

An Introduction to Metamodelling and Graph Transformations

with eMoflon



Part III: Story Driven Modelling

For eMoflon Version 1.7.0

Copyright © 2011–2015 Real-Time Systems Lab, TU Darmstadt. Anthony Anjorin, Erika Burdon, Frederik Deckwerth, Roland Kluge, Marius Lauder, Erhan Leblebici, Daniel Tögel, David Marx, Lars Patzina, Sven Patzina, Alexander Schleich, 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 (February 2015)

Contents

1	Leitner's learning box reviewed	2
2	Transformations explained	4
3	Removing a card	8
4	Checking a card	26
5	Running the Leitner's Box GUI	44
6	Emptying a partition of all cards	45
7	Inverting a card	51
8	Growing the box	58
9	Conditional branching	68
10	A string representation of our learning box	75
11	Fast cards!	83
12	Reviewing eMoflon's expressions	93
13	Conclusion and next steps	95
	Glossary	96
	References	98

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*¹ 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, 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.

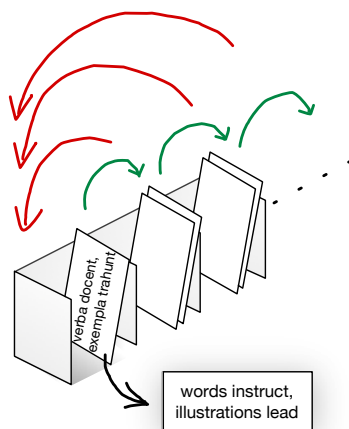


Figure 1: Static Structure of a Leitner's Learning Box

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. 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.

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

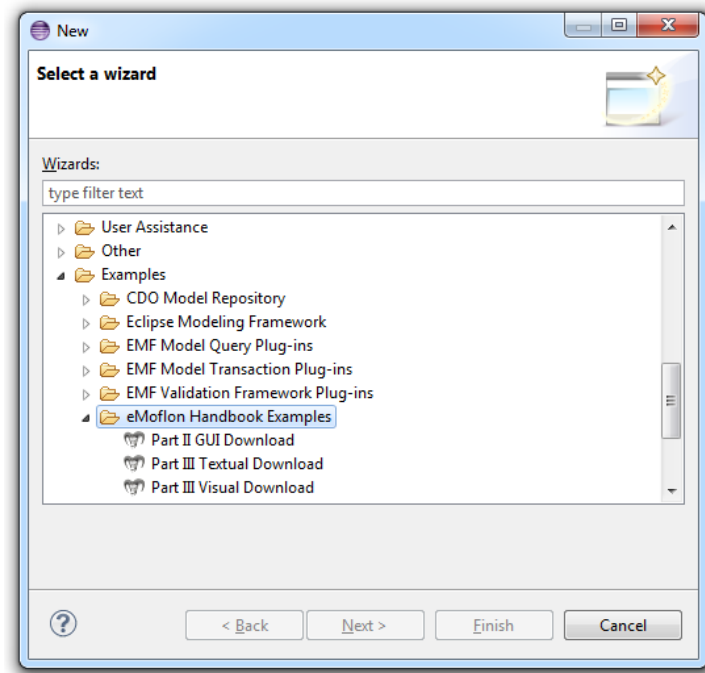


Figure 2: Choose a cheat package

- 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,² 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.

²To 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.3).

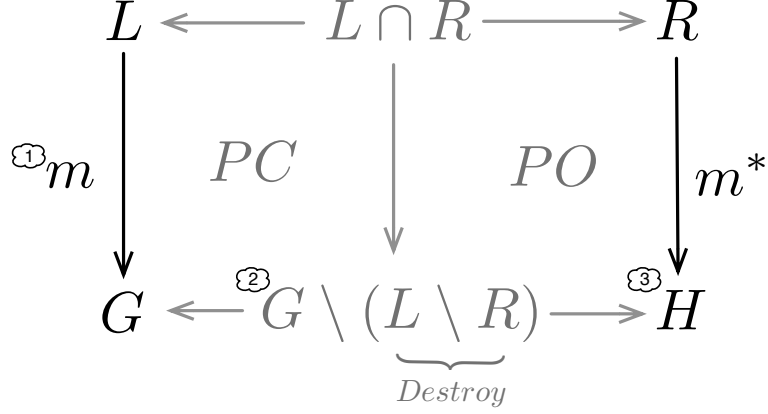


Figure 3: Applying a rule $r : (L, R)$ to G to yield H

Let's review this application.

(1) is determined by a process called *graph pattern matching* i.e., finding an occurrence or *match* of the precondition or *pattern* in the model G . *Pattern Matching*

(2) is determined by building a *push-out complement* $PC = (G \setminus \text{Destroy})$, such that $L \cup PC = G$.

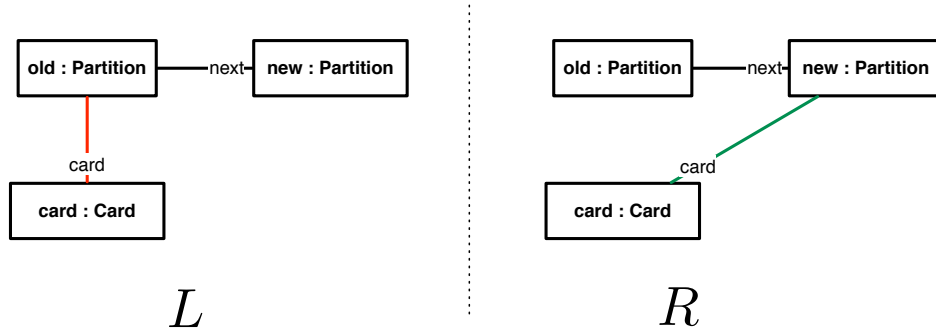
(3) is determined by building a *push-out* $PO = H$, so that $(G \setminus \text{Destroy}) \cup R = H$.

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.

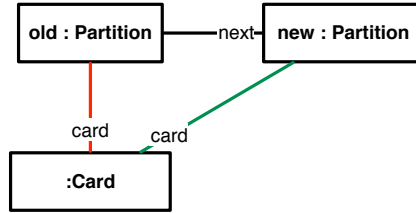
Since this black box formalisation involves two push-outs - one when cutting $\text{Destroy} := (L \setminus R)$ from G to yield $(G \setminus \text{Destroy})$ (deletion), and one when inserting $\text{Create} := (R \setminus L)$ in $(G \setminus \text{Destroy})$ to yield H (creation) - this construction 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 [2] 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 4 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 4: *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. 5).

Figure 5: 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. 6.

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* *Activity 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.

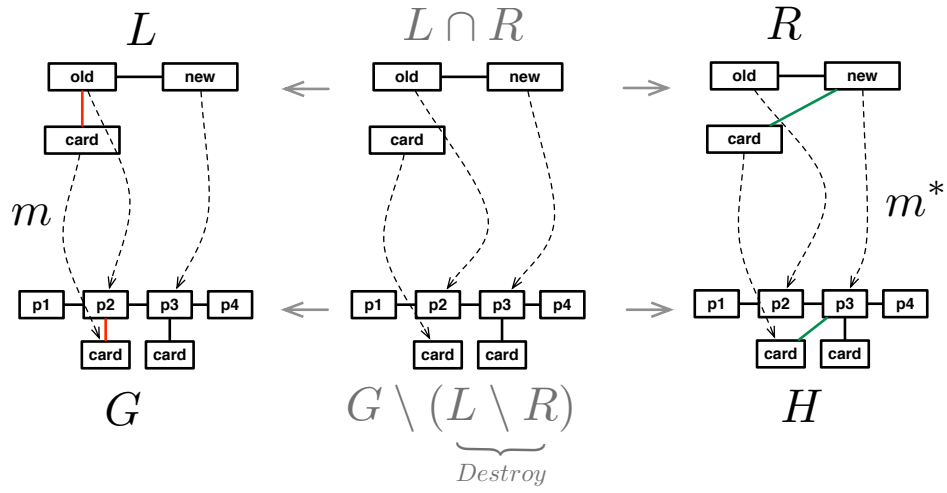


Figure 6: Applying *moveCard* to a learning box

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

3 Removing a card

Since we’re just getting started with SDMs,³ let’s re-implement the method previously specified directly in Java as an injection.⁴ 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. 7).

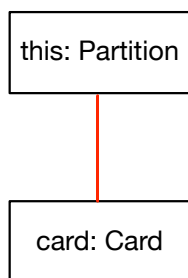


Figure 7: Removing a card from its partition

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 8).
- Now right-click on `LeitnersLearningBox` and go to “eMoflon/Clean(and 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.

³As 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.

⁴Refer to Part II, Section 6

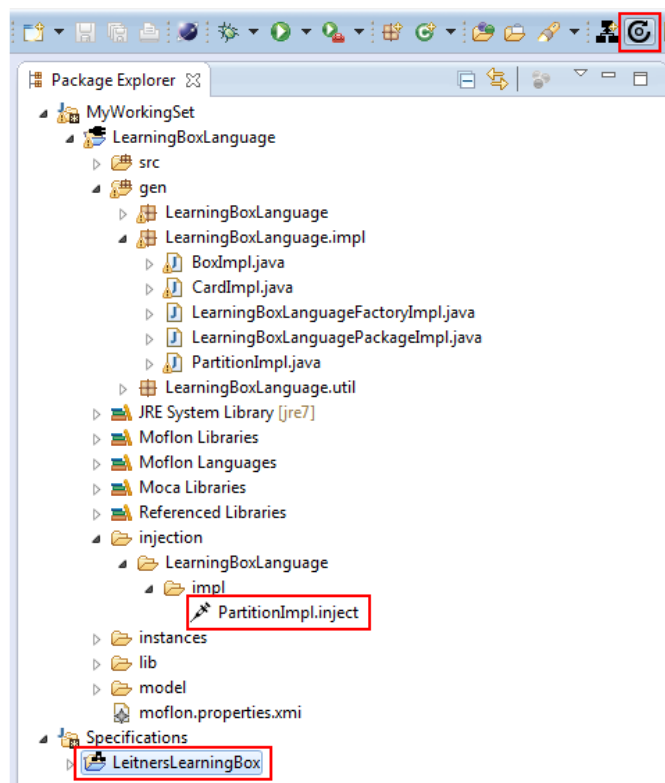


Figure 8: Remove injection content

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.⁵

⁵We 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 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. *Link Variable*

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 default, every object variable is *unbound*, or a *free variable*. Values for these variables can be determined automatically by the pattern matcher. By declaring an object variable that is to be *bound* however, it will have 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. *Binding State*
Free Variable
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. 9), and (3) *Double-click* on the chosen method to indicate that you want to implement it.

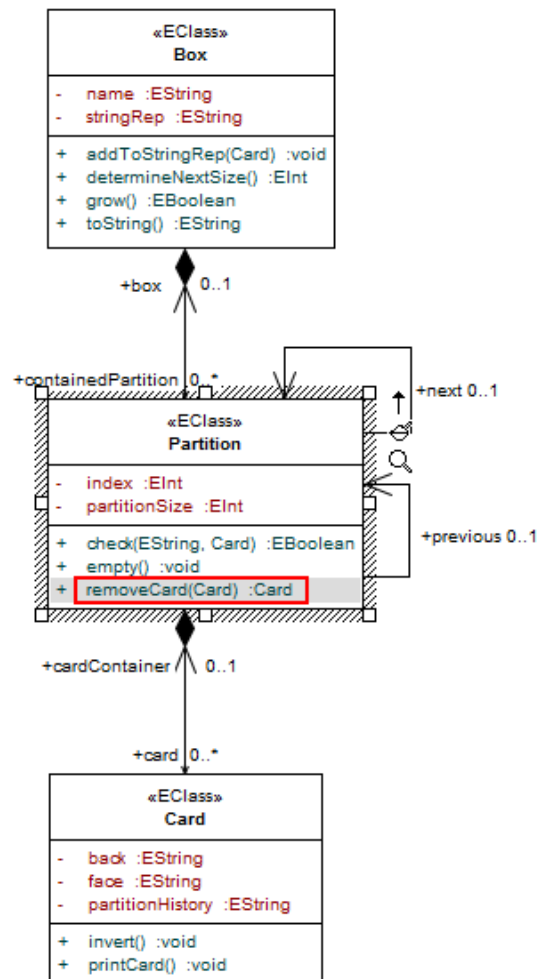


Figure 9: Double-click a method to implement it

- 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. 10).

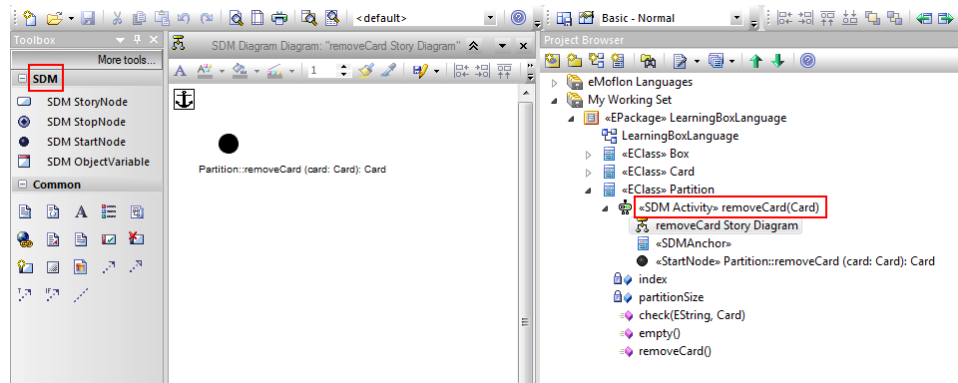


Figure 10: Generated SDM diagram and start node

- 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. 11).

