> entry
25 }
26
27 target {
28 dictionary : Dictionary {
29 ++ -entry-> entry
30 }
31
32 ++ entry : Entry
33 }
34
35
36 constraints {[[
37 eq(contentNode.name, entry.content)
38 eq(indexNode.name, entry.level)
39 setDefaultNumber(entryNode.index, 2)
40]]}
41 }

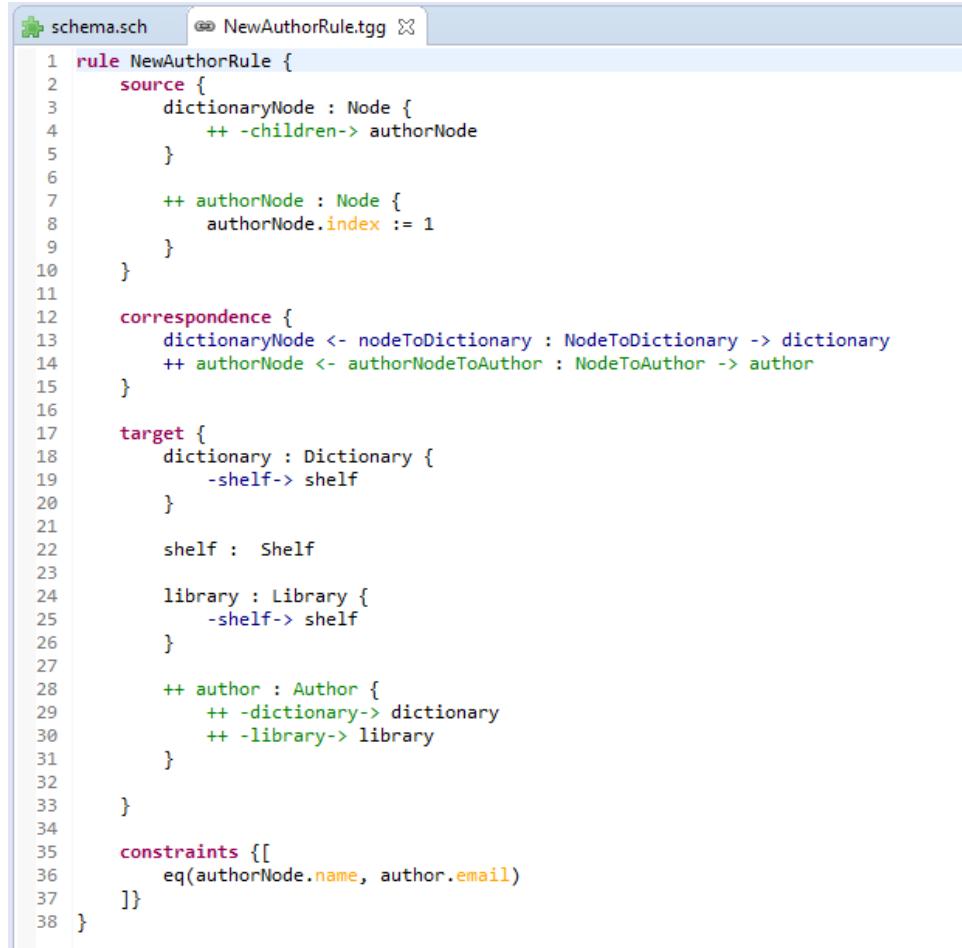
```

Figure 3.20: ForAllEntryRule

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

## NewAuthorRule

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



```

1 rule NewAuthorRule {
2 source {
3 dictionaryNode : Node {
4 ++ -children-> authorNode
5 }
6
7 ++ authorNode : Node {
8 authorNode.index := 1
9 }
10
11
12 correspondence {
13 dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
14 ++ authorNode <- authorNodeToAuthor : NodeToAuthor -> author
15 }
16
17 target {
18 dictionary : Dictionary {
19 -shelf-> shelf
20 }
21
22 shelf : Shelf
23
24 library : Library {
25 -shelf-> shelf
26 }
27
28 ++ author : Author {
29 ++ -dictionary-> dictionary
30 ++ -library-> library
31 }
32 }
33
34
35 constraints {
36 eq(authorNode.name, author.email)
37 }
38 }

```

Figure 3.21: Creating new authors in `NewAuthorRule`

### ExistingAuthorRule

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

```

1 rule ExistingAuthorRule {
2 source {
3 dictionaryNode : Node {
4 ++ -children-> authorNode
5 }
6
7 ++ authorNode : Node {
8 authorNode.index := 1
9 }
10 }
11
12 correspondence {
13 dictionaryNode <- nodeToDictionary : NodeToDictionary -> dictionary
14 ++ authorNode <- authorNodeToAuthor : NodeToAuthor -> author
15 }
16
17 target {
18 dictionary : Dictionary {
19 -shelf-> shelf
20 }
21
22 shelf : Shelf
23
24 library : Library {
25 -shelf-> shelf
26 }
27
28 author : Author {
29 ++ -dictionary-> dictionary
30 -library-> library
31 }
32 }
33
34
35 constraints {[[
36 eq(authorNode.name, author.email)
37 }}}
38 }

```

Figure 3.22: Checking for existing authors in `ExistingAuthorRule`

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

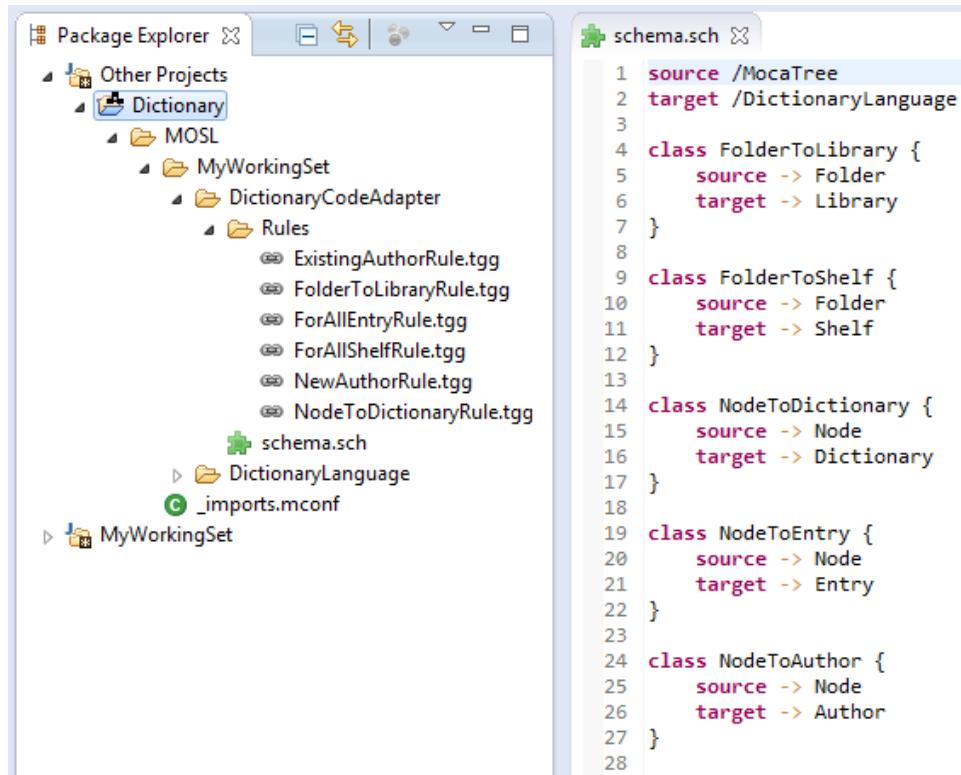


Figure 3.23: Your final `rules` project structure and `schema`

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

### 3.3 Additional author handling

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

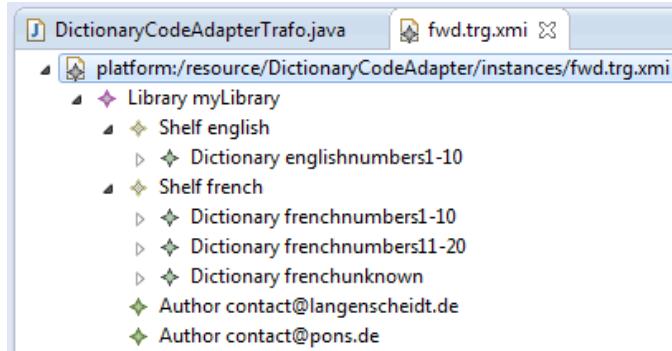


Figure 3.24: Dictionary result of the forward transformation

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

There are two ways to do this:

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

---

<sup>14</sup>If you haven't already, read Part IV, Section 6 for details on to run a transformation

## Option 1: Run-time decision

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

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

```

18 import org.moflon.tgg.algorithm.configuration.Configurator;
19 import org.moflon.tgg.algorithm.configuration.RuleResult;
20
21 public class DictionaryCodeAdapterTrafo extends SynchronizationHelper {
22 public DictionaryCodeAdapterTrafo() throws IOException {}
23 public static void main(String[] args) throws IOException {}
24
25 public void performForward(String source) {
26 try {
27 loadSrc(source);
28 } catch (IllegalArgumentException iae) {
29 System.err.println("Unable to load " + source + ", " + iae.getMessage());
30 return;
31 }
32
33 setConfigurator(new Configurator() {
34 @Override
35 public RuleResult chooseOne(Collection<RuleResult> alternatives) {
36 Optional<RuleResult> preferred = alternatives.stream()
37 .filter(rr -> rr.isRule("NewAuthorRule"))
38 .findAny();
39 return preferred.orElse(Configurator.super.chooseOne(alternatives));
40 }
41 });
42
43 integrateForward();
44
45 saveTrg("instances/fwd.trg.xmi");
46 saveCorr("instances/fwd.corr.xmi");
47 saveSynchronizationProtocol("instances/fwd.protocol.xmi");
48
49 System.out.println("Completed forward transformation!");
50 }
51
52 public void performBackward(String target) {}
53 }
```

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

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

## Option 2: Design-time decision

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

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

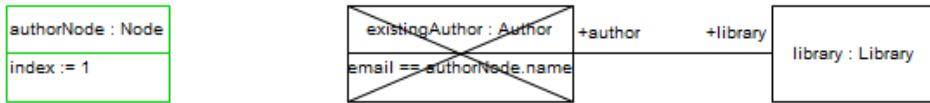


Figure 3.26: Adjust `NewAuthorRule` by adding a NAC

```

17 target {
18 dictionary : Dictionary {
19 -shelf-> shelf
20 }
21
22 shelf : Shelf
23
24 library : Library {
25 -shelf-> shelf
26 }
27
28 | ! existingAuthor : Author {
29 author.email == authorNode.name
30 -library-> library
31 }
32
33 ++ author : Author {
34 ++ -dictionary-> dictionary
35 ++ -library-> library
36 }
37
38 }

```

Figure 3.27: Add a NAC to `NewAuthorRule`

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

## 4 Tree-to-text transformation

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

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

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

- ▶ Expand “src/org.moflon.moca.dictionary.unparser”, open `DictionaryTreeGrammar.g`, and edit the contents as depicted in Fig. 4.1.
- ▶ Next, open `DictionaryUnparserAdapter.java` (Fig 4.2). You'll notice that this file contains a (commented out) `StringTemplateGroup` method for retrieving a group of templates and needs to be implemented. The comments explain how to use either a folder containing different template files, or a single file containing all templates. The latter is better for numerous small templates, while the former makes sense when the templates contain a lot of static text.
- ▶ For this small example, a single file with all templates is ideal. Uncomment line 44 (the option for a group file) and remove the line throwing an `UnsupportedOperationException`.

```

1 tree grammar DictionaryTreeGrammar;
2
3 options {
4 ASTLabelType = CommonTree;
5 output = template;
6 }
7
8 // Tokens used internally by Moca
9 // ID: ('a'...'z' | 'A'...'Z')+;
10 // STRING: (ID | ('0'...'9'))+;
11 // ATTRIBUTE: Used as an imaginary...
12
13 tokens {
14 ID;
15 STRING;
16 ATTRIBUTE;
17 }
18
19@members {□
20}
21
22@header {□
23}
24
25 // tree grammar rules:
26 main: ^('DICTIONARY' name=STRING author=STRING? entries+=entry+)
27 -> dictionary(name={$name}, author={$author}, entries={$entries});
28
29 entry: ^('ENTRY' entryLabel=STRING level=STRING)
30 -> entry(entry={$entryLabel}, level={$level});
31
32
33
34
35
36
37
38
39
40
41

```

Figure 4.1: Tree grammar for the unparser

```

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

```

Figure 4.2: Two options of how to store templates

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



The screenshot shows a code editor window titled "Dictionary.stg". The code is written in a template language, likely XText or similar, and defines a "group" named "dictionary". This group contains a "title" and a block of entries. The entries are defined by a "group" named "entry" which takes two parameters: "entry" and "level". The code uses double-angle brackets (<>) for template parameters and single-angle brackets (<>) for XML elements like <name> and <author>. The code is numbered from 1 to 13.