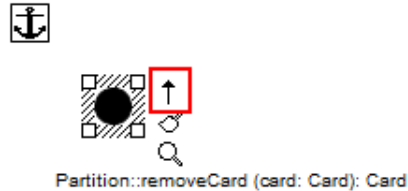


Figure 11: Quick link in SDM diagram to create new activity node

- Similar to quick linking,⁶ 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. 12, choose **Append StoryNode** to create a *Story Node*.

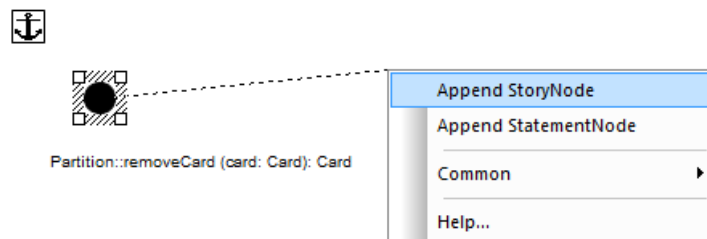


Figure 12: Create new activity node

- 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. 13).

⁶This was discussed in Part II, Section 2.5

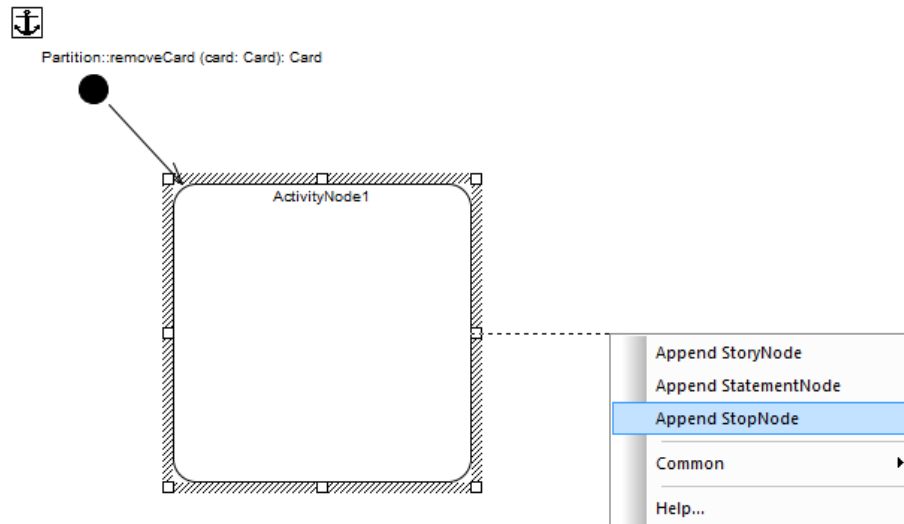


Figure 13: Complete the activity with a stop node

- ▶ 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 nodes. Statement nodes cannot and are used instead to invoke an 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. 14. 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. 15). A properties window will automatically appear (Fig. 16).

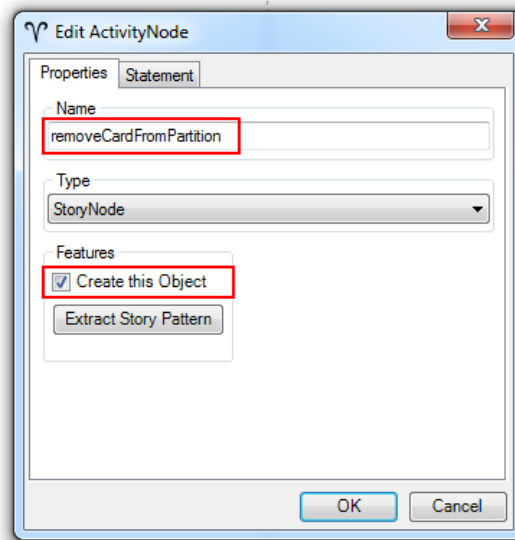


Figure 14: Initializing a story node

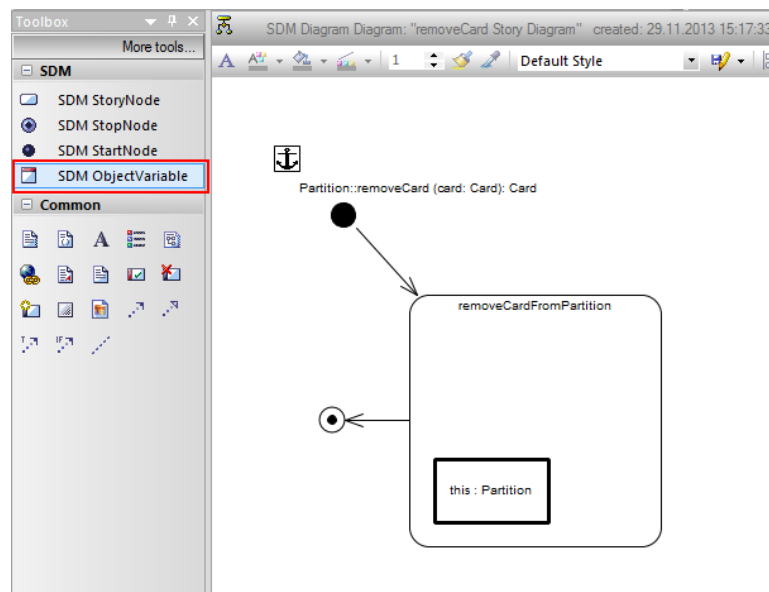


Figure 15: Add a new object variable from the toolbox

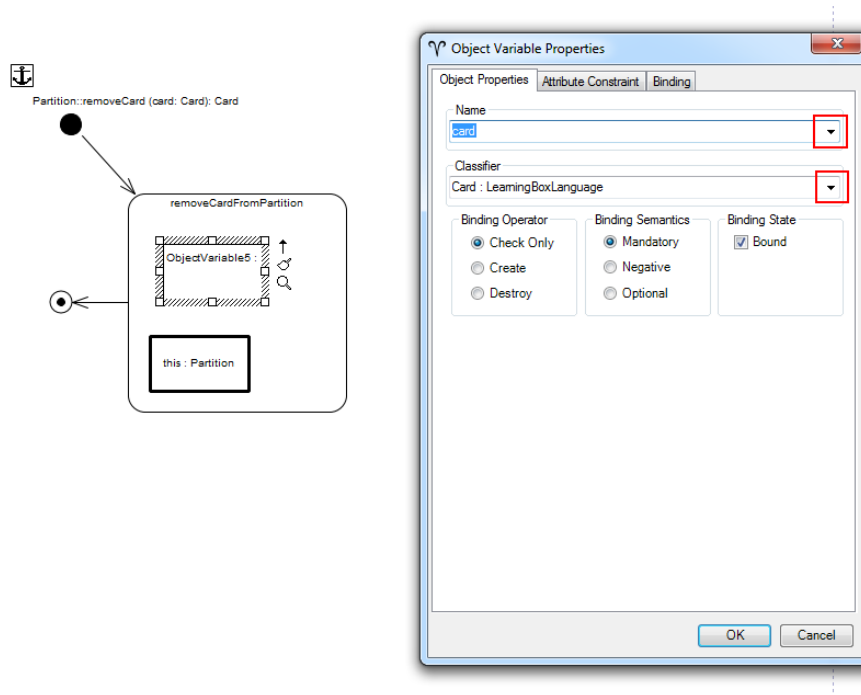


Figure 16: Specify properties of the added object variable

- Using the drop-down menus, choose **card** 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 **card** 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 **card** (Fig. 17).
- 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. 18). The reference names will automatically appear in the diagram.

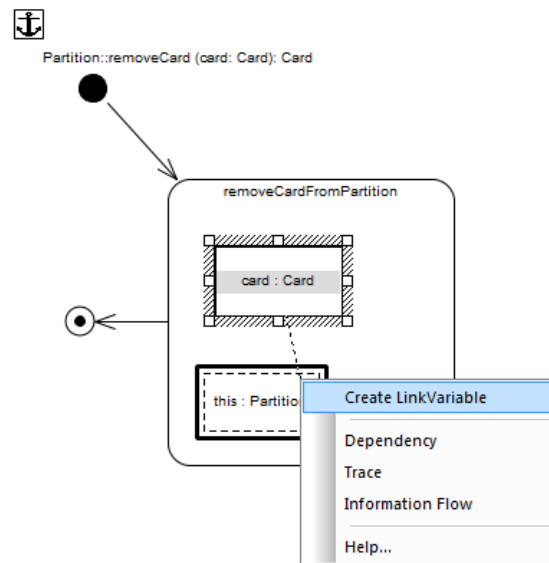


Figure 17: Create a link variable

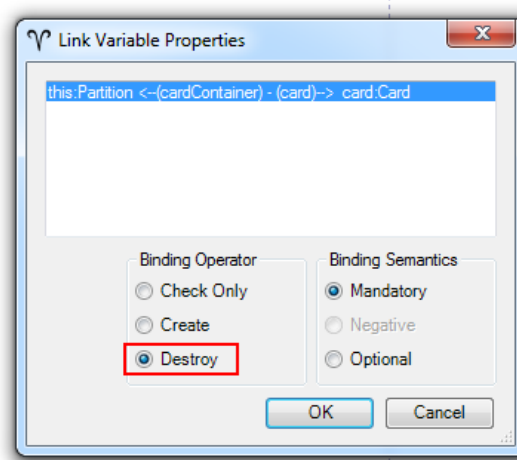


Figure 18: Specify properties for created link variable

- 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. 19, double-click the stop node to prompt the `Edit StopNode` dialogue.

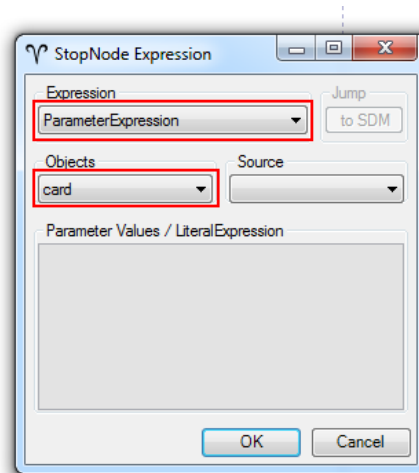


Figure 19: Adding a return value to the stop node

- 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 **card** 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. 20, 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,⁷ then build your metamodel's code from the package explorer.

⁷Go 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.

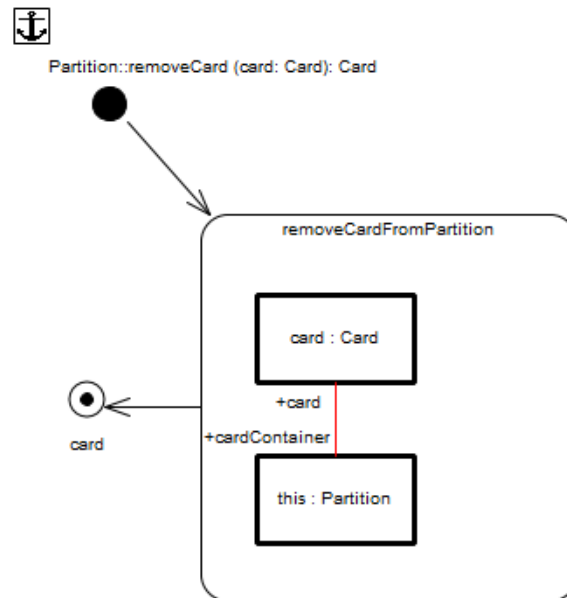


Figure 20: Complete SDM for `Partition::removeCard`

- If you're unable to export or generate code successfully, compare your SDM carefully with Fig. 20 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. 25 in the next section.

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.

- ▶ 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 Fig 21. 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 an *ObjectVariableExpression*. This expression implicitly refers to object variables from the preceding story node. In `removeCard`'s case, the returned object refers to the `card` from the pattern. *ObjectVariable-Expression*
- ▶ It should be mentioned that MOSL limits method definitions exclusively to the method's control flow. All actual transformation rules are modeled in separate *pattern* files. In this case, `removeCard`'s link deletion will be modeled 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. Well, this makes sense. You haven't created it yet! Click this message and press **Ctrl + 1** to invoke a "Quick Fix" dialogue (Fig 22). It offers to create a pattern file for you. Given that's exactly what you'd like, select the option and press **Finish**.
- ▶ The new file will open in the editor, and you'll be able to see a new directory structure under "LearningBoxLanguage/_patterns" (Fig. 23). 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.

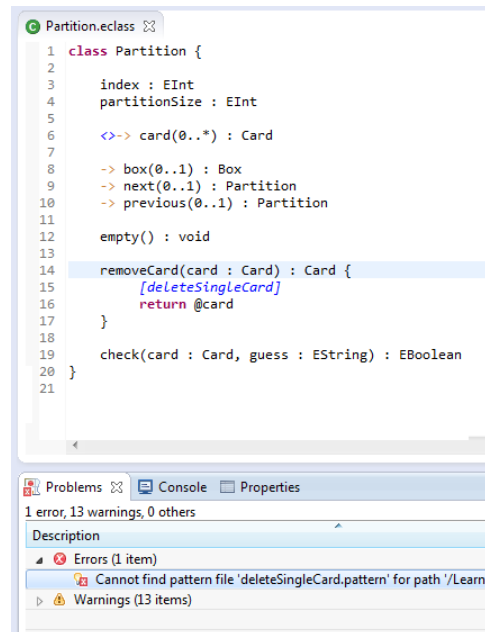
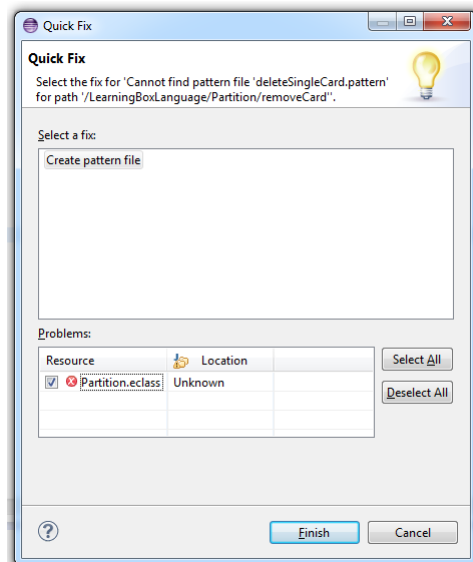
Figure 21: Control flow for `removeCard`

Figure 22: A quick fix to a missing pattern error

- 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*.