```
1 group dictionary;
2
3 dictionary(name, author, entries) ::= <<
4 title: "<name>"
5 <if(author)>email: "<author>"<endif>
6 {
7 <entries; separator="\n">
8 }
9 >>
10
11 entry(entry, level) ::= <<
12 "<entry>", <level>
13 >>
```

Figure 4.3: The dictionary template

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

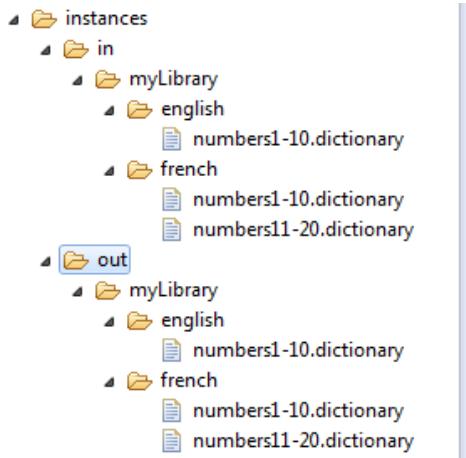


Figure 4.4: The final input and output filesystems

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

## 5 Conclusion and next steps

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

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

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

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

Cheers!

# Glossary

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

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

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

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

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

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

# An Introduction to Metamodelling and Graph Transformations

---

*with eMoflon*



---

## Part VI: Miscellaneous

For eMoflon Version 2.0.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Lars Kliegel, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, Sascha Edwin Zander, Jerome Reinländer, Martin Wieber, and contributors. All rights reserved.

This document is free; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version. Please visit <http://www.gnu.org/copyleft/fdl.html> to find the full text of the license.

For further information contact us at [contact@emoflon.org](mailto:contact@emoflon.org).

*The eMoflon team*  
Darmstadt, Germany (August 2015)

# Contents

|    |                                                  |    |
|----|--------------------------------------------------|----|
| 1  | Grokking Enterprise Architect . . . . .          | 2  |
| 2  | Using EA with subversion . . . . .               | 13 |
| 3  | Using existing EMF projects in eMoflon . . . . . | 20 |
| 4  | MOSL syntax . . . . .                            | 27 |
| 5  | eMoflon in a Jar . . . . .                       | 28 |
| 6  | Useful shortcuts . . . . .                       | 31 |
| 7  | Legacy support for CodeGen2 . . . . .            | 33 |
| 8  | Creating and Using Enumerations . . . . .        | 34 |
| 9  | Glossary . . . . .                               | 38 |
| 10 | GNU General public license . . . . .             | 42 |

## Part VI:

# And all that (eMoflon) jazz

URL of this document: <http://tiny.cc/emoflon-rel-handbook/part6.pdf>

Welcome to the miscellaneous part of our eMoflon handbook. You can consider this Part to be the ‘bonus’ or appendix area of the entire handbook series. Here we have collected and documented a series of advanced topics related to our tool. These include some tips and tricks you may find helpful while using the tool with Enterprise Architect (EA), a syntax reference table of our Moflon Specification Language (MOSL), and information about the protocol file generated with every Triple Graph Grammar (TGG) transformation which we were never able to explain in Parts IV or V. This entire part is kept rather compact, intended to be used mainly as a reference and consulted on demand.

Please note that if you’re looking for instructions on how to properly export and import separate metamodels into the same project for work with TGG transformations, please refer to Part IV, Section 2.2 (visual/EAP files) and 2.3 (textual/Eclipse projects), where we included detailed steps in the context of an example.

If you feel anything is missing from this part, or if you have any other comments or suggestions about the handbook series and our tool, feel free to contact us anytime at [contact@emoflon.org](mailto:contact@emoflon.org).

# 1 Grokking Enterprise Architect

Grok: "...to understand so thoroughly that the observer becomes a part of the observed."

- Robert A. Heinlein, *Stranger in a Strange Land*

This section is a collection of a few of what we feel are the most important tips and tricks for working productively with Enterprise Architect (EA). We truly believe that spending the time to learn and practice these is necessary for a pleasant modelling experience.

## 1.1 Positioning elements

Layout is always an important factor when using a visual language: A well laid-out diagram is easiest to understand and, by centralizing important elements or clustering related elements, you can actually impart additional information.

- ▶ To select a group of elements, either drag a selection box around the items or hold **Ctrl** and select each element one-by-one.
- ▶ In the top right corner of the last selected element, a small colon-styled symbol will appear (Fig. 1.1). Click on this for a context list of different options you can simultaneously apply to all active elements. The same list appears on the toolbar above the diagram.
- ▶ Experiment to find out what effect each option has. The last symbol in the list opens a further drop-down menu with standard layout algorithms to organize your diagram automatically.
- ▶ Right-clicking any of the selected elements opens a different menu with a further set of layout options and their descriptions (Fig. 1.2). **Align Centers** or **Same Height** and **Width** can be especially useful.

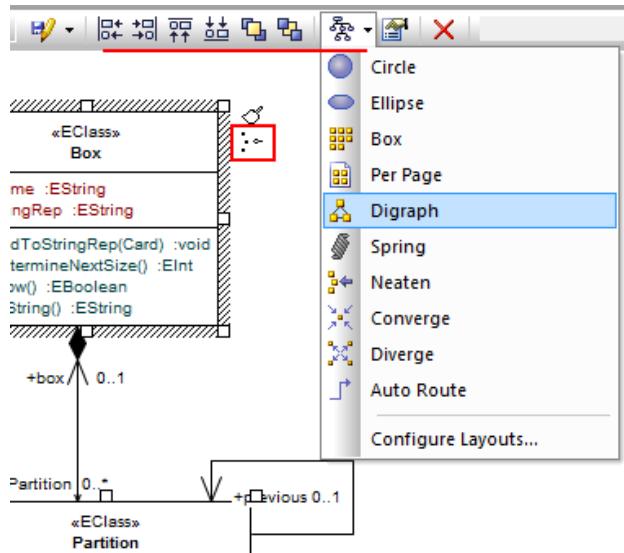


Figure 1.1: Setting the layout of multiple elements

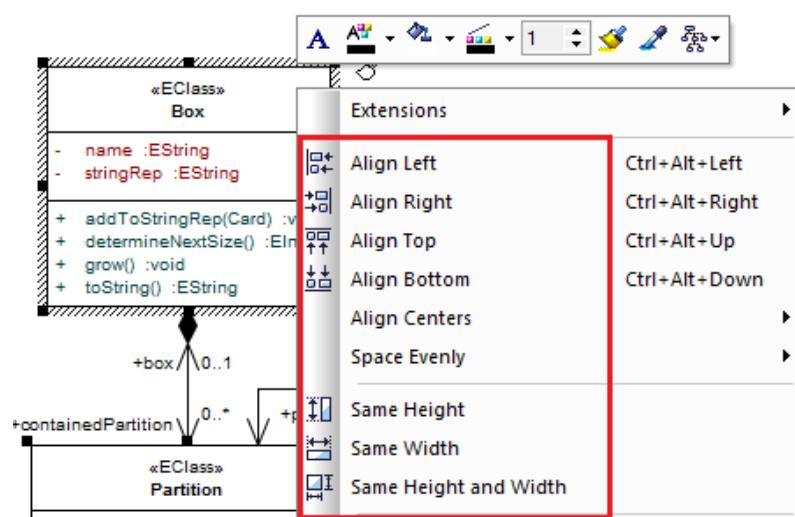


Figure 1.2: Further layout options

## 1.2 Bending lines to your will

Another important part of a good layout is getting lines to be just the way you want them to be. In EA you can add and remove bending points which can be used to control the appearance of a line.

- Hold down **Ctrl** and click on a line to create a bending point (Fig. 1.3). You can now pull the bending point and shift the line as you wish.

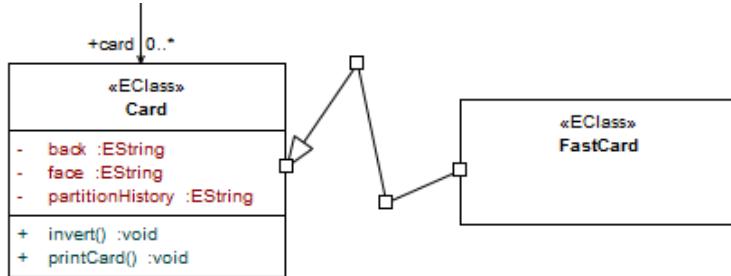


Figure 1.3: Adding bending points to a line

- You can create as many bending points as you wish, and you can *remove* them by holding down **Ctrl** and clicking once on the unwanted point.

## 1.3 Deleting vs. removing elements from diagrams

A central feature that new users should understand as soon as possible is the way EA handles diagrams. *A diagram is simply treated as a view of the complete model.* The complete model can always be browsed in its entirety via a tree view in the package browser. This space contains all elements that will be exported. The driving reason behind this setup is that diagrams typically do not contain all elements and one usually uses multiple (possibly redundant) diagrams to show the different parts of the model. Thinking in this frame is crucial and provides a pragmatic solution to the problem of having huge, unmaintainable diagrams.

A tricky consequence one must get used to is that removing an element from a diagram does *not* delete it from the model. We have added some support with the validation in the eMoflon add-in control panel, which can prompt a warning when an element cannot be found in any diagram,<sup>1</sup> but there's currently no way to recover a deleted element.

---

<sup>1</sup>Review Part II, Section 2.8 for an example

A common mistake new users make is to remove an element by pressing **Del**, and expecting the element to be deleted from the model. As you can probably guess, this is not the case as evidenced in the package browser (Fig. 1.4).

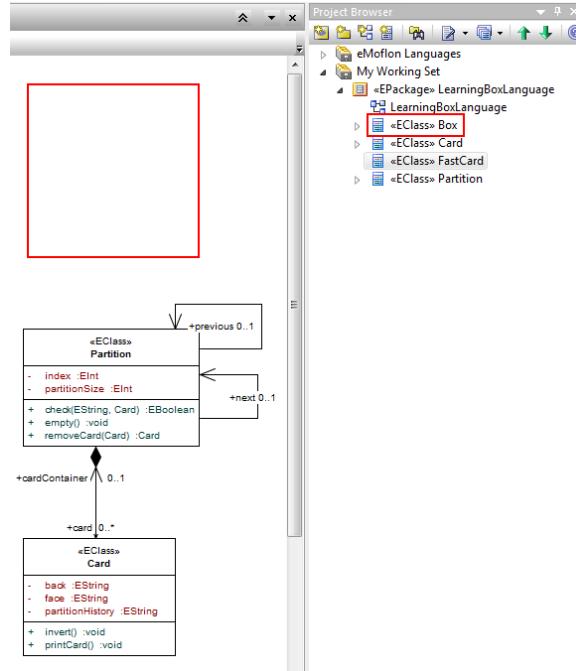


Figure 1.4: Removing an element from a diagram via pressing **Del** does not delete it from the model and it is still present in the package browser

- ▶ To fully delete an element from a model (not just a diagram), select it in the diagram and press **Ctrl + Del**. Confirm the action in the warning dialogue (Fig. 1.5), and the element should no longer be in the project browser.
- ▶ Alternatively, elements can be deleted directly from project browser by right-clicking the item and navigating to the large red ‘x’ at the bottom of the context menu

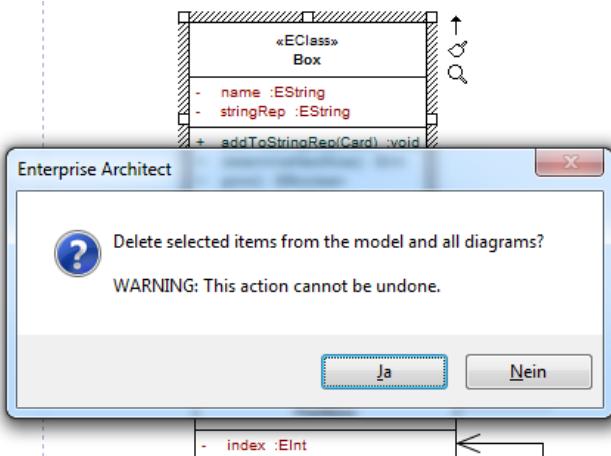


Figure 1.5: Deleting an element from a diagram and the model

## 1.4 Excluding certain projects from the export

You may find it sometimes necessary to exclude certain projects from your diagram export (such as the `MocaTree` model used in Part V). Some reasons for this could be (i) because the project is still a work in progress and simply not ready to be exported, (ii) because the complete project is present in the Eclipse workspace but has not been modelled completely in EA, and you wish to do this gradually on-demand, (iii) because the project is not meant to be present in your Eclipse workspace as generated code and is instead provided via a plugin (this is usually the case for standard metamodels like Ecore, UML etc.), or (iv) because the project is rather large and stable and you do not want to wait for EA to process a known, unchanging model. Whatever the reason, you can prevent unnecessary exports by setting a certain *tagged value* of the project.

- ▶ Open your project in EA, and navigate to “View/Tagged Values” from the menu bar (Fig. 1.6).
- ▶ The tagged value, `Moflon::Export`, should already be present and be set to a default `true` value (Fig. 1.7). If you want the project to be ignored by the eMoflon’s validation and/or export functions, change the value to `false` (and conversely back to `true` to export it again).

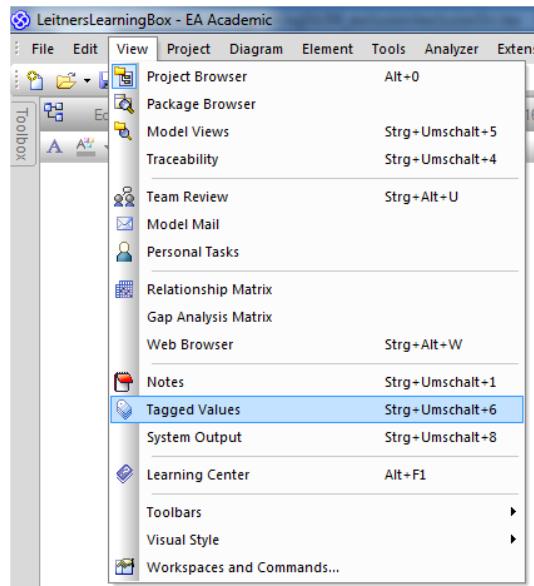


Figure 1.6: Opening the tagged values view

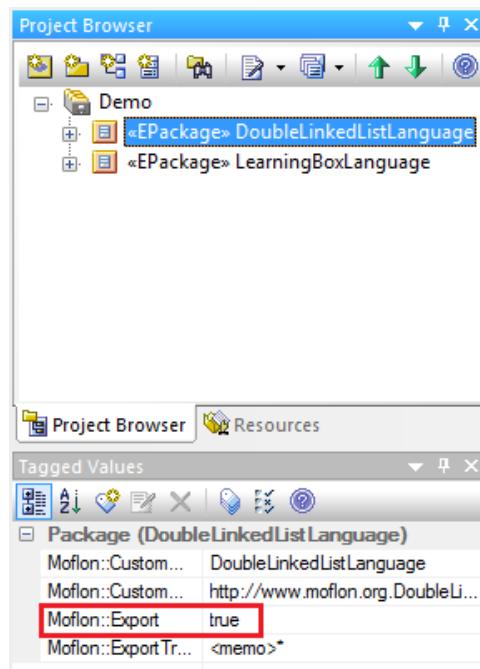


Figure 1.7: The Moflon::Export setting determines ignored projects

## 1.5 Getting verbose!

Although we use colours in SDMs to indicate when an element is to be matched (black), created (green), or destroyed (red), it sometimes makes sense to indicate these binding operators via explicit stereotypes (i.e., for black-and-white printouts of a model).

- ▶ Open the relevant diagram in the EA editor window and, depending on what type it is, press the **Verbose** button in either the **eMoflon SDM Functions** or **eMoflon TGG Functions** panel (Fig. 1.8).

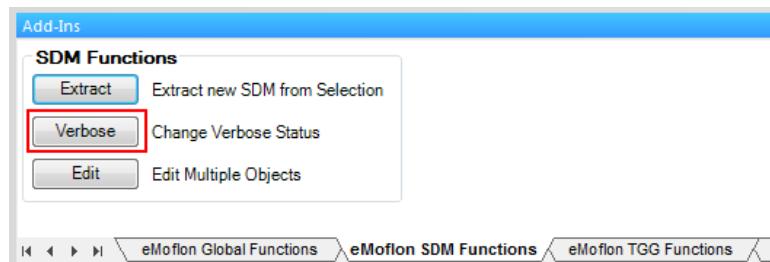


Figure 1.8: Add extra markup to colored links and objects in the current diagram

- ▶ This will add small **++** or **--** symbols next to deleted and created elements in the current diagram (Fig. 1.9). Press the button again to deactivate these indicators.

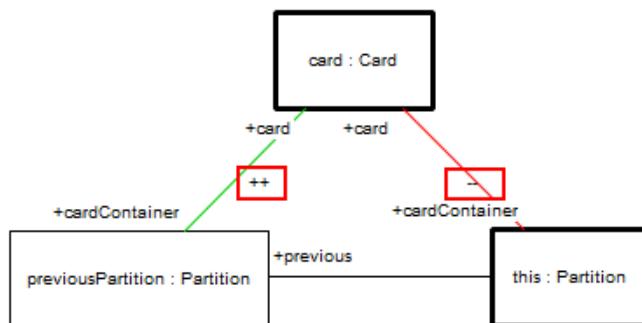


Figure 1.9: Diagram in verbose mode

## 1.6 Duplicating elements via drag-and-drop

Sometimes you'll have an element (or many) that are nearly identical, and life would be *so* much easier if you could copy and paste an existing one already. Suppose you want a copy of a `this` element, so you press `Ctrl + C`, followed by `Ctrl + V`. An error dialogue preventing the action will immediately raise, stating that the "... diagram already contains an instance of the element you are trying to paste." EA can only support unique objects, so you'll need to use the following process.

- In either a diagram or in the project browser, hold `Ctrl`, then drag the element you wish to duplicate. A confirmation-style dialogue will appear (Fig. 1.10), and a properties window will follow. You must assign a unique name to the new element, or else you'll receive an error when you try to export the project later.

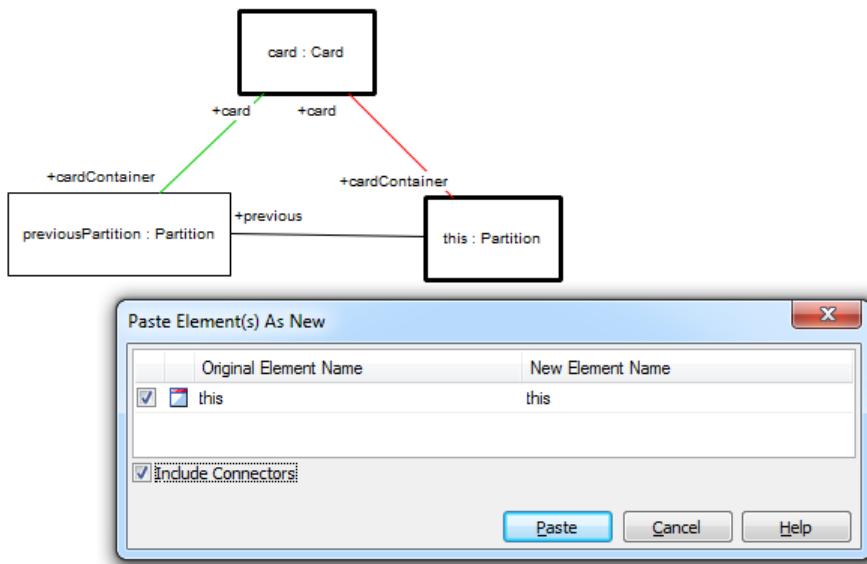


Figure 1.10: Copying elements

## 1.7 Seek, and ye shall find . . .

EA has a model search function that can be quite handy for large models with thousands of elements and a brain that can't *quite* remember where something is.

- Select **Model Search Window** in the toolbar and enter the name of an element you wish to find (Fig. 1.11).<sup>2</sup>

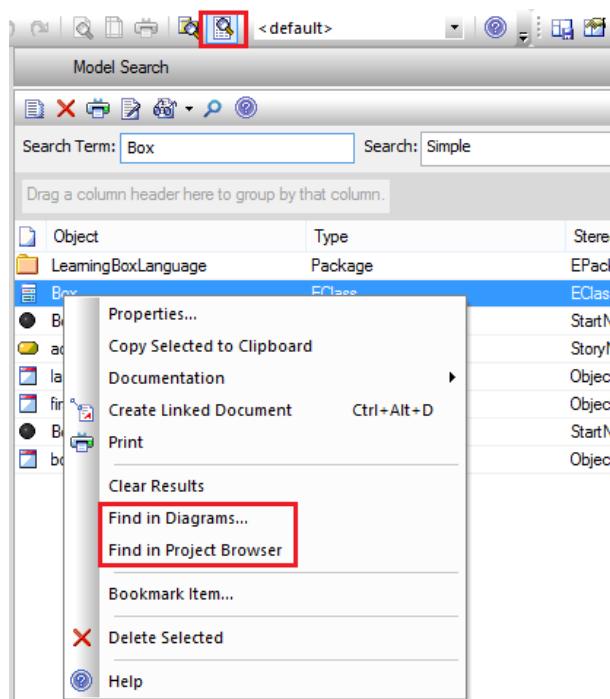


Figure 1.11: Model Search Window

- All elements that meet the search criteria are listed and you can right-click on each of the items and select one of the options above to locate the element.
- In a similar way, you can locate the corresponding class of an object by right clicking and selecting "Find/Locate Classifier in Project Browser."

<sup>2</sup>You can also access this window by pressing **Ctrl+Alt+A**

## 1.8 Advanced search

EA offers an even more advanced search capability using SQL.<sup>3</sup>

- ▶ To use this, first open the model search window via either the menu bar or by pressing **Ctrl + Alt + A**.
- ▶ Click the “Builder” button, and switch to the **SQL** tab (Fig. 1.12).

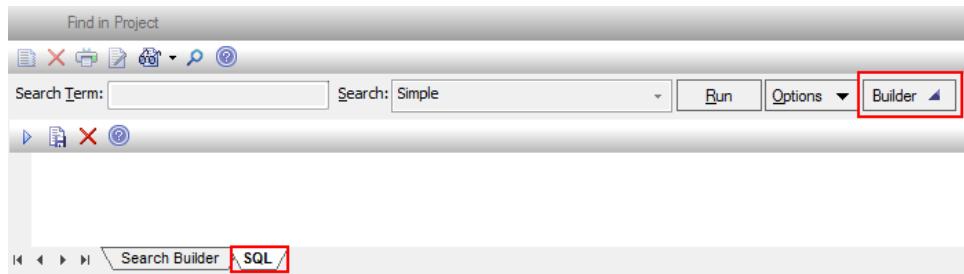


Figure 1.12: Advanced project search window

Here you can formulate any query on the underlying database. The SQL-editor helps you with syntax-highlighting and auto-completion. Here are some basic examples to get you started:

- ▶ To find all eClasses