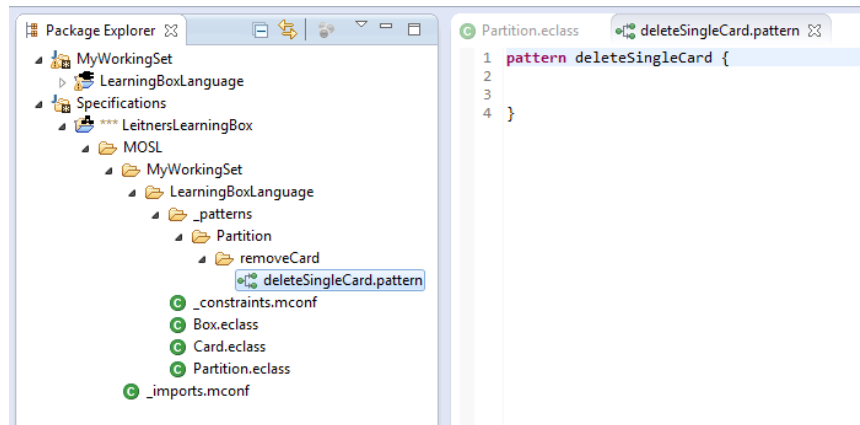
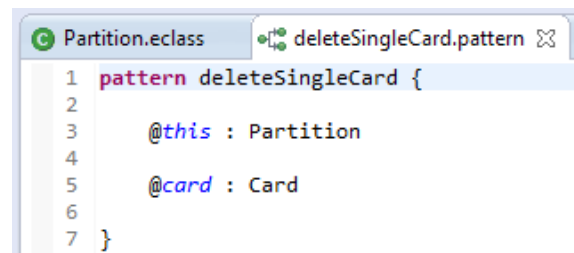


Figure 23: Directory structure for a pattern

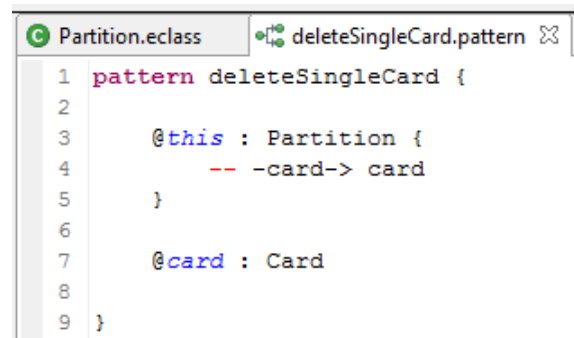
- Create two object variables, `@this:Partition` and `@card:Card` (Fig. 24). When working with MOSL patterns, ‘@’ again indicates that the variable is *bound*. `this` is bound to the object whose method is invoked, while `card` is bound to the value of the parameter with the same name.

Figure 24: Object variables for `removeCard`

- 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 Fig. 25.



```

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

```

Figure 25: 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 := '--' | '++' | '!'
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*,⁸ linked together as opposites in “LearningBoxLanguage/_constraints.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. 20 from the previous section.

⁸Technically 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 “LearningBoxLanguage/gen/LearningBoxLanguage/impl/PartitionImpl.java” to the `removeCard` declaration (approximately line 352). Inspect the generated implementation for your method (Fig. 26). Notice the null check that is automatically created - only a very conscientious (and probably slightly paranoid) programmer would program so defensively!

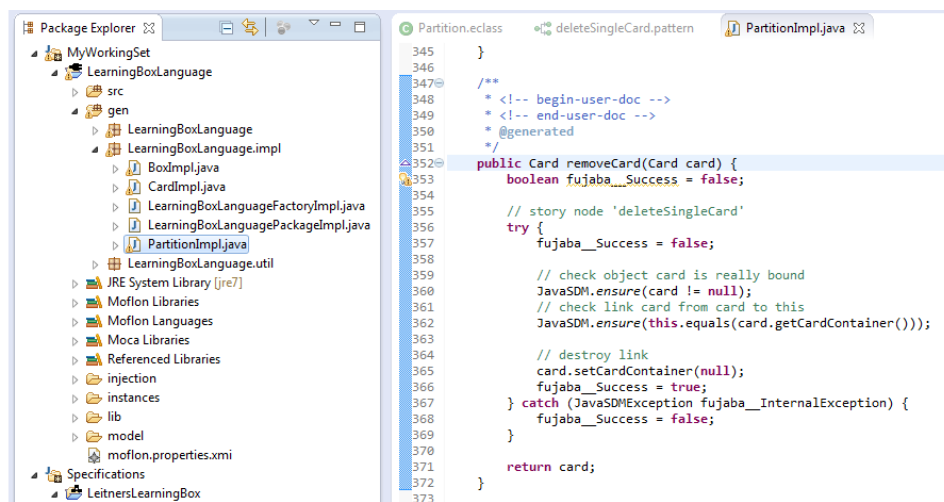


Figure 26: Generated implementation code

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,⁹ then go to any partition and select **Remove Card** (Fig. 27). 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?

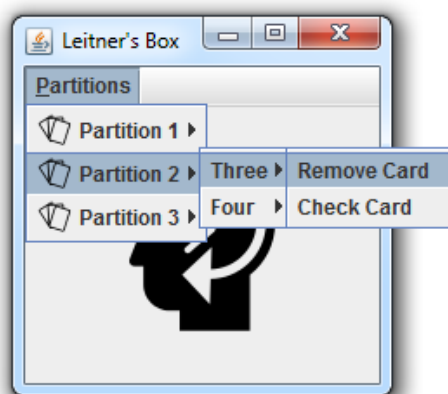


Figure 27: Testing `removeCard`

⁹Refer 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. 28). Following our rules established in Fig. 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.

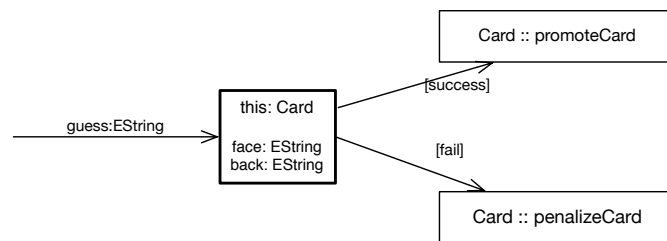


Figure 28: Checking a card with a guess

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. 29 briefly shows the intended create and destroy transformations.

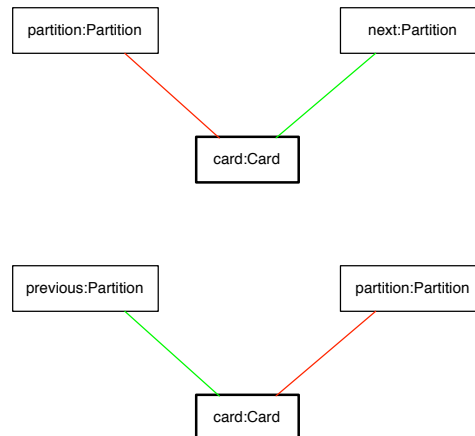


Figure 29: Promote (above) and penalize (below) card patterns

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. 30.

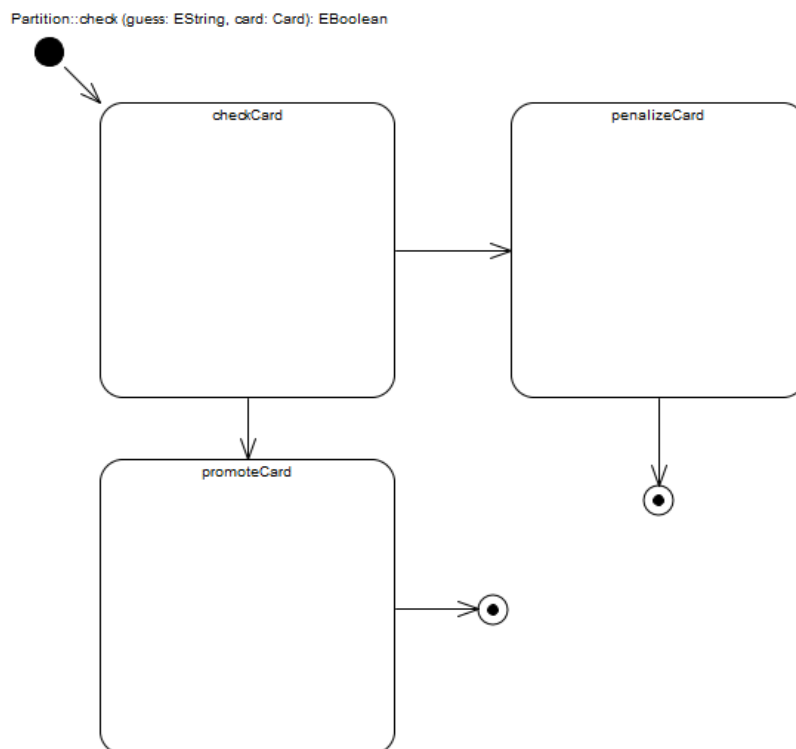


Figure 30: Activity diagram for `Partition::check`

- In `checkCard`, create an object variable that is bound to the parameter argument, `card` (Fig. 31). 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.

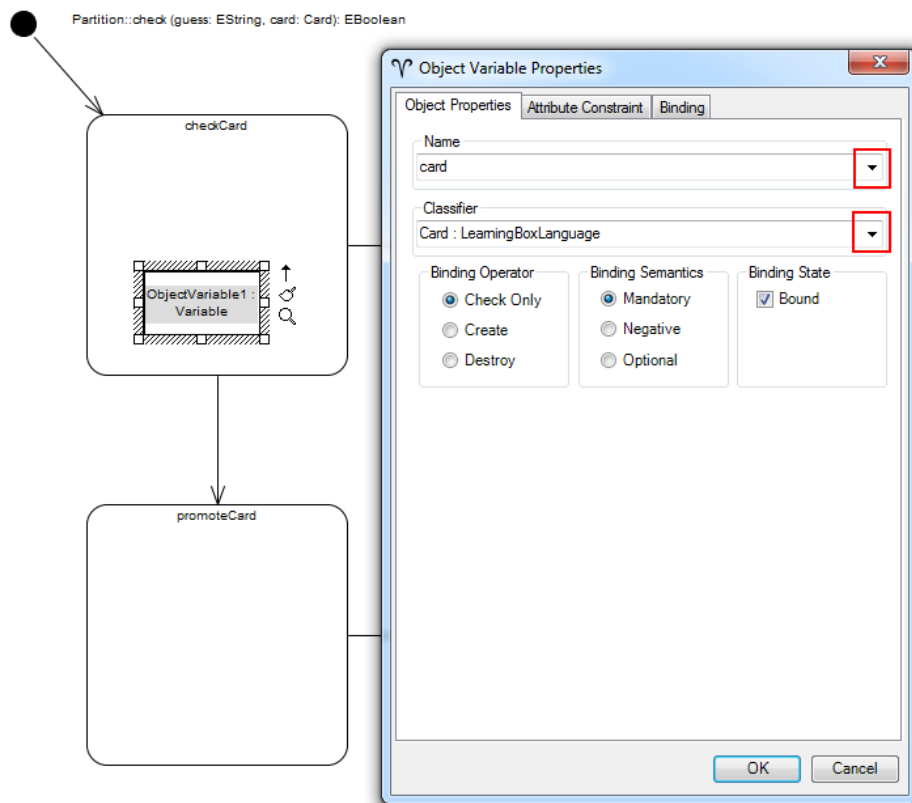


Figure 31: Creating the card variable

- 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. 32, and select the correct **Attribute** and **Operator**.

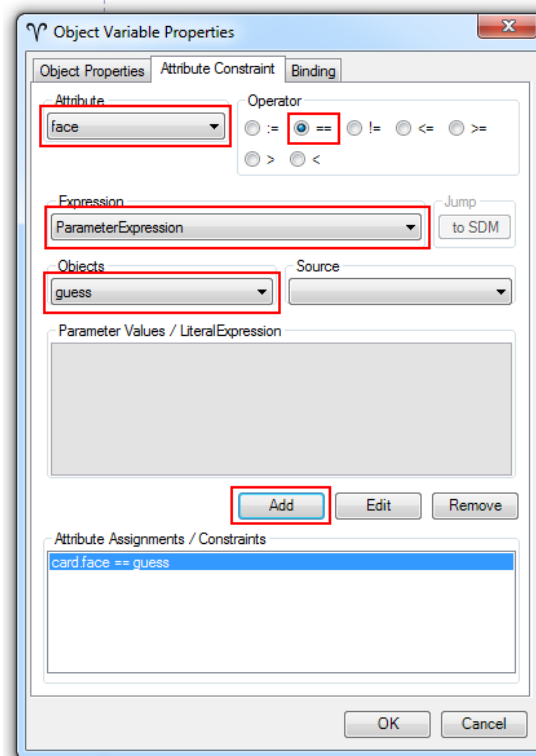


Figure 32: Creating an attribute constraint

- 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* construct based on the results of the *attribute constraint*.

*Edge
Guards*

- To add a guard to the edge leading from `checkIfGuessIsCorrect` to `penalizeCard`, double click the edge and set the *Guard Type* to *Failure* (Fig. 33). Repeat the process for the *Success* edge leading to `promoteCard`.