```
SELECT * FROM t_object
WHERE Object_Type='Class' AND Stereotype='eClass';
```

- ▶ To find all associations

```
SELECT * FROM t_connector
WHERE Connector_Type='Association';
```

- ▶ To find all inheritance relations

```
SELECT * FROM t_connector
WHERE Connector_Type='Generalization';
```

- ▶ To find all connectors attaching a note to an element

```
SELECT * FROM t_connector
WHERE Connector_Type='NoteLink';
```

---

<sup>3</sup>For some detailed insights to the general database schema used by EA cf. [http://www.sparxsystems.com.au/downloads/corp/scripts/SQLServer\\_EASchema.sql](http://www.sparxsystems.com.au/downloads/corp/scripts/SQLServer_EASchema.sql)

- ▶ To find all control flow edges (used in SDMs)

```
SELECT * FROM t_connector
WHERE Connector_Type='ControlFlow';
```

- ▶ To find all associations connected to a class named “EClass”

```
SELECT t_object.Name, t_connector.* FROM t_connector,t_object
WHERE t_connector.Connector_Type='Association'
 AND (t_connector.Start_Object_ID=t_object.Object_ID
 OR t_connector.End_Object_ID=t_object.Object_ID)
 AND t_object.Name='EClass';
```

- ▶ To determine all subtypes of “EClassifier”

```
SELECT a.Name FROM t_connector,t_object a,t_object b
WHERE t_connector.Connector_Type='Generalization'
 AND t_connector.Start_Object_ID=a.Object_ID
 AND t_connector.End_Object_ID=b.Object_ID
 AND b.Name = 'EClassifier';
```

- ▶ To determine all supertypes of “EClassifier” (cf. above)

```
...
 AND t_connector.Start_Object_ID=b.Object_ID
 AND t_connector.End_Object_ID=a.Object_ID
...
```

To run the search, either hit the Run SQL button in the upper left corner of the editor toolbar (it shows a triangular shaped “play” icon), or press F5 on your keyboard.

## 2 Using EA with subversion

The following steps are required to setup EA for use with Subversion. This is highly recommended when working in a group and sharing a single EA Project (EAP) file, which is otherwise a huge binary blob. This section assumes two things – you wish to use (i) Subversion and (ii) Windows. For other SCM and operating systems please consult the official documentation from EA.

### 2.1 Initial preparation and setup

Download and install **Slik SVN** (mandatory):

- ▶ x32: <http://www.sliksvn.com/pub/Slik-Subversion-1.8.9-win32.msi>
- ▶ x64: <http://www.sliksvn.com/pub/Slik-Subversion-1.8.9-x64.msi>
- ▶ You may also check for a more recent version at <https://www.sliksvn.com/en/download/>.

For public/private key authentication, you need **Tortoise SVN**:

- ▶ x32: <http://sourceforge.net/projects/tortoisesvn/files/1.7.9/Application/TortoiseSVN-1.7.9.23248-win32-svn-1.7.6.msi/download>  
<http://downloads.sourceforge.net/project/tortoisesvn/1.7.9/Application/TortoiseSVN-1.7.9.23248-win32-svn-1.7.6.msi/download>
- ▶ x64: <http://sourceforge.net/projects/tortoisesvn/files/1.7.9/Application/TortoiseSVN-1.7.9.23248-x64-svn-1.7.6.msi/download>  
<http://downloads.sourceforge.net/project/tortoisesvn/1.7.9/Application/TortoiseSVN-1.7.9.23248-x64-svn-1.7.6.msi/download>
- ▶ You may check for a more recent version at <http://tortoisesvn.net/downloads.html>.

If you do not want to have your private key password in plain text in an SVN configuration file, then also download **Pageant**:

- ▶ <http://the.earth.li/~sgtatham/putty/latest/x86/pageant.exe>

With all of the tools installed, we now have to setup the SSH tunnel:

- ▶ Locate the file %APPDATA%\Subversion\config and open it with your favourite editor. Locate the [tunnels] section.

- ▶ If you do not want to install Pageant and do not mind entering your password in plain text enter the following command:  

```
ssh = "<path/to/Tortoise/SVN>/bin/TortoisePlink.exe" -l
<username> -pw <password for your private key> -i "<path to
your private key>"
```
- ▶ If you wish to use Pageant then the command can be simplified to:  