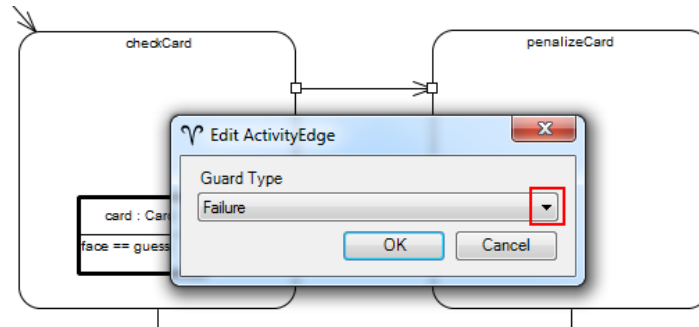


Figure 33: Add a transition with a guard

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. 34). Note the new diagram that is immediately created and opened in the project browser (Fig. 35).

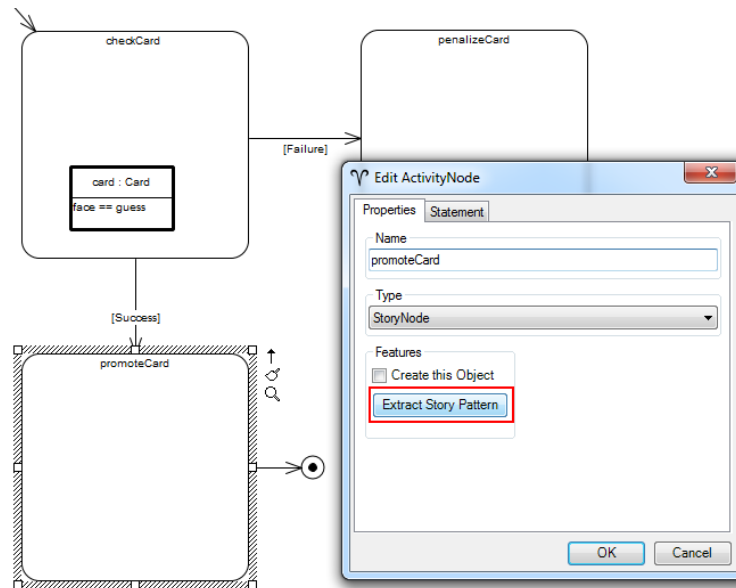


Figure 34: Extract a story pattern for more space and a better overview

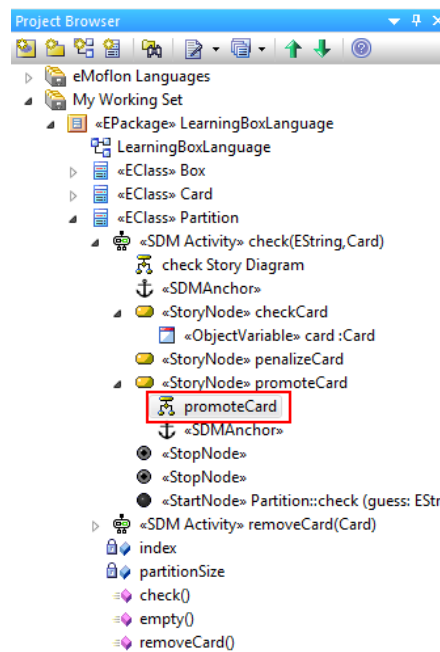


Figure 35: A new subdiagram is created automatically

Another EA gesture¹⁰ 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.¹¹

- 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. 36).

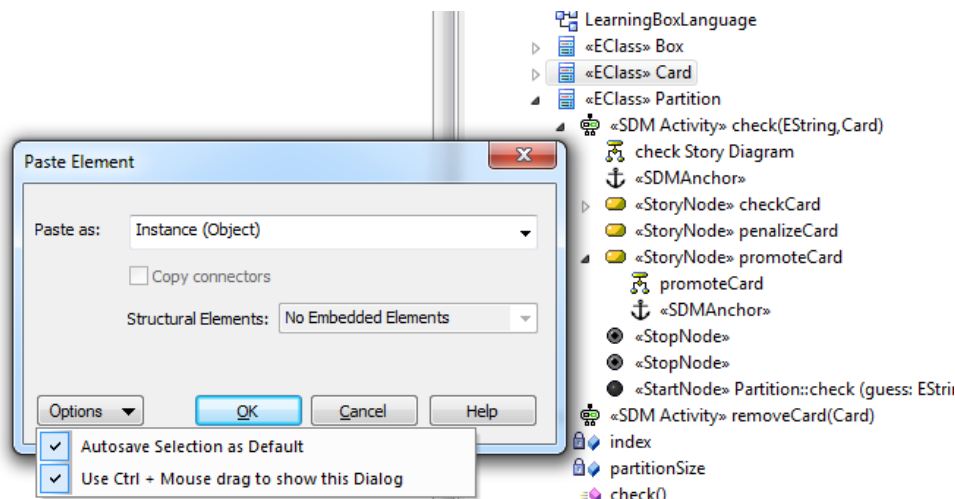


Figure 36: Add a new object variable per drag-and-drop

- A dialogue will appear asking what kind of visual element should be 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**

¹⁰The other two gestures we have learnt are “Quick Link” and “Quick Create”

¹¹Drag-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.

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. 37).

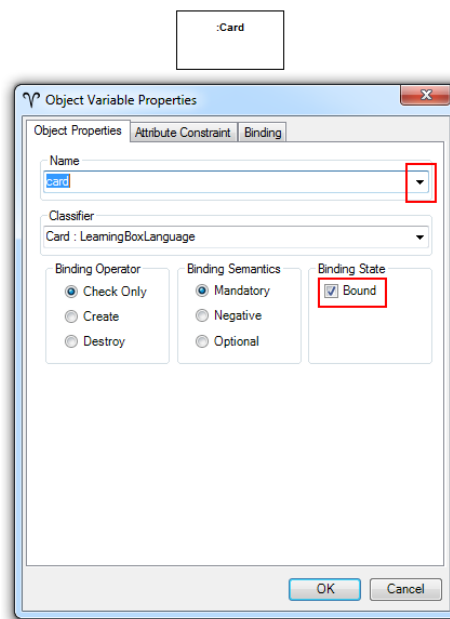


Figure 37: Object variable properties of the new card

- 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. 38.

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

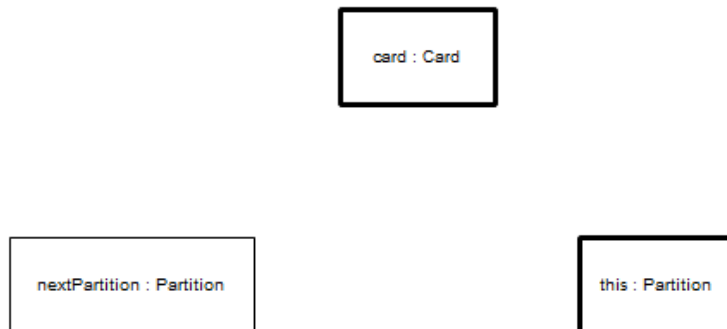
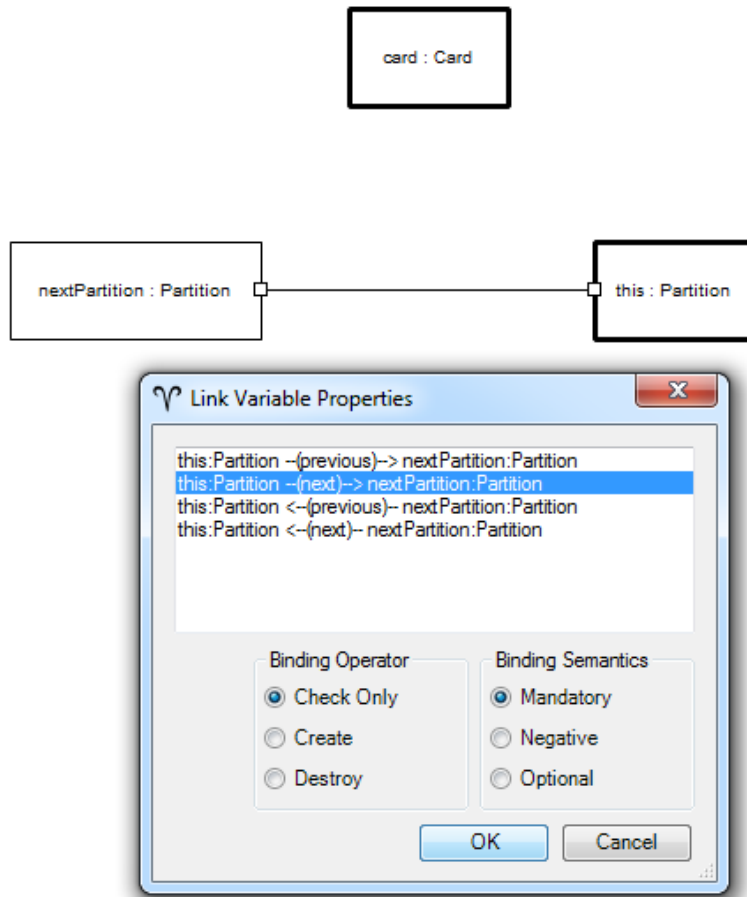
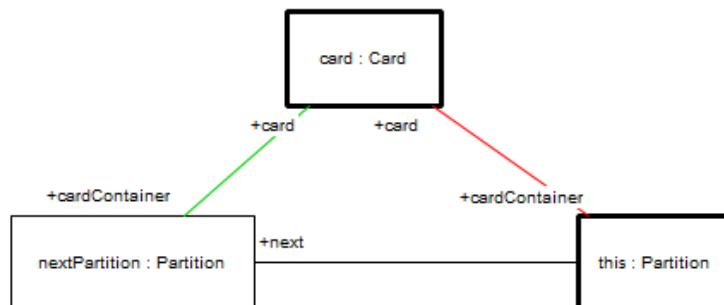


Figure 38: All object variables for story pattern `promoteCard`

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.

- To specify this, quick link from `this` to `nextPartition` (or vice-versa) to establish `next`, as shown in Fig. 39. 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`.
- 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. 40.

Figure 39: Possible links between `this` and `nextPartition`Figure 40: 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. 41. As you can see, this pattern is nearly identical to `promoteCard`, except it moves `card` to a `previousPartition`.

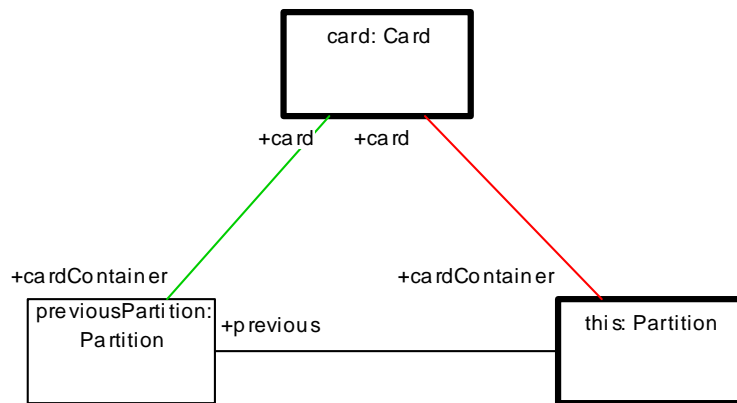


Figure 41: Complete story pattern for `penalizeCard`

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 dirty¹² and should be avoided when possible.

- To implement a literal, double click the stop node stemming from `promoteCard`, and change the expression type from `void` to `LiteralExpression` (Fig. 42). 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` fails¹³. This is handled by placing parallel Success

¹²It defeats, for example, any attempt to guarantee type safety

¹³For 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. 43.

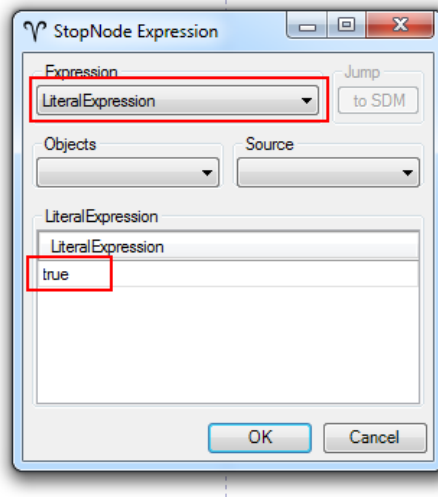


Figure 42: Add a return value with a literal expression

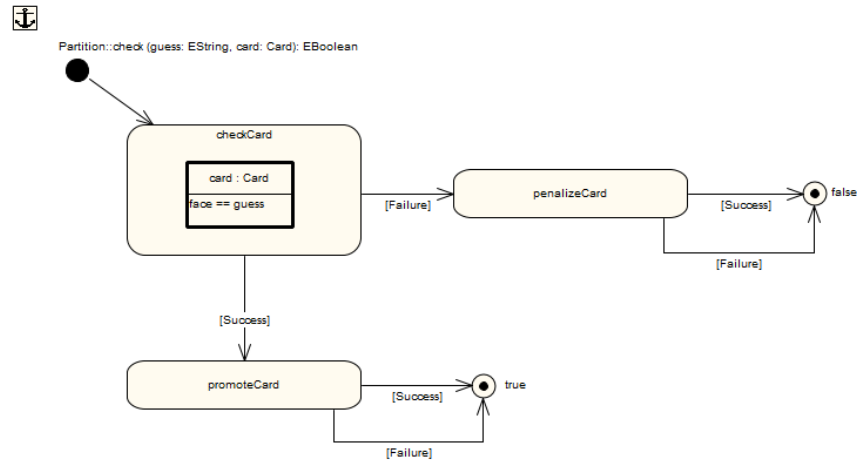


Figure 43: Complete SDM for `Partition::check`

-
- ▶ 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. 49 and 50 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 Fig. 44. Our auto completion feature includes a template for this.

```

19  check(card : Card, guess : EString) : EBoolean {
20      if [checkCard] {
21          [promoteCard]
22      } else {
23          [penalizeCard]
24      }
25  }
26 }
27

```

Figure 44: An if/else construct in `check`

- Upon saving, use the “Quick Fix” wizard again to generate the required pattern files. Your package explorer should now resemble Fig. 45.