```
ssh = "<path/to/Tortoise/SVN>/bin/TortoisePlink.exe" -l
<username> as you can add your private key to Pageant.
```

*Note:* The connection to Pageant may sometimes fail, so additional steps may be necessary.
- ▶ It is **very important** that you use **forward slashes** for any filenames, otherwise SVN will not find the files.
- ▶ If you just use direct passwords for authentication then you can leave out the **-i** option in both cases.

## 2.2 How to setup a version-controlled EAP file

The following assumes an EAP file has already been placed under version control as instructed in the previous section, and that you wish to checkout this file and work with it. If our instructions do not work, the EAP file may have been placed under version control in a different manner. If this is the case, please contact whoever checked in the file, and setup the file for working with EA and SVN for further instructions.

- ▶ Check-out the project with the EAP file from the server using Tortoise-SVN or Eclipse/Subclipse (or any SVN client of your choice). You should now have a *.svn* folder in the directory where you saved the revision.
- ▶ Open the EAP file. If the EAP file is already under version control *and* has been set-up correctly, a dialogue similar to Fig. 2.1 should immediately pop-up.
- ▶ Click “Yes” to open the “Version Control Settings” dialogue (Fig. 2.2).
- ▶ To work with the EAP file, you now have to *redefine* the SVN variable for the file in your EA workspace. To accomplish this, choose the local path to the folder which contains the EAP file in the “Working Copy Path” text-box, and correct the value in “Subversion Exe Path” if necessary (to fit your Slik installation location).

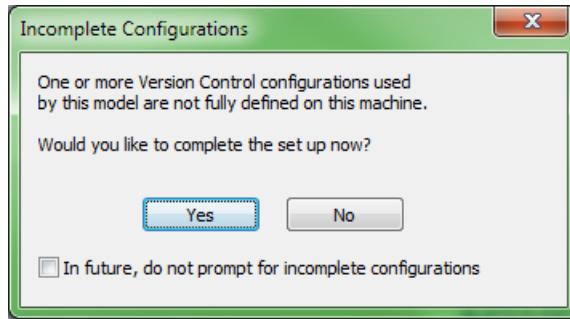


Figure 2.1: Configure an EAP file which is already under version control

### 2.3 Working with a version-controlled EAP file

- ▶ A **Check Out** retrieves the lock for a certain package and gives you exclusive access, i.e., no one else can change the package. Very important: if subpackages are also under version control, they are not affected by checking out the “super”-package and remain locked. A **Check Out** also updates the package to the latest version.
- ▶ A **Check In** commits your work to the server and gives up the lock on the package so others can work on it. If you do not want to commit your changes, you can just use **Undo Check Out...** to revert all local changes.
- ▶ The corresponding **..Branch** options perform the actions for the current package and all subpackages. Please note, this has nothing to do with “branching” in normal SVN lingo.
- ▶ **Get Latest/Get All Latest** retrieves the latest version of the selected package / all packages. This is basically an update but does not retrieve the lock for any package.
- ▶ Conversely, **Put Latest** saves all your changes without giving up any locks.
- ▶ **Compare with controlled version** can be used to review incoming changes. Green elements will be added, red will be deleted.
- ▶ **File History** gives you a summary of all commits made while you were lying on the beach. For a useful file history, always use meaningful commit statements when checking in! A date stamp is created automatically.

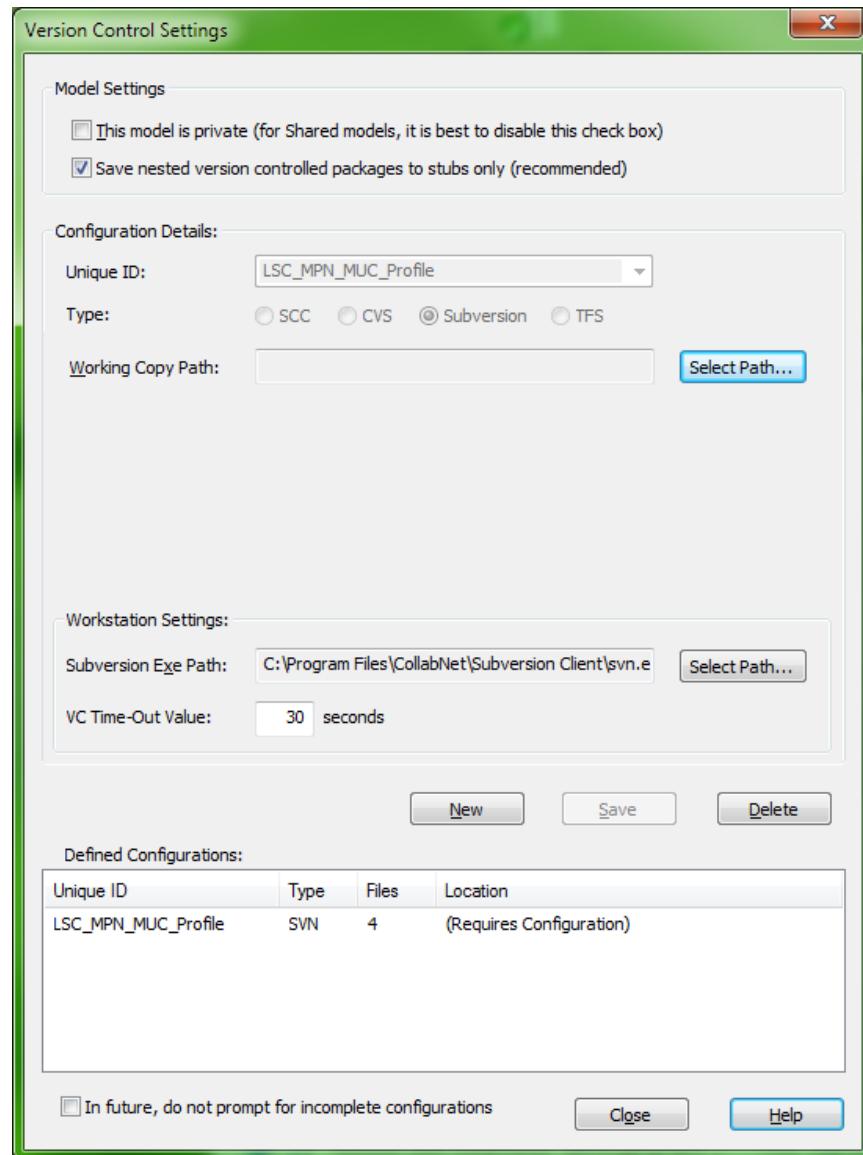


Figure 2.2: Update settings as required

## 2.4 Placing an EAP file under version control

If you already have an EAP file and would like to place it under version control, you first have to check it in as usual on the server using your favourite SVN client. Once the project is checked in, the required .svn folder should be in the folder containing the EAP file. The next step is to register an SVN-variable in EA:

- ▶ Open the EAP file, right click on a root folder and select “Package Control” and then “Version Control Settings...” (Fig. 2.3).

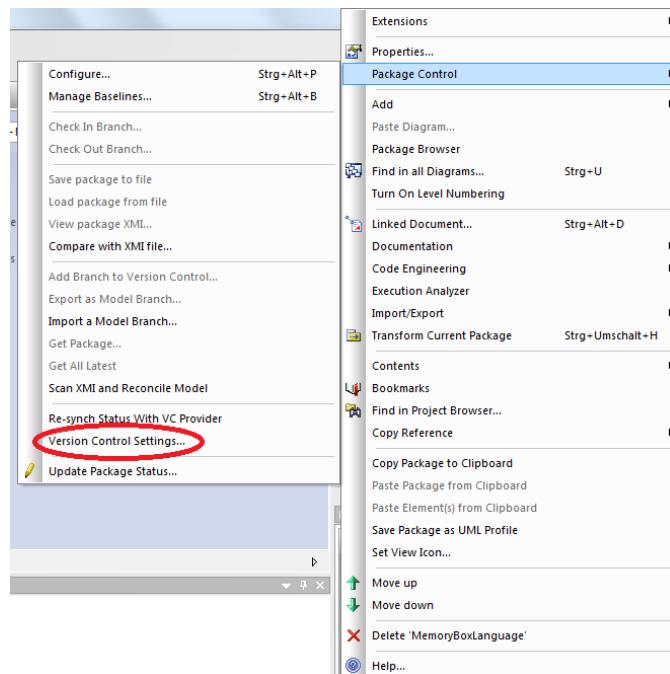


Figure 2.3: Select version control settings

- ▶ In the dialogue, choose a unique ID of your choice (we suggest you use the name of the EAP file) for the settings and activate the “Subversion” radio button below.
- ▶ Choose the local path to the folder which contains the EAP file in the “Working Copy Path” text-box.
- ▶ The field “Workstation Settings” must point to where you installed Sliksvn, i.e., <path to SlikSVN>\bin\svn.exe"). Press “Save” and close the dialogue (Fig. 2.4). If the dialogue closes without an error message, then you can be sure to have configured everything correctly.

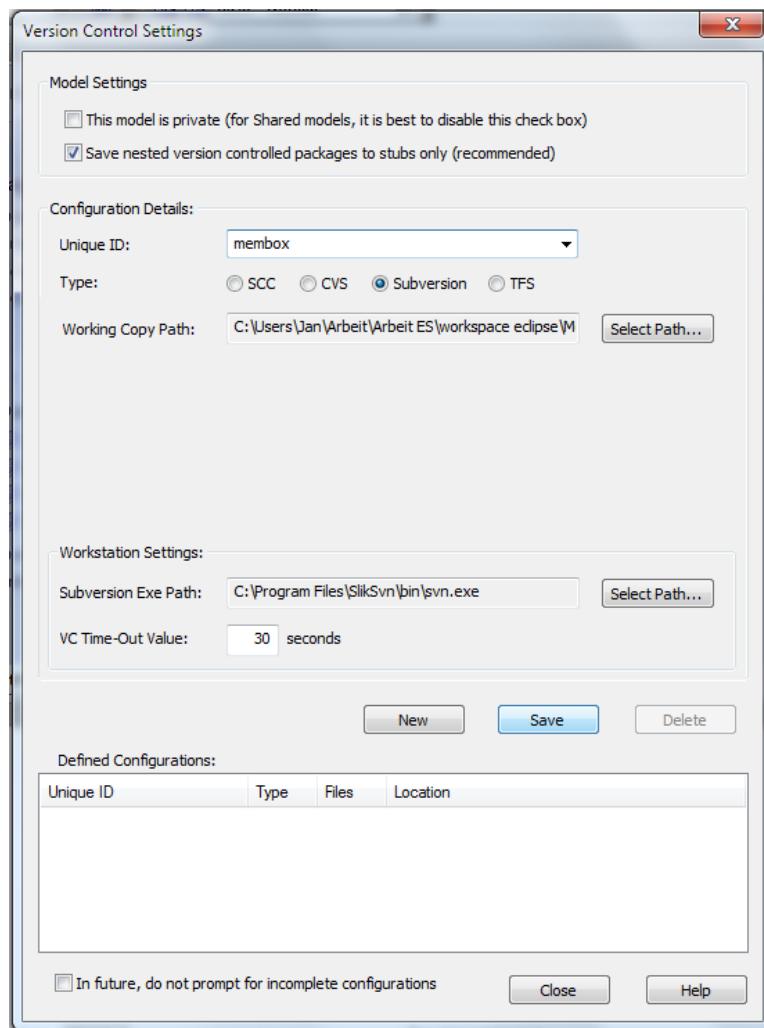


Figure 2.4: Register an SVN variable in EA

- ▶ In the EAP file, choose “Package Control\Configure...” for *each package* you wish to place under version control.
- ▶ In the ensuing dialogue, activate “Control Package” and select your previously defined SVN variable from the drop-down menu. Enter the path where the XML file for the project should be placed. Although this is not enforced in any way, we recommend you create a folder structure that mirrors the package structure in EA (Fig. 2.5). This process has to be repeated *for all sub-packages* as soon as their super-package has been placed under version control.

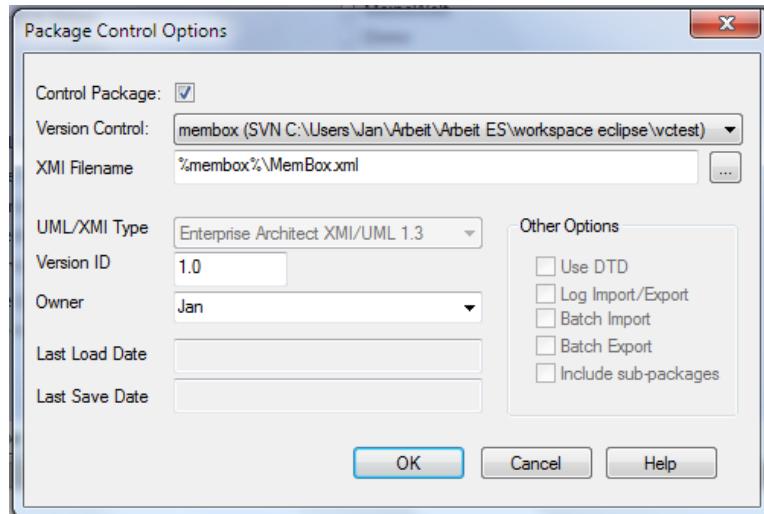


Figure 2.5: Placing a package under version control

- ▶ As a final step, check-in the current state of the EAP file directly with your SVN client. As from this point, the EAP file should not be checked-in anymore, and all versioning actions should be performed via EA (and not directly with your SVN client).

### 3 Using existing EMF projects in eMoflon

This chapter contains stepwise instructions on how to use existing EMF/Ecore projects with an eMoflon project specified using the visual syntax via EA. We will present an example of an existing metamodel which must be integrated with eMoflon before, for example, its transformation using SDMs can be specified. The basic workflow for using an existing EMF project in eMoflon is described in the following and may of course be similarly applied to a metamodel specified in the textual syntax via MOSL.

We will begin by implementing a small subset of the `Ecore -> GenModel` transformation, where `GenModel` is part of the EMF/Ecore standard. The *GenModel* for a given Ecore model can be viewed as a *wrapper* that contains additional generation-specific Java code details. These details are separated from the Ecore model to keep it free of such “low-level” information and settings.

#### 3.1 Modelling relevant aspects in EA

The first step is to load an existing metamodel into EA. A complete and automatic import of existing Ecore files in EA is currently not possible and therefore, *relevant parts* of the existing metamodel (`GenModel`) have to be modelled manually. Although this might sound frightening (especially for large, complex metamodels), the emphasis here on *relevant* indicates that only elements that are needed for the transformation have to be present in EA, where more can be added iteratively as the transformation grows.

If you find this section challenging or unclear, refer to Part II: Ecore for a detailed review of metamodel construction.

- ▶ Open Eclipse and create a new metamodel project named `EcoreToGenModel`, do not select the `Add Demo Specification` option in the project wizard window.
- ▶ A new specifications folder with the project name should have been loaded into the workspace.
- ▶ Double-click the generated `EcoreToGenModel.eap` file to open your project in EA. Explore the project browser and make note of the packages already present in EA under `eMoflon Languages`, especially `Ecore` which we shall use in this transformation.
- ▶ Select the root note `My Working Set` and create a new package named `GenModelLanguage`.

- ▶ Add a new Ecore diagram and model the elements as depicted in Fig. 3.1. You'll need to create the three EClasses on the left, but `Ecore::EPackage` and `Ecore::EClass` are to be drag-and-dropped and pasted as links from the project browser.

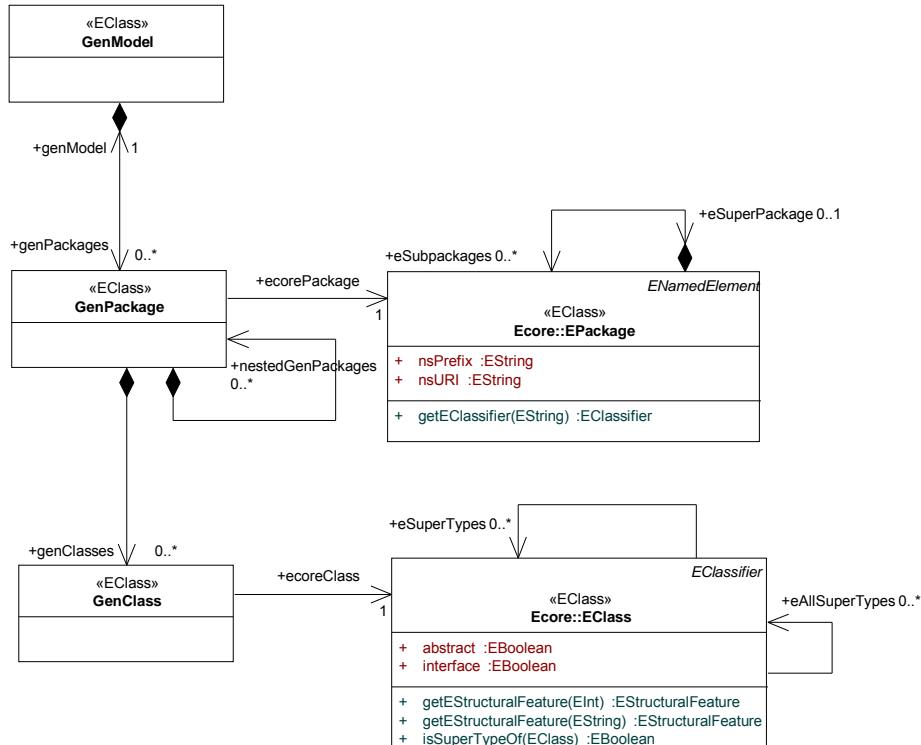
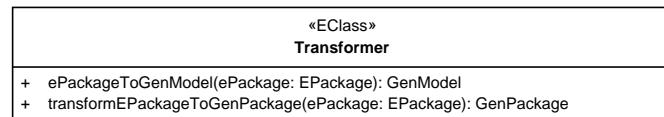
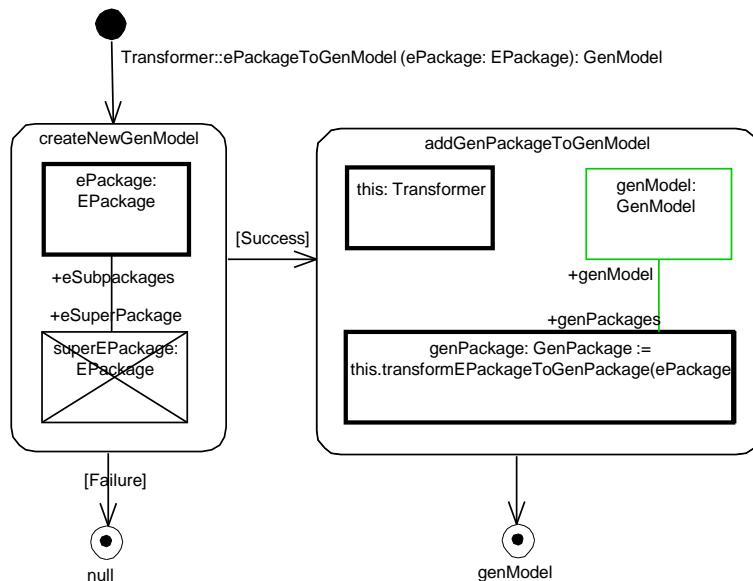


Figure 3.1: Metamodel of `GenModel`

- ▶ Please note that the actual `GenModel` metamodel contains many more elements, but this subset is sufficient for our task. Although this subset can be incomplete, it must be correct and not contradict the actual `GenModel` metamodel in any way!
- ▶ Navigate to the project browser again and create another package named `Ecore2GenModel`. This will contain the `Transformer` class; Create and complete its Ecore diagram as depicted in Fig. 3.2.
- ▶ Carefully double-click each method to create and implement their SDMs as depicted in Figs. 3.3 and 3.4.

Figure 3.2: Methods in **Transformer**Figure 3.3: Main method for **EPackage** to **GenModel** transformation

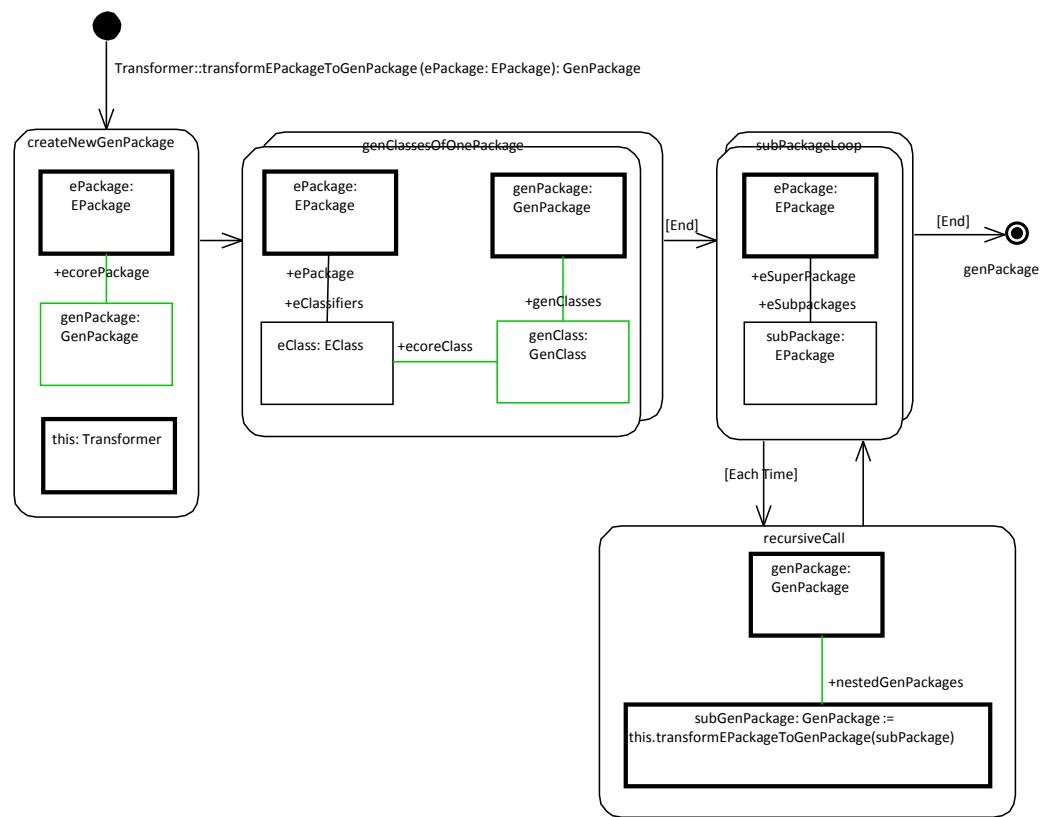


Figure 3.4: Helper function to transform all EPackages to GenPackages

### 3.2 Configuration for code generation in Eclipse

Since there is already generated code for the existing `GenModel` metamodel (provided via the Eclipse plugin), we do *not* want to export our incomplete subset of `GenModel` from EA. Instead, we need to configure Eclipse to access the elements specified in our partial metamodel from the complete metamodel.

- ▶ In EA, right-click your `GenModelLanguage` package and select “Properties...”
- ▶ Navigate to “Properties/Moflon” in the dialogue window and update the tagged `Moflon::Export` value to `false` (Fig. 3.5).

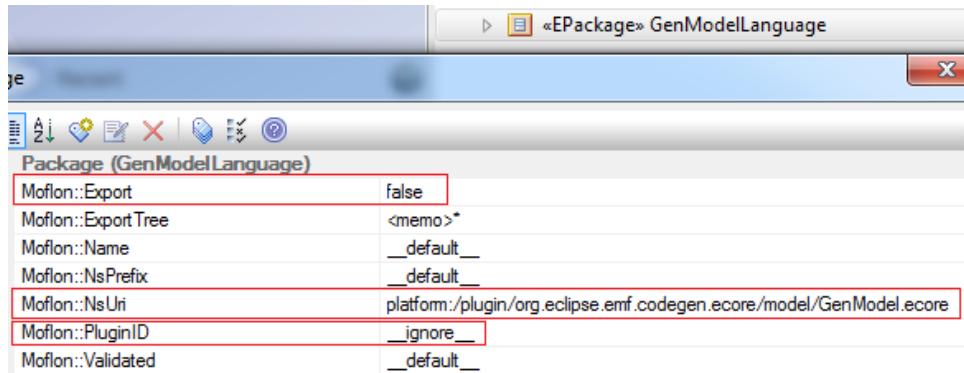


Figure 3.5: Update the `GenModel` export option and other tagged values

- ▶ Next we have to set the “real” URI of the project to be used in Eclipse so that the relevant references are exported properly. Set the value of `Moflon::NsUri` to `platform:/plugin/org.eclipse.-emf.codegen.ecore/model/GenModel.ecore`. As the default plugin ID generation provided by eMoflon is also not valid here, set the value of `Moflon::PluginID` to `__ignore__` (two underscores before and after!). The three relevant values to be set are shown in Fig. 3.5.
- ▶ Validate and export all projects as usual to your Eclipse workspace, and update the metamodel project by pressing F5 in the package explorer.
- ▶ Right-click `Ecore2GenModel` once more and navigate to “Plug-in Tools/Open Manifest.” The plug-in manager should have opened in the editor with a series of tabs at the bottom.

- ▶ Switch to the Dependencies tab. Press Add and enter `org.eclipse.emf.codegen.ecore`. This plug-in includes both the `Ecore` and `GenModel` libraries we require for successful compilation of the transformation code.

Although we have already specified the URI of the existing project (in this example, `GenModel`) as tagged project values, we still have to configure a few things for code generation.

- ▶ Expand the `Ecore2GenModel` project folder and open the `moflon.properties.xmi` file tree. Right-click the properties container, and create a new `Additional Dependencies` child. Double click the element to open its properties tab below the editor, and as shown in Fig. 3.6, update its `Value` to:

```
platform:/plugin/org.eclipse.emf.codegen.ecore/model/GenModel.ecore
```

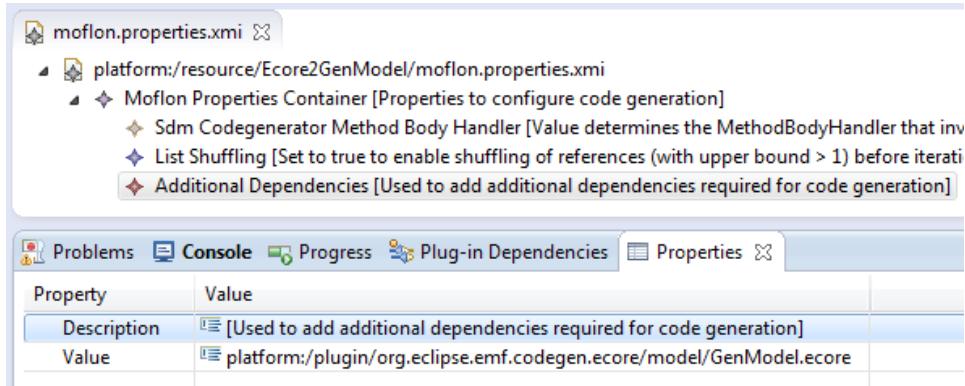


Figure 3.6: Setting properties for code generation

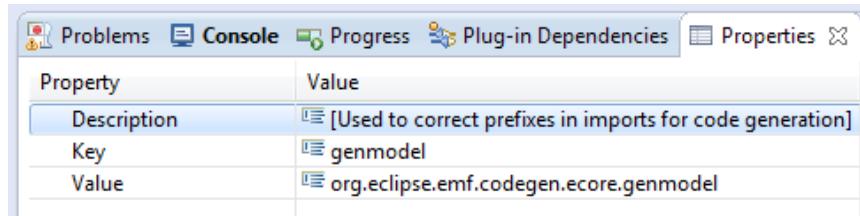
- ▶ Similarly, add a second `Used Gen Packages` child and set its value to:

```
platform:/plugin/org.eclipse.emf.codegen.ecore/model/GenModel.genmodel
```

---

Finally, to compensate for some cases where our naming conventions were violated, analogously add the following mapping as corrections:

- ▶ Add an *import mapping* child for correct generation of imports, setting the key as `genmodel` (depicted in Fig. 3.7) and value to:  
`org.eclipse.emf.codegen.ecore.genmodel`



The screenshot shows the Eclipse IDE's Properties view. The top bar has tabs: Problems, Console, Progress, Plug-in Dependencies, Properties (which is selected), and a close button. Below the tabs is a table with two columns: Property and Value. There are three rows in the table:

| Property    | Value                                                     |
|-------------|-----------------------------------------------------------|
| Description | [Used to correct prefixes in imports for code generation] |
| Key         | genmodel                                                  |
| Value       | org.eclipse.emf.codegen.ecore.genmodel                    |

Figure 3.7: Correcting default conventions for generating imports

- ▶ Finally, add a *factory mapping* to ensure that `GenModelFactory` is used as the factory for creating elements in the transformation instead of `GenmodelFactory`, which would be the default convention. Set its key as `genmodel`, and its value to: `GenModelFactory`.
- ▶ Its now time to generate code for the project. If everything worked out and the generated code compiles, you can ensure that the transformation behaves as expected by invoking the methods and transforming an ecore file to a corresponding genmodel.

As a final remark, note that import and factory mappings are not always necessary, `GenModel` is in this sense a particularly nasty example as it violates all our default conventions.

## 4 MOSL syntax

We have created this sheet for you to keep handy while working on your own projects. It describes many key statements in context-free EBNF grammar that you'll need to use when building metamodels, SDM patterns, and TGG rules.

Please keep in mind that this has been designed as a quick-reference page, and is not intended to be a teaching tool. If you are unsure how to use any of the following statements, the best reference for you will be the explanations in Part II, III, IV, or V where they are introduced and described in the context of an example.

### Basics

```

attribute_constraints := '==' | '!='
binding_operator := '++' | '--'
binding_semantics := '!' | '?'
binding_state := '@'
multiplicity := '0..1' | '0..*' | '1' | ...

```

### Metamodels

```

attribute := attribute_name : type
link := reference_name : target_type
operation := operation_name '(' param_list ')' ':' return_type
opposites := link '<->' link
parameter := parameter_name : type
param_list := parameter*
reference := [binding_operator] ['<>'] '-'> reference_name
 ('multiplicity') ':' target_type

attribute_name, ov_name, parameter_name, reference_name, return_type,
target_type, type := STRING

```

### SDM Patterns and TGG Rules

```

assignment := ov_name '.' attribute_name ':=' expression
statement_node := '<' method_call '>'
link_variable := [binding_operator] '-'> reference_name '->' source_type
object_variable := [binding_semantics] [binding_operator] [binding_state]
 ov_name ':' type_reference

AttributeValueExpression := ov_name '.' attribute_name
LiteralExpression := boolean_literal | integer_literal
 | any_literal
ObjectVariableExpression := '@' ov_name
ParameterExpression := '$' parameter_name
MethodCallExpression := (objectVariableExpression |
 ParameterExpression) '.' ID '(' argument_list ')'

boolean_literal := true, false
integer_literal := ['+' | '-'] ('1' | ... | '9')
any_literal := STRING

attribute_name, method_call, ov_name, parameter_name, reference_name,
source_type, type_reference := STRING

```

## 5 eMoflon in a Jar

This section describes how to package code generated with eMoflon into runnable Jar files, which is useful if you want to build applications for end-users.

We distinguish between repository, i.e., SDM-based, and integration, i.e., TGG-based, projects.

### 5.1 Packaging SDM projects into a Jar file

The following explanations use the demo specification that is shipped with eMoflon to explain the workflow of building a runnable Jar file.

- ▶ Open a fresh workspace and add to it the eMoflon Demo specification (visual or textual syntax) by selecting “File/New/Other...” and then “eMoflon/New Metamodel Wizard”. Do not forget to tick the “Add demo specification” checkbox!
- ▶ Generate code for the demo and verify the result by running the test cases in *DemoTestSuite*.
- ▶ Add a suitable main method to *NodeTest*, for instance:

```
public static void main(String[] args) {
 System.out.println("Begin of test runs");
 new NodeTest().testDeleteNode();
 new NodeTest().testInsertNodeAfter();
 new NodeTest().testInsertNodeBefore();
 System.out.println("End of test runs");
}
```

- ▶ Run *NodeTest* as “Java Application” (*not* as “JUnit Test”). Now you have a new launch configuration named “*NodeTest*”.
- ▶ Now, select the repository project (containing the generated code) and the project *DemoTestSuite*. You do not need to add the project containing the EA project. Right-click and select “Export...”. Choose “Runnable JAR file”.
- ▶ On the next page, select the launch configuration you just created by running *NodeTest* and an appropriate target location for your Jar file. The libraries should be packaged or extracted into the generated Jar file.
- ▶ Afterwards, open up a console in the folder containing the generated Jar file and execute it as follows:

```
java -jar [GeneratedJarFile.jar]
```

## 5.2 Packaging TGG projects into a Jar file

In the following, you will create a runnable Jar from a TGG specification. We assume that you have some existing TGG implementation and that you want to execute the `main` method in class `org.moflon.tie.MyIntegrationTrafo`.

*Note:* The following instructions show how to use Eclipse's built-in facility for generating runnable Jars. There are other build tools such as ant, Maven or Gradle that facilitate this process.

- ▶ Ensure that your TGG rules from within Eclipse. For simplicity, we assume that your main method currently resembles the following snippet:

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

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

The default transformation helper should print a short success message when the forward transformation has finished.

- ▶ Before packaging your project, you have to change the ways how the TGG rules are being loaded (in the constructor of `MyIntegrationTrafo`). Replace this method call

```
loadRulesFromProject("...");
```

with

```
File jarFile = new File(MyIntegrationTrafo.class
 .getProtectionDomain().getCodeSource()
 .getLocation().toURI().getPath());
loadRulesFromJarArchive(
 jarFile,
 "/MyIntegration.sma.xmi");
```

This is a tiny trick to find out the name of the Jar file that you are about to build. If you already know the name of your Jar file (e.g., “`tggInAJar.jar`”), you could simply use the following code:

```
loadRulesFromJarArchive(
 "tggInAJar.jar",
 "/MyIntegrationTrafo.sma.xmi");
```

- ▶ Next, make the “model” directory a source folder by right-clicking it and selecting “Build Path/Use as Source Folder”. This will make the contents of “model” available in the Jar file to be built.
- ▶ Now, your projects are ready to be packaged. Select all projects that are involved in your TGG, that is, the project of the source and target metamodel as well as the actual integration project.  
Right-click the projects and select “Export...” and then “Java/Runnable JAR File”.
- ▶ Select the appropriate launch configuration (named “MyIntegrationTraf”), choose the export destination, and make sure that the library handling is set to “Extract required libraries”.
- ▶ After a successful export, locate the generated Jar file. The program expects to find the source model of the transformation at the following path, relative to the folder containing your Jar file: “instances/fwd.src.xmi”.

Now, let's take the transformation for a spin:

```
java -jar [GeneratedJarFile.jar]
```

## 6 Useful shortcuts

This page is a simple list of special hotkeys you might find useful while working with eMoflon in either EA or Eclipse. Please note standard shortcuts, such as **Ctrl + S** and **Ctrl + Z**, are still applicable in most cases.

### 6.1 In Eclipse (general)

Note: I indicates *in Integrator window*, and GK indicates *German keyboards only*

|                                             |                            |
|---------------------------------------------|----------------------------|
| <b>Ctrl + Space</b>                         | Auto-type completion       |
| <b>Ctrl + 1</b> (problems tab)              | Quick-fix menu             |
| <b>Alt + arrow (I)</b>                      | Proceed to next step       |
| <b>Shift + Alt + arrow (I)</b>              | Fast navigation            |
| <b>Shift + Ctrl + Alt + arrow (I)</b>       | Proceed to next breakpoint |
| <b>Shift + Ctrl + AltGr + arrow (I, GK)</b> | Proceed to next breakpoint |

### 6.2 In Eclipse (eMoflon Plugin)

|                           |                                             |
|---------------------------|---------------------------------------------|
| <b>Alt + Shift + E</b>    | Open list of all available eMoflon commands |
| <b>Alt + Shift + E, B</b> | Trigger a build without clean               |
| <b>Alt + Shift + E, C</b> | Trigger a clean and build                   |
| <b>Alt + Shift + E, D</b> | Convert file to visual representation (dot) |
| <b>Alt + Shift + E, G</b> | Start integrator (on correspondence file)   |
| <b>Alt + Shift + E, I</b> | Create/update injections                    |
| <b>Alt + Shift + E, M</b> | Convert project to textual syntax           |
| <b>Alt + Shift + E, P</b> | Add ANTLR parser and/or unparser            |
| <b>Alt + Shift + E, V</b> | Validate Ecore file                         |
| <b>Alt + Shift + E, X</b> | Export and build EAP file                   |

### 6.3 In EA

Note: D indicates *in Diagram*, and PB indicates *in Project Browser*

|                                    |                                             |
|------------------------------------|---------------------------------------------|
| <b>Alt + Enter</b>                 | Selected element <b>Properties</b> dialogue |
| <b>F9 + EClass</b>                 | Class <b>Attribute</b> editor               |
| <b>F10 + EClass</b>                | Class <b>Operations</b> editor              |
| <b>Space (D)</b>                   | Current toolbar menu                        |
| <b>Del + Ctrl + element (D/PB)</b> | Delete element from model                   |
| <b>Ctrl + element (D/PB)</b>       | Duplicate and create new element            |
| <b>Ctrl + Alt + A</b>              | Open Model Search Window                    |
| <b>Alt + G + element (D)</b>       | Highlight Element (PB)                      |

## 7 Legacy support for CodeGen2

Since eMoflon 1.8, the default code generator is *Democles*. The previous code generator *CodeGen2* is available, but no longer officially supported.

This section describes how to configure your project to use CodeGen2.

- Install the eMoflon CodeGen2 feature: Select “Help/Install new software...”, choose the eMoflon update site and tick “eMoflon CodeGen2” (Fig. 7.1). Proceed with “Next” and follow the instructions to install the feature.

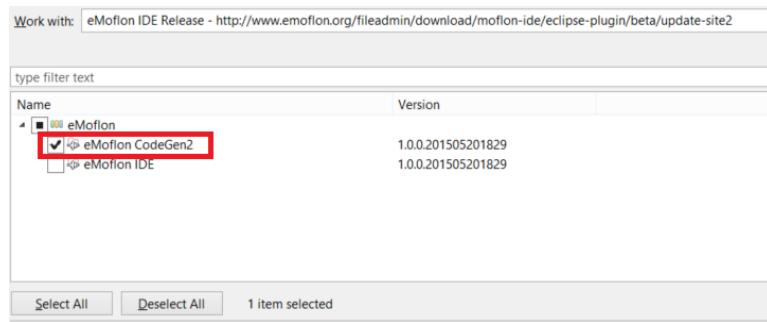


Figure 7.1: Installing the eMoflon CodeGen2 feature

- Open the file “moflon.properties.xmi” in your project(s) and set the code generation strategy to CODEGEN2 (Fig. 7.2).



Figure 7.2: Code generation strategy selection in “moflon.properties.xmi”

- Open the file “META-INF/MANIFEST.MF” in your project(s) and add the following dependency: *org.moflon.sdm.codegen2.runtime* (Fig. 7.3).
- Clean and build your project using the eMoflon context menu (Alt+Shift+E, B).

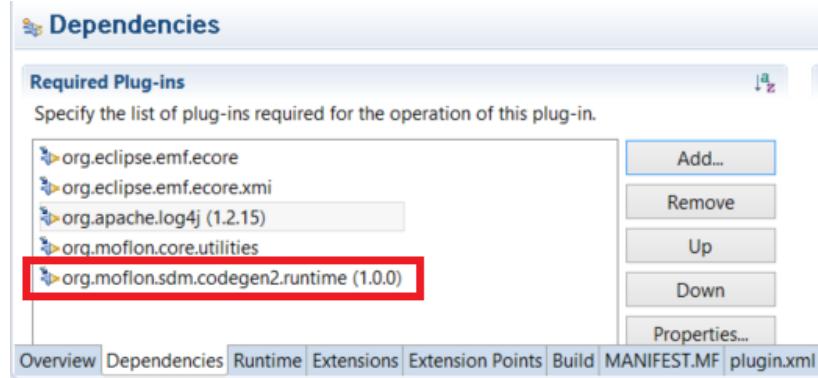


Figure 7.3: Dependency to CodeGen2 runtime in “MANIFEST.MF”

## 8 Creating and Using Enumerations

This section describes how to create enumeration types in your metamodel. For illustration purposes, we use the linked-list demonstration project from Part I.

Suppose we want to assign one of several predefined colors (e.g., red, green, blue) to each `Node` in a `List`. To represent the colors, we create an enumeration called `Color` with three elements: `Color.RED`, `Color.GREEN` and `Color.BLUE`.

The following sections show how to create and use enumerations in our visual and textual syntax.

## 8.1 Creating Enums in Enterprise Architect

- ▶ Let's start with the double-linked list demonstration specification, which is readily provided with eMoflon: Create a new meta-model project ("File/New/Other...", then "eMoflon/New Metamodel Wizard"), call your project "Demo", and tick "Add Demo Specification".
- ▶ Open the EAP file "Demo.eap" and navigate to the diagram "org.-moflon.demo.doublelinkedlist".
- ▶ Now, add a new "EEnum" type called color. This can be done via the Toolbox ("Diagram/Toolbox") or by pressing space while the cursor is inside the diagram area. Your diagram should resemble Figure 8.1.

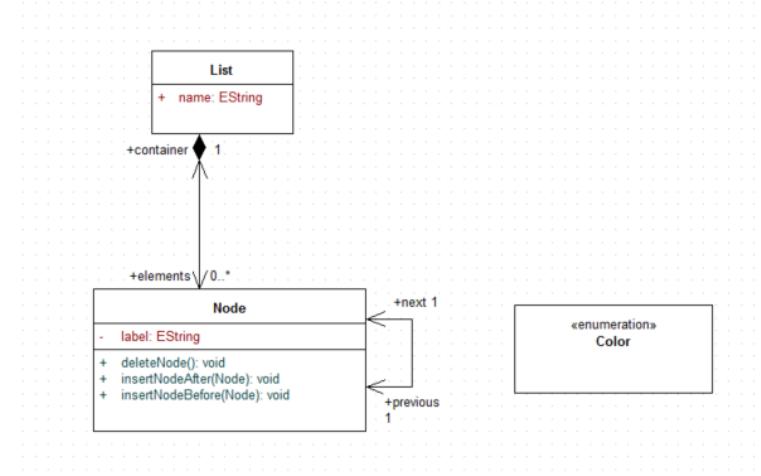
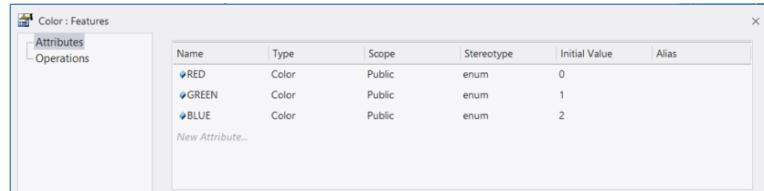


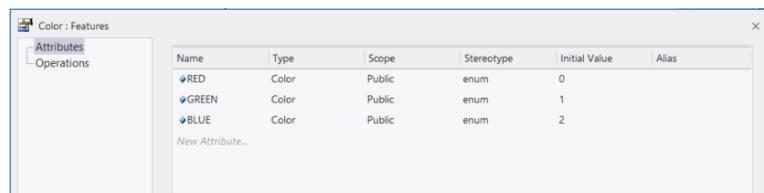
Figure 8.1: Creating a new EEnum

- ▶ We now create the three colors that our list nodes may have. An enum constant is a special attribute. Therefore, open the attributes view of **Color** ("Right-click/Features & Properties/Attributes...") and add three attributes as shown in Figure 8.2. Make sure that the type of the attributes is **Color** and that each attribute has an "initial value".
- ▶ Finally, create a **color** attribute of type **Color** in EClass **Node** (Figure 8.3).
- ▶ Validate and export and build your metamodel.
- ▶ A minimal test for the new feature could be implemented in the class **NodeTest** as follows:



| Name  | Type  | Scope  | Stereotype | Initial Value | Alias |
|-------|-------|--------|------------|---------------|-------|
| RED   | Color | Public | enum       | 0             |       |
| GREEN | Color | Public | enum       | 1             |       |
| BLUE  | Color | Public | enum       | 2             |       |

Figure 8.2: Creating the three color attributes RED, GREEN, and BLUE



| Name  | Type  | Scope  | Stereotype | Initial Value | Alias |
|-------|-------|--------|------------|---------------|-------|
| RED   | Color | Public | enum       | 0             |       |
| GREEN | Color | Public | enum       | 1             |       |
| BLUE  | Color | Public | enum       | 2             |       |

Figure 8.3: color attribute of EClass Node

Listing 8.1: Test for coloring nodes

```
@Test
public void testAddColor() throws Exception {
 Node node =
 DoublelinkedlistFactory.eINSTANCE.createNode();
 node.setColor(Color.RED);
}
```

## 8.2 Creating Enums

Coming soon...

## 9 Glossary

**Abstract Syntax** Defines the valid static structure of members of a language.

**Activity** Top-most element of an SDM.

**Activity Edge** A directed connection between activity nodes describing the control flow within an activity.

**Activity Node** Represents atomic steps in the control flow of an SDM. Can be either a story node or statement node.

**Assignments** Used to set attributes of object variables.

**Attribute Constraint** A non-structural constraint that must be satisfied for a story pattern to match. Can be either an assertion or assignment.

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

**Binding State** Can be either *bound* or *unbound/free*. See *Bound vs Unbound*.

**Binding operator** Determine whether a variable is to be *checked*, *created*, or *destroyed* during pattern matching.

**Binding Semantics** Determines if an object variable *must* exist (*mandatory*), may not exist (*negative*; see *NAC*), or is *optional* during *pattern matching*.

**Bound vs Unbound** Bound variables are completely determined by the current context, whereas unbound (free) variables have to be determined by the *pattern matcher*. `this` and parameter values are always bound.

**Concrete Syntax** How members of a language are represented. This is often done textually or visually.

**Constraint Language** Typically used to specify complex constraints (as part of the static semantics of a language) that cannot be expressed in a metamodel.

**Correspondence Types** Connect classes of the source and target metamodels.

**Dangling Edges** An edge with no target or source. Graphs with dangling edges are invalid, which is why dangling edges are avoided and automatically deleted by the pattern matching engine.

**Dynamic Semantics** Defines the dynamic behaviour for members of a language.

**EA** Enterprise Architect; The UML visual modeling tool used as our visual frontend.

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

**Edge Guards** Refine the control flow in an activity by guarding activity edges with a condition that must be satisfied for the activity edge to be taken.

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

**Exogenous** Transformations between models in different languages (i.e., unique metamodel instances).

**Grammar** A set of rules that can be used to generate a language.

**Graph Grammar** A grammar that describes a graph language. This can be used instead of a metamodel or type graph to define the abstract syntax of a language.

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

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

**Link or correspondence Metamodel** Comprised of all correspondence types.

**Link Variable** Placeholders for links between matched objects.

**Literal Expression** Represents literals such as true, false, 7, or “foo.”

**Meta-Language** A language that can be used to define another language.

**Meta-metamodel** A *modeling language* for specifying metamodels.

**Metamodel** Defines the abstract syntax of a language including some aspects of the static semantics such as multiplicities.

**MethodCallExpression** Used to invoke any method.

**Model** Graphs which conform to some metamodel.

**Modelling Language** Used to specify languages. Typically contains concepts such as classes and connections between classes.

**Monotonic** In the context of TGGs, a non-deleting characteristic.

**NAC** Negative Application Condition; Used to specify structures that must not be present for a transformation rule to be applied.

**Object Variable** Place holders for actual objects in the current model to be determined during pattern matching.

**ObjectVariableExpression** Used to reference other object variables.

**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.

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

**Parameter Expression** Used to refer to method parameters.

**(Graph) Pattern Matching** Process of assigning objects and links in a model to the object and link variables in a pattern in a type conform manner. This is also referred to as finding a match for the pattern in the given model.

**Statement Nodes** Used to invoke methods as part of the control flow in an activity.

**Static Semantics** Constraints members of a language must obey in addition to being conform to the abstract syntax of the language.

**Story Node** *Activity nodes* that contain *story patterns*.

**Story Pattern** Specifies a structural change of the model.

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

**Type Graph** The graph that defines all types and relations that form a language. Equivalent to a metamodel but without any static semantics.

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

**Unification** An extension of the object oriented “Everything is an object” principle, where everything is regarded as a model, even the metamodel which defines other models.

## 10 GNU GENERAL PUBLIC LICENSE



GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

### TERMS AND CONDITIONS

## 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

## 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

## 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

## 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

## 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

## 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; how-

ever, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

#### 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

#### 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

#### 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

#### 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

#### 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

#### 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

## 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

## 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the

combination as such.

**14. Revised Versions of this License.**

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

**15. Disclaimer of Warranty.**

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

**16. Limitation of Liability.**

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**17. Interpretation of Sections 15 and 16.**

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

#### END OF TERMS AND CONDITIONS

##### How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at

least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.> Copyright (C) <year>  
<name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lGPL.html>.