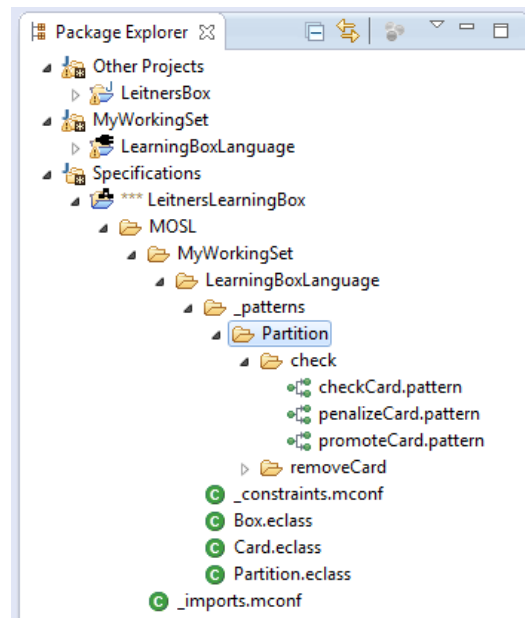


Figure 45: Project directory after creating patterns

- 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.¹⁴
- 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. 46.

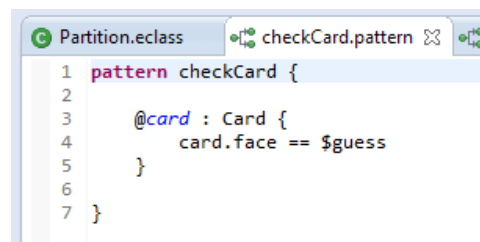


Figure 46: Completed `checkCard` pattern

- 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. 47.
- Given that `this` is bound while `next` partition is free, we need to establish a reference link between the `box` and partition. This will

¹⁴See the Quick Reference at the end of this part for a listing of all operators

allow the card to be moved around. Create an *outgoing link variable*, **next**, with target object variable **nextPartition**. Your file should now resemble Fig. 48.

```

1 pattern promoteCard {
2
3   @this : Partition
4
5   @card : Card
6
7   nextPartition : Partition
8
9 }
10

```

Figure 47: Object variables for promoting a card

```

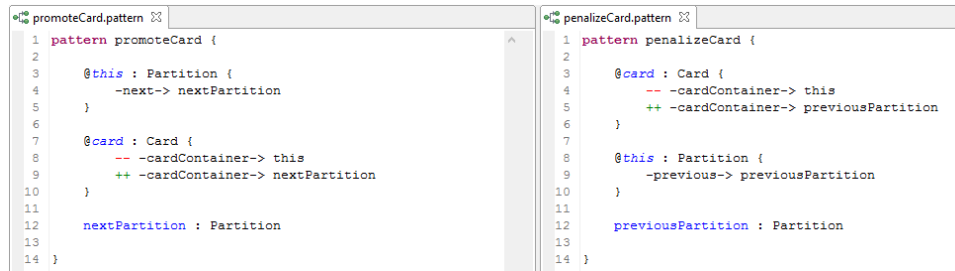
1 pattern promoteCard {
2
3   @this : Partition {
4     -next-> nextPartition
5   }
6
7   @card : Card
8
9   nextPartition : Partition
10
11 }

```

Figure 48: The @this object variable

- Finally, let's update the **cardContainer** reference of **card**. Simply delete the link to **this**, and create a new link to **nextPartition**.¹⁵ Your pattern is now complete.
- 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. 49.

¹⁵There are several different ways you could have implemented this movement, such as updating the link from **nextPartition** to **card**.



```

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

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

```

Figure 49: 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 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 Fig.50.
- 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.

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

Figure 50: Completed control flow for `check`

- 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. 43 for the control flow, and Figs. 40 and 41 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. 51).

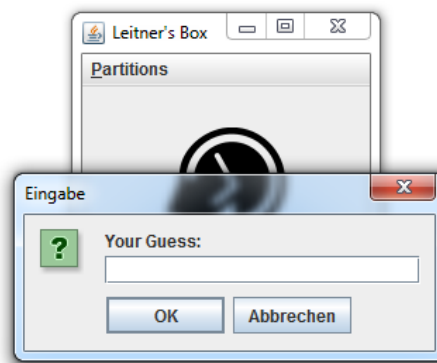


Figure 51: Enter your guess

- 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,¹⁶ we obviously need some construct for repeatedly deleting each card in the partition (Fig. 52).

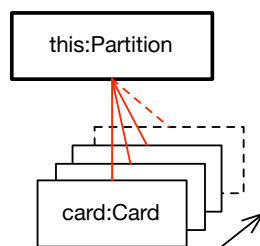


Figure 52: Emptying a partition of every card

In SDM, this is accomplished via a *for each* story node. It performs the 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? *For Each*

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.¹⁷ 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? *Dangling Edges*

¹⁶If there was only one, we would just invoke `removeCard`

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

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. 53). Press OK, and you'll see that a *for each* node is represented as a stacked node to indicate the potential for repetition.

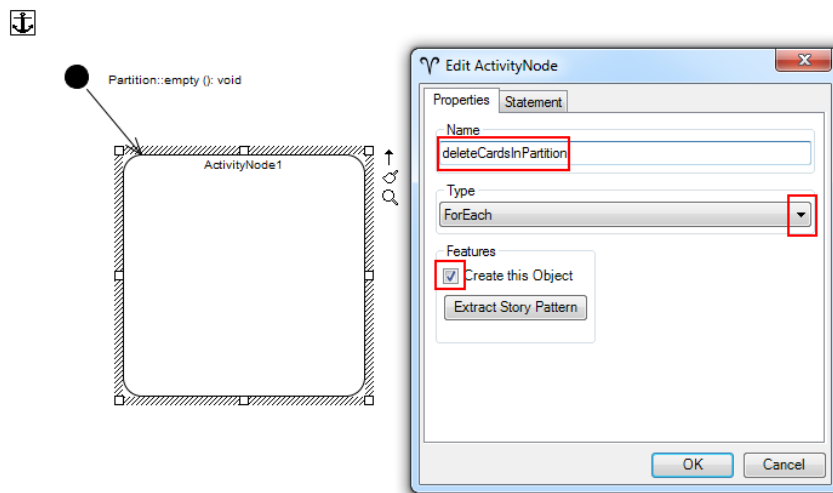
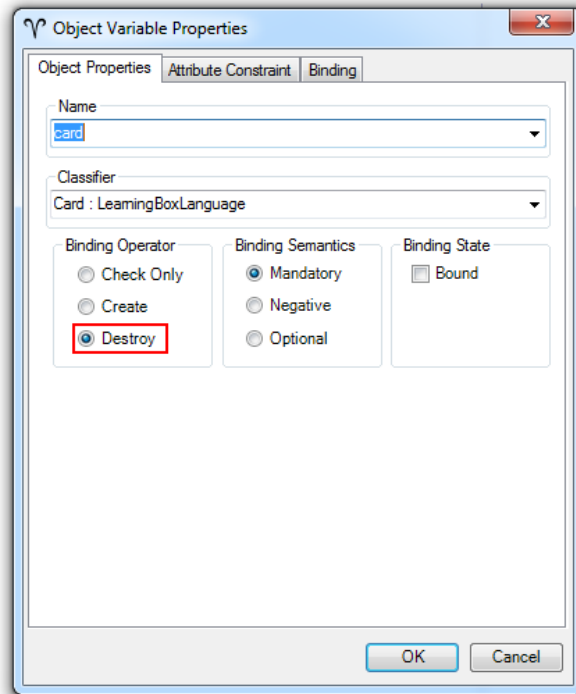
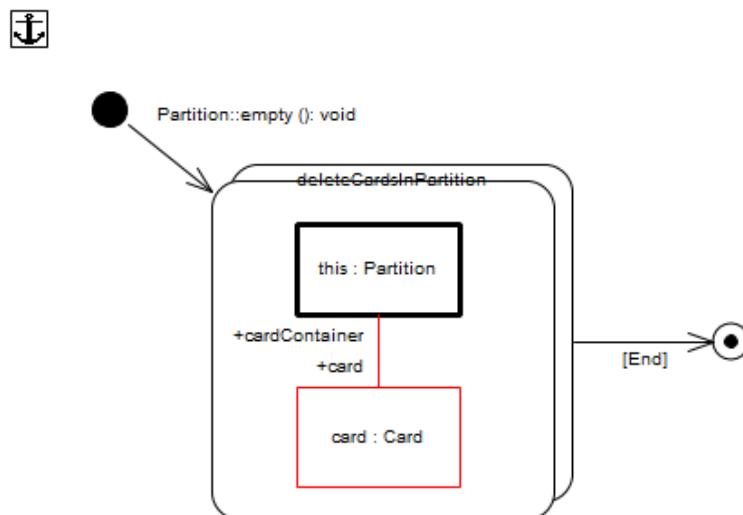


Figure 53: Creating a looping story node

- Now create the `card` object variable needed to complete this SDM. Unlike `removeCard` (Fig. 20) 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. 54).
- Complete the story pattern as indicated in Fig. 55. 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 54: Editing `card` so it gets destroyedFigure 55: 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. 57 and Fig. 58 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. 56).

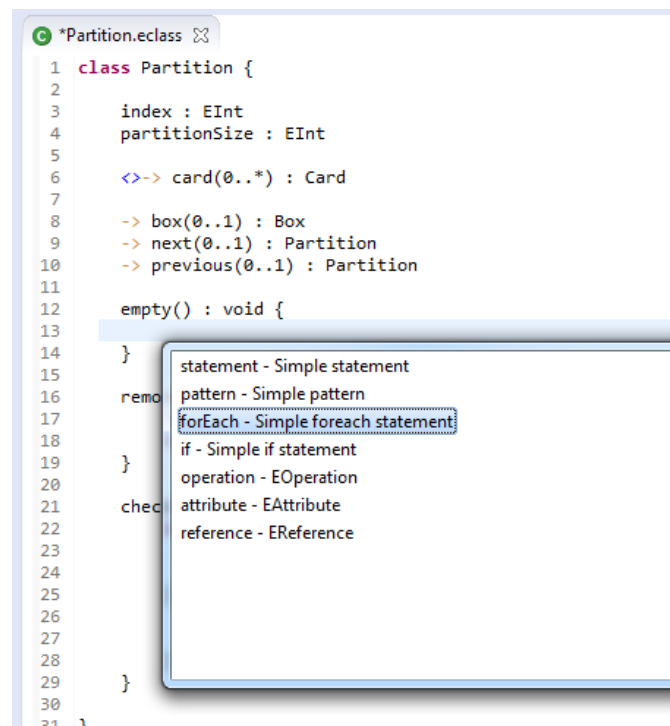


Figure 56: eMoflon's auto completion

- 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. 57.

```

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

```

Figure 57: Control flow for `partition.empty()`

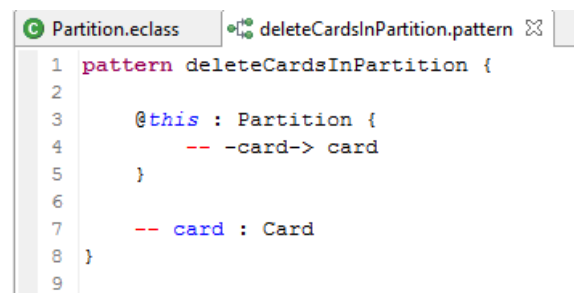
- While similar to `removeCard` (Fig. 25), this new pattern will go one step further by requesting a full destruction of `card`, instead of just deleting the link between the object variables. This means that in addition to destroying the link between the partition and `card`, we need to destroy the 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. 58.



```

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

```

Figure 58: 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. 55 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. 59). 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.

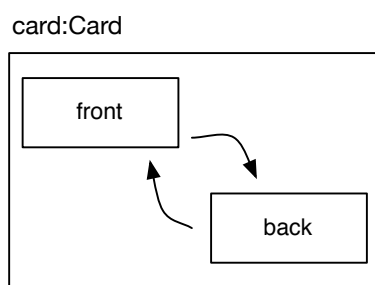


Figure 59: Inverting the attributes of a `Card`

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 constraint¹⁸ 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. *Assignments*

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 metamodeling. If you’re interested in why *unification* is considered cool, check out [1].

¹⁸Which 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. 60.

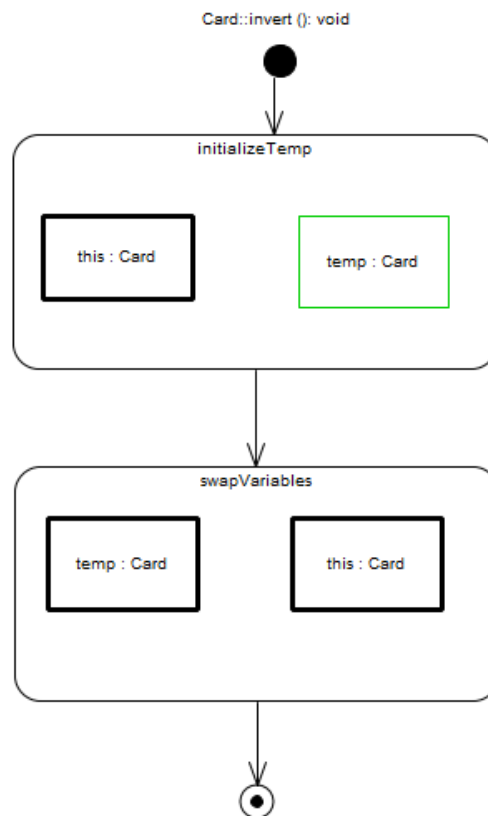


Figure 60: Imperative control layer for inverting a card

- 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. 61).

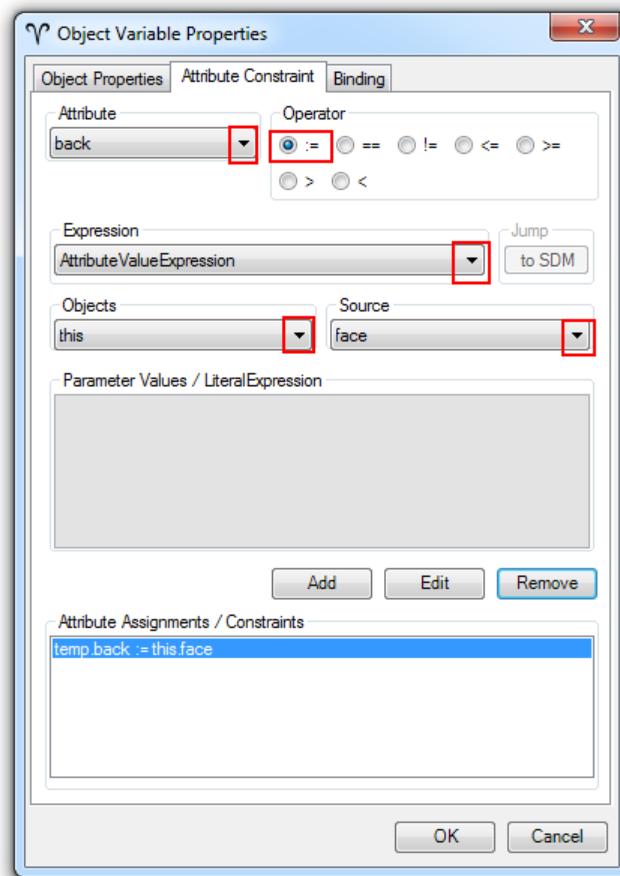


Figure 61: Store the **back** and **face** values of the card in **temp**

- Complete the SDM with the remaining constraints according to Fig. 62 below.

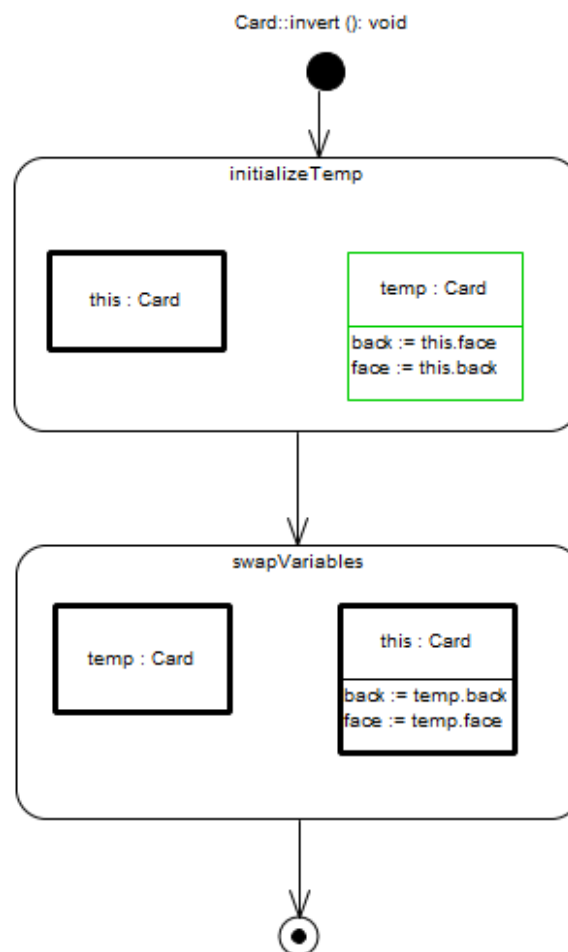


Figure 62: Swap back and face of the card

- Believe or not, that's it! Check out how this method is implemented in the textual syntax by reviewing Fig. 65 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.

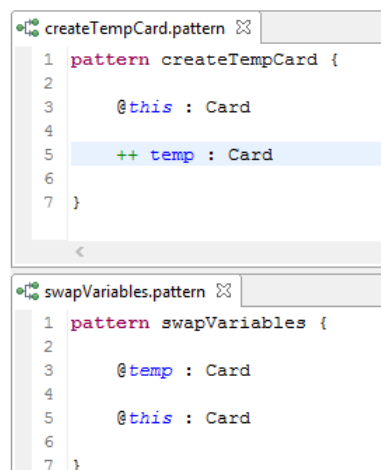
7.2 Implementing invert

- 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. 63).

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

Figure 63: Control flow for `card.invert`

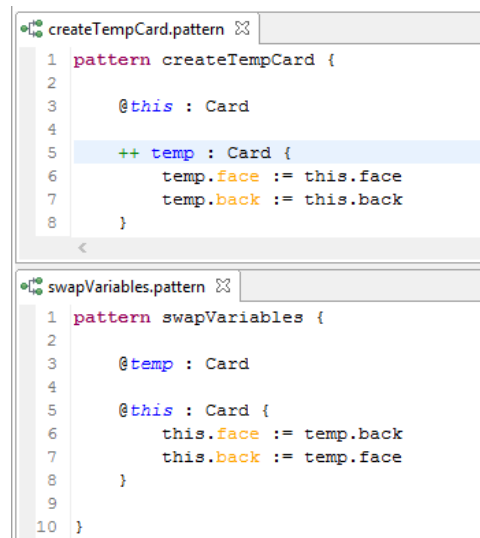
- Create the `this` and `temp` Card variables in each pattern until your workspace resembles Fig. 64.



```
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 64: 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. 65.



```
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 }

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 65: 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. 62 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`?¹⁹ 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!

¹⁹See Fig. 62 (Visual) or Fig. 65 (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). 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. 66).

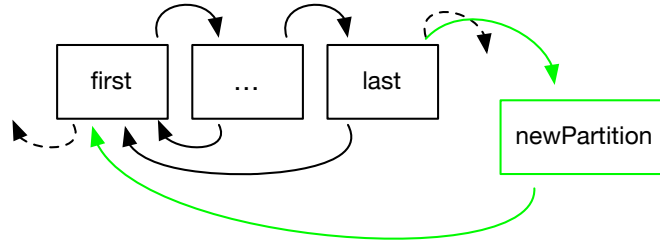


Figure 66: Growing a box by inserting a new partition

SDMs provide a declarative means of identifying specific partitions via *Negative Application Conditions*, simply referred to as NACs.²⁰ 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 [2], NACs can be arbitrarily complex graphs that are much more general and powerful than what we currently support in our implementation,²¹ namely only single negative elements (object or link variables).

As depicted in Fig. 66, 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

²⁰Pronounced \ˈnak\

²¹To 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. 67. 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 67: 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. 68. This matches the box and *any* two partitions.²²

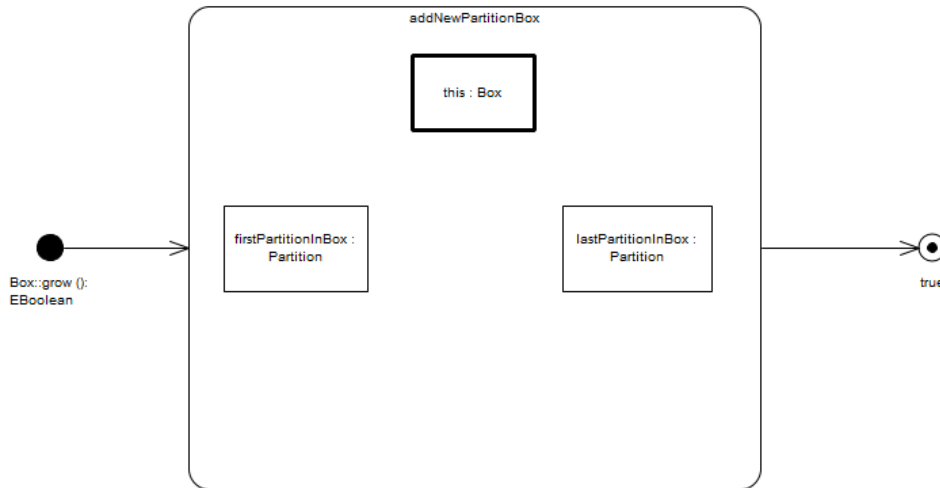


Figure 68: Context elements for SDM

- 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. 69). 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. 70.

²²Remember, the *pattern matcher* is non-deterministic.

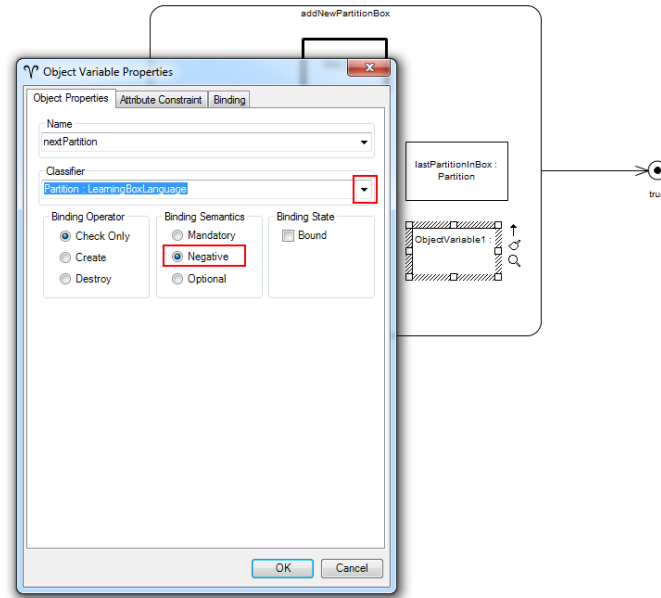


Figure 69: Adding a negative element

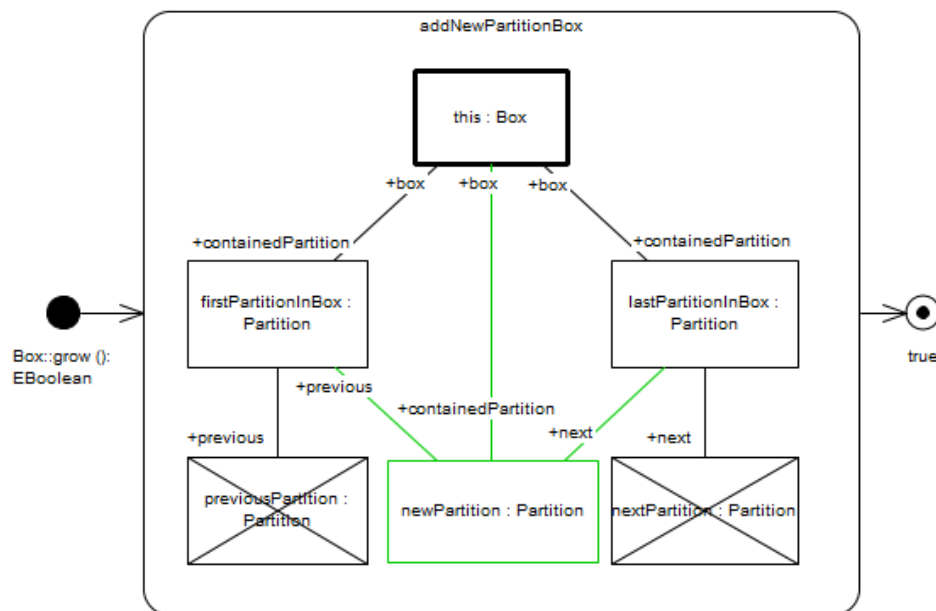


Figure 70: Determining the first and last partitions with NACs

- 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). 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. 71.

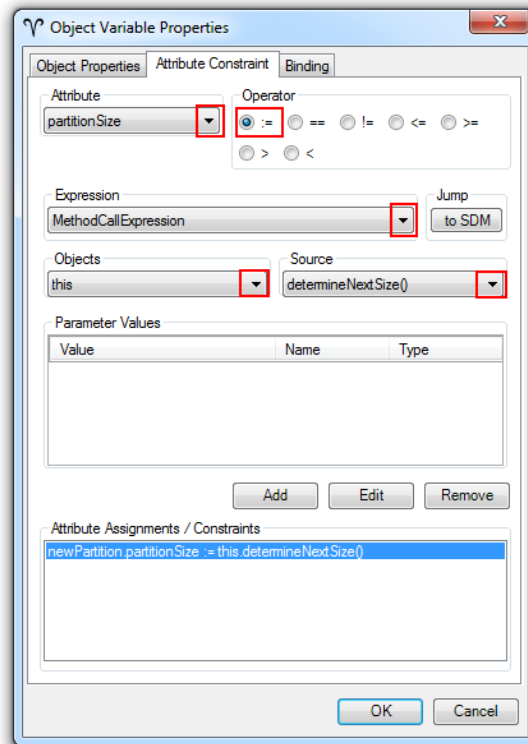


Figure 71: Invoking a method via a `MethodCallExpression`

- 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. 72. 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. 78.

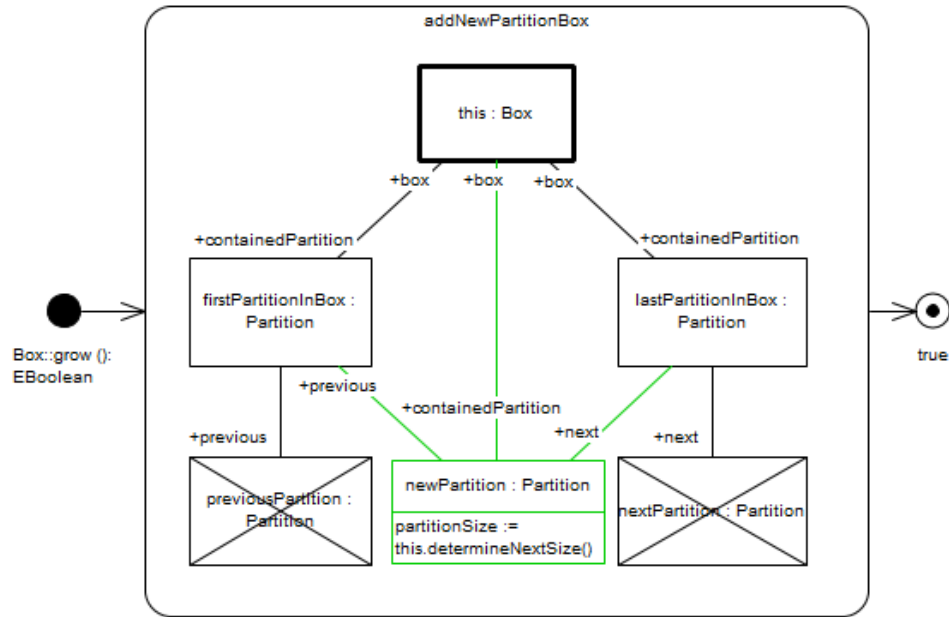


Figure 72: Complete SDM for `Box::grow`

8.2 Implementing grow

- In `box.grow()`, create a simple control flow with one story pattern (Fig. 73).

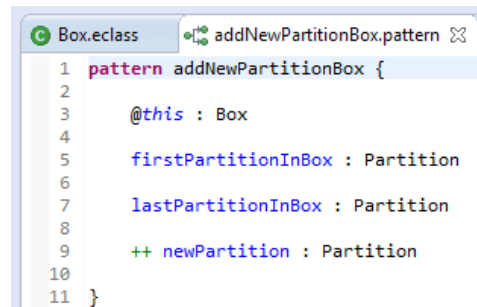
```

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

```

Figure 73: 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. 74).



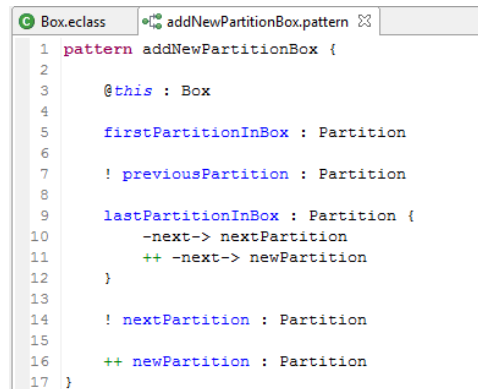
```

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

```

Figure 74: 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. 75). This constraint will only be fulfilled if the NAC fails, and it establishes the `newPartition` as the final partition.



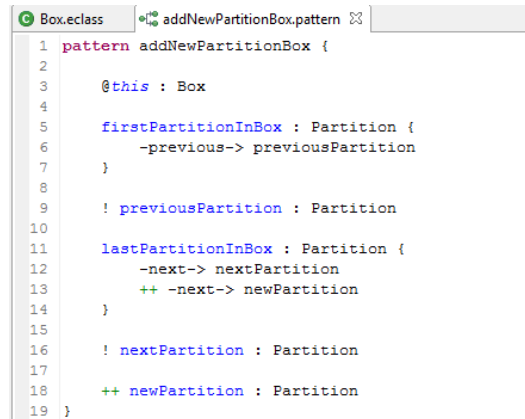
```

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 75: 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. 76).



```

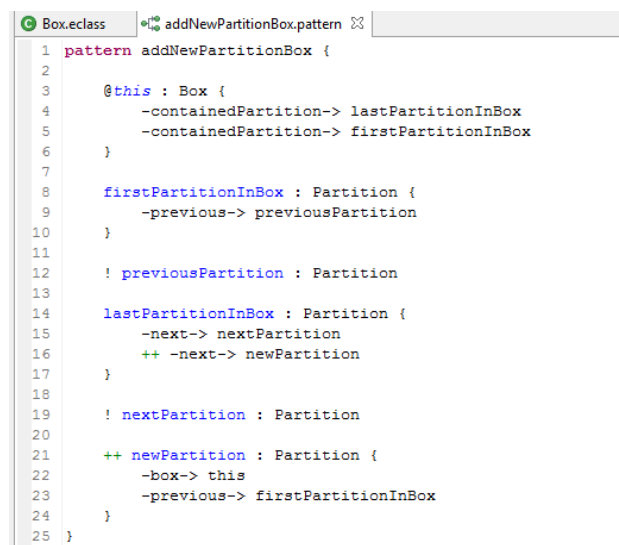
Box.eclass  addNewPartitionBox.pattern
1 pattern addNewPartitionBox {
2
3   @this : Box
4
5   firstPartitionInBox : Partition {
6     -previous-> previousPartition
7   }
8
9   ! previousPartition : Partition
10
11  lastPartitionInBox : Partition {
12    -next-> nextPartition
13    ++ -next-> newPartition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

```

Figure 76: 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.²³ 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 77.



```

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 77: 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

²³Remember 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. 78.



```

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 78: 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. 72 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.

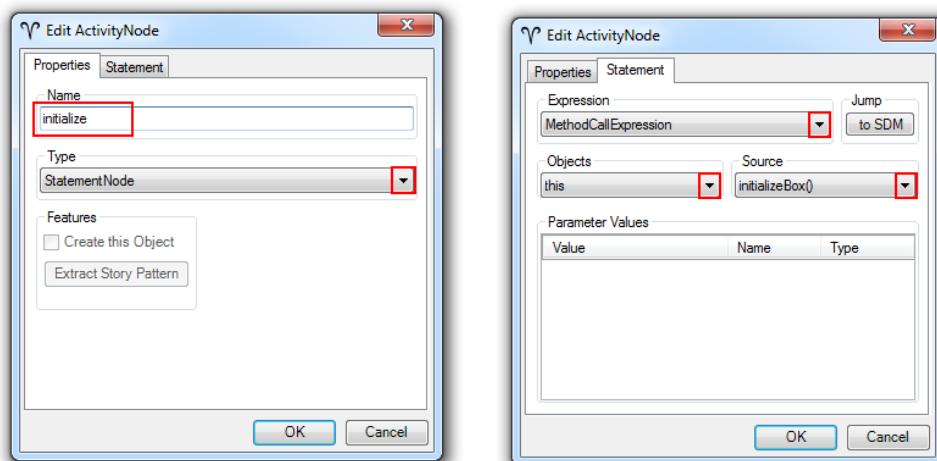
Let’s reconsider `grow`, the method we just completed that adds a new partition to our box. Reviewing either Fig. 72 (Visual) or Fig. 78 (Textual), the current pattern assumes there are already at least two partitions in `box` (the `firstPartitionInBox` and `lastPartitionInBox`). What would happen if `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.

▷ Next [visual]
▷ Next [textual]

9.1 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 F10.²⁴
- 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 activity 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. 79a).
- Before closing the dialogue, switch to the `Statement` tab, and create a `MethodCallExpression` to invoke your newest method (Fig. 79b). We want to access the `Box` object (`this`) and its `inititalizeBox` method. It doesn't require any parameters, so leave the values field empty.



(a) Create a new *StatementNode*

(b) Edit the *MethodCallExpression*

Figure 79

²⁴To 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. 80.

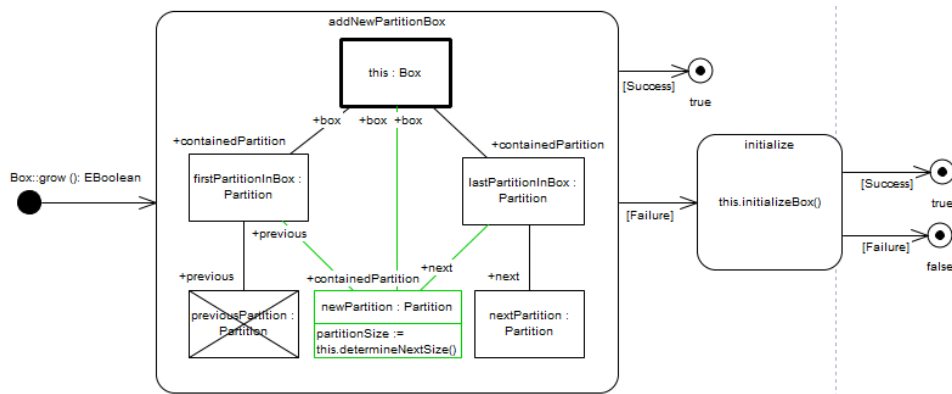


Figure 80: Extending `grow` with a *MethodCallExpression*

- 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. 81.

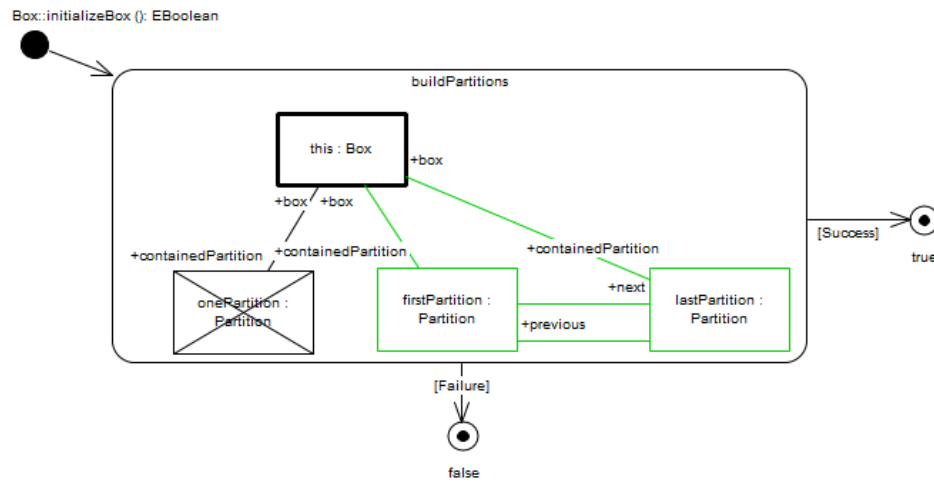


Figure 81: Complete SDM

- 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. 82 and Fig. 83.

9.2 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 `[addNewPartitionBox]` 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. 82.

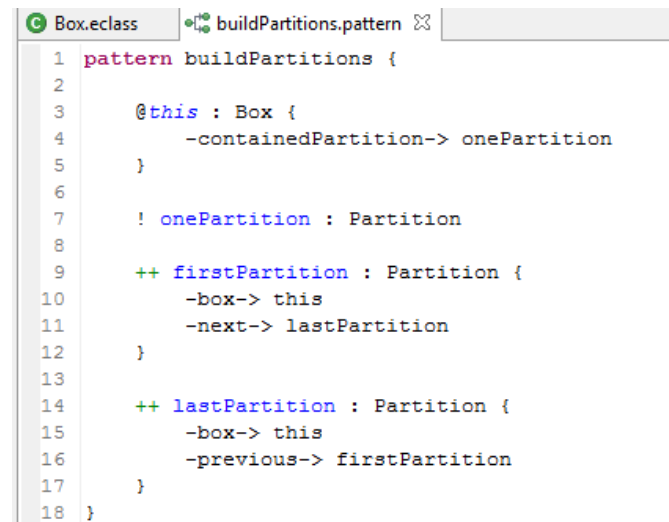
```

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 82: 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. 83.
- 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 image shows a code editor window with two tabs: 'Box.eclass' (active) and 'buildPartitions.pattern'. The code in the active tab is a pattern definition for 'buildPartitions' in the 'Box.eclass' file. 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 83: 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. 80 and Fig. 81.

9.3 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. 84). 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.

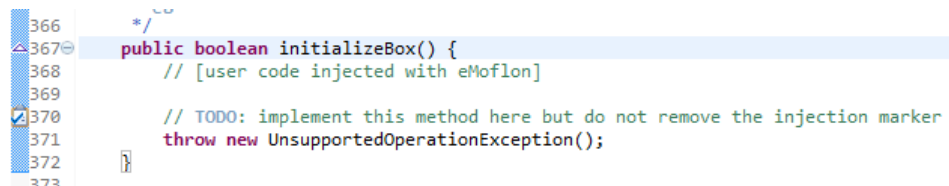
```

    } else {
        // statement node 'initialize'
        fujaba__Success = this.initializeBox();
        if (fujaba__Success) {
            return true;
        } else {
            return false;
        }
    }
}

```

Figure 84: Code generated for branching with a statement node

Hold `Ctrl` while clicking on `initializeBox()` to automatically jump to its declaration. If you didn’t complete the SDM, it would look like Fig. 85.



```

366  */
367  public boolean initializeBox() {
368      // [user code injected with eMoflon]
369
370      // TODO: implement this method here but do not remove the injection marker
371      throw new UnsupportedOperationException();
372  }
373

```

Figure 85: 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.

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. 86). Further still, we'll need to call a helper method to accumulate the contents of each card to a single string.

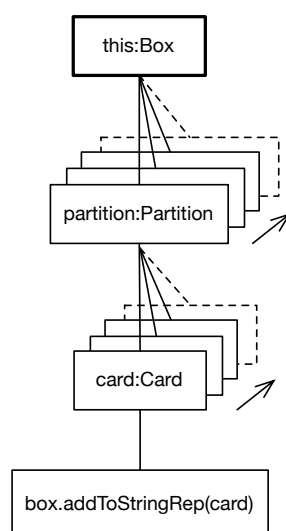


Figure 86: Nested *For Each* loops

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. 87. 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 87: Implementation of addToStringRep

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. 88.

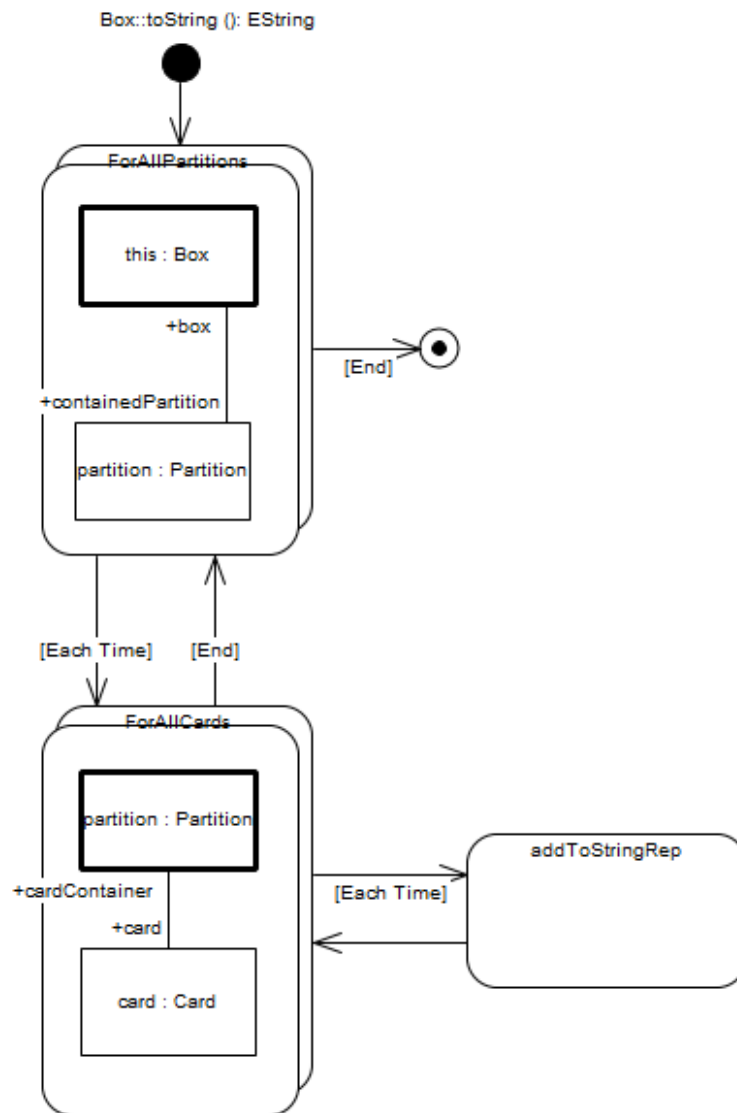


Figure 88: Control flow with nested loops

- 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. 89).
- You can see in the **Source** statement that a `Card` parameter is required. Update the **Parameter Values** field to include `card` so we may pass the object variable to the method. Although you can simply type in `card`, the best practice is to double-click the field to pop-up a new dialogue. This way, complex expressions can be constructed by repeatedly double-clicking the fields.

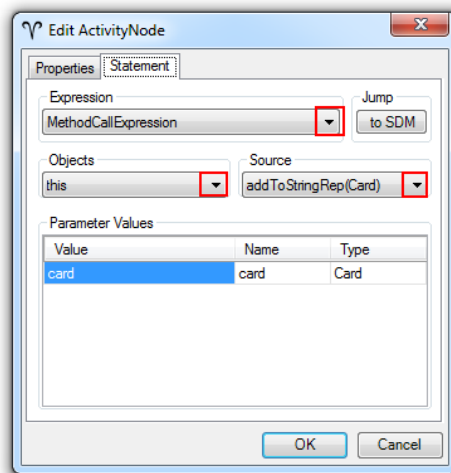


Figure 89: Add a parameter to the *MethodCallExpression*

- To complete the SDM, return the final string representation value of the box via an *AttributeValueExpression* in the stop node (Fig. 90). 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

*AttributeValue-
Expression*

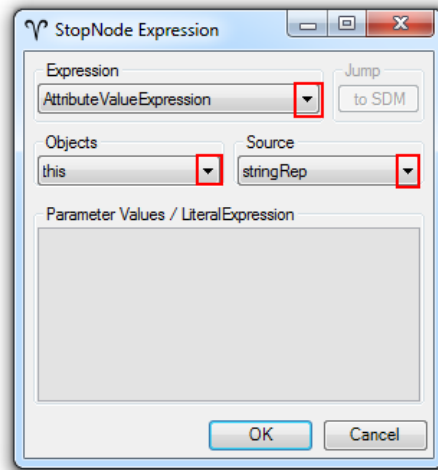
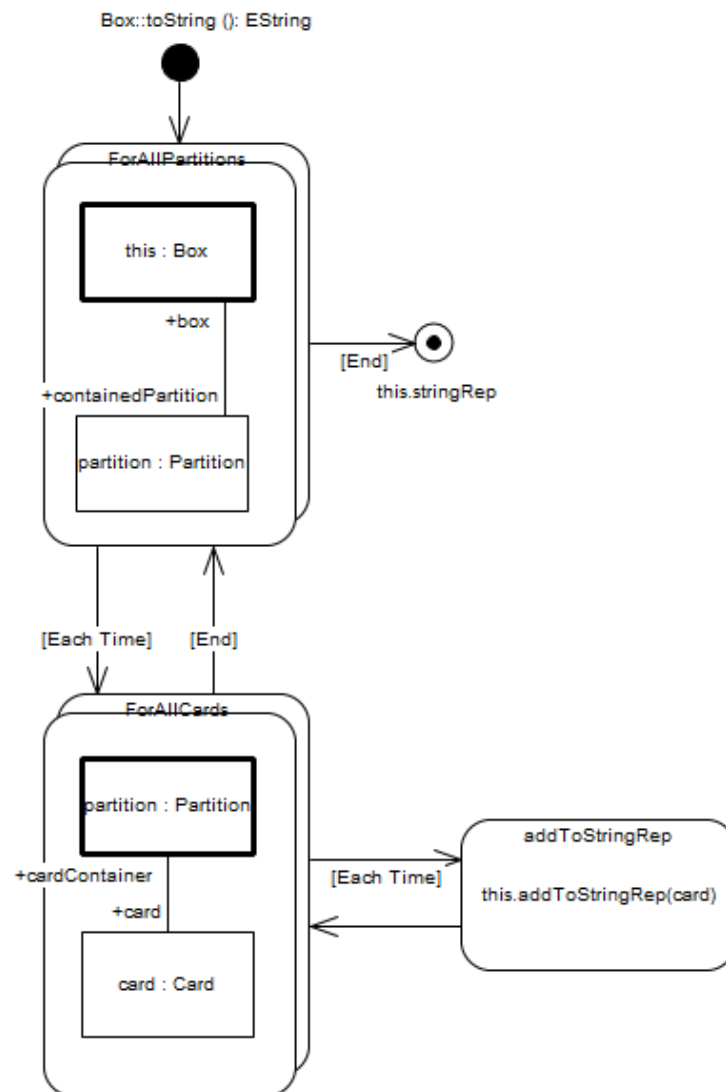


Figure 90: Specify a return value as an *AttributeValueExpression*

- Take some time to compare and reflect on the complete SDM as depicted in Fig. 91. 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. 93 and each pattern in Fig. 94.

Figure 91: The complete SDM for `Box::toString`

10.2 Implementing toString for Box

- Closely following Fig. 86, 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. 92. `this.stringRep` is referred to as an *Attribute Value Expression* 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 92: 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 second 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. 93.

```

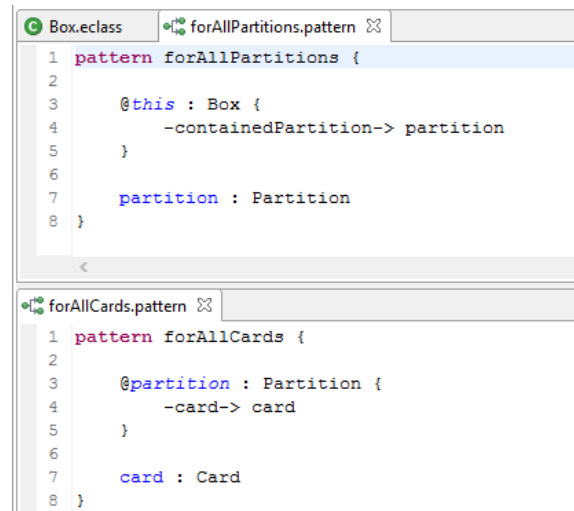
15  toString() : EString {
16      forEach [forAllPartitions] {
17          forEach [forAllCards] {
18              <@this.addToStringRepCard(card)>
19          }
20      }
21      return this.stringRep
22  }

```

Figure 93: 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. 94. 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.



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

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

Figure 94: Box traversal patterns

- As crazy as it may seem, that's it! To see how this SDM is represented visually, check out Fig. 91.

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. 95).

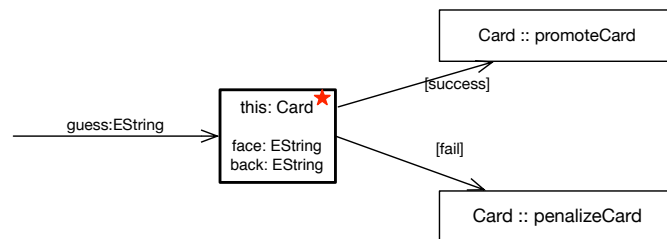


Figure 95: Checking a fast card against a guess

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 type²⁵ 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 specified for a *bound* object variable and is the final case in which an object variable can be marked as being bound. *Binding*

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**.

²⁵In 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 meta-model 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. 96). Your metamodel should then resemble Fig. 97.

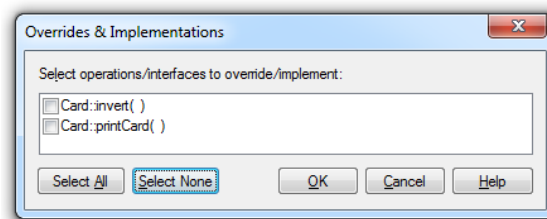


Figure 96: Selecting operations to override

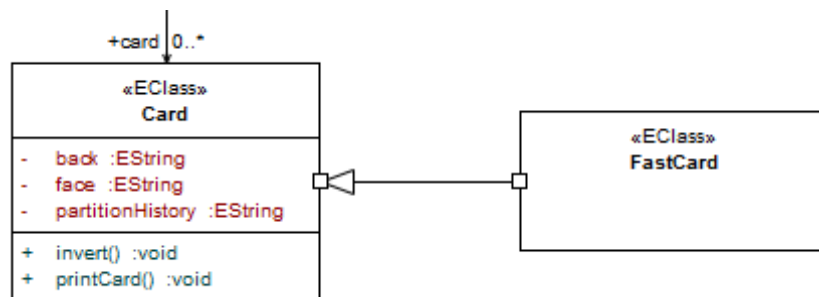


Figure 97: Fast cards are a special kind of card

- Now return to the **check SDM** (in **Partition**) and extend the control flow as depicted in Fig. 98.
- 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. 99).

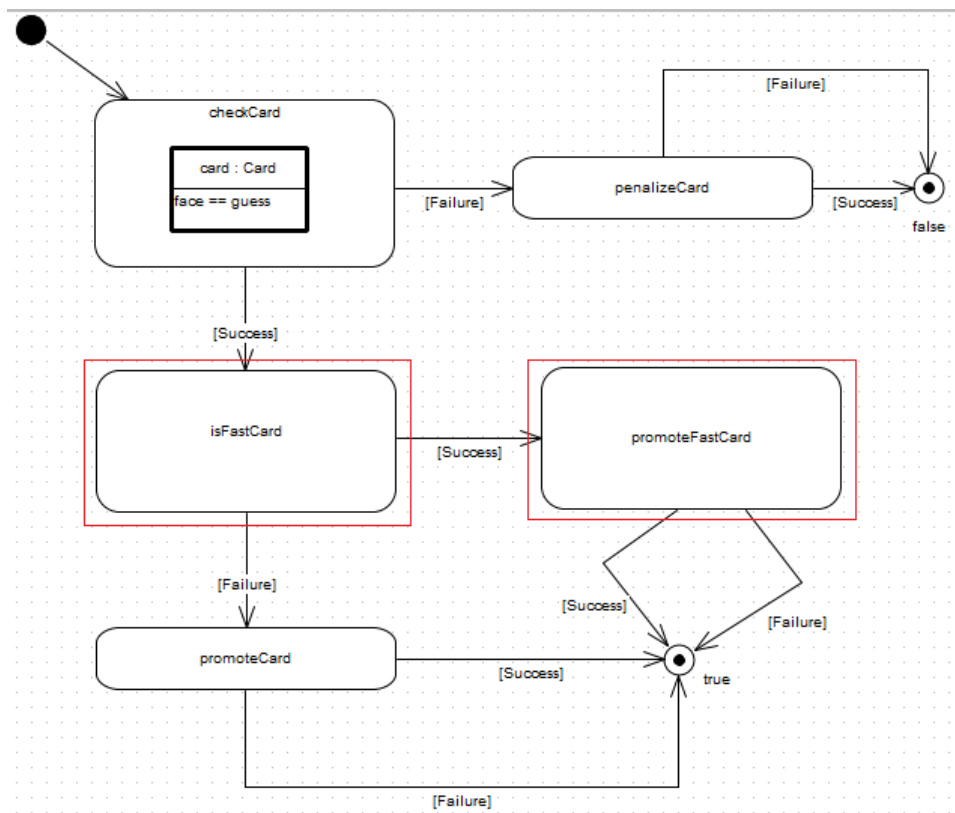


Figure 98: Extend check to handle fast cards.

- 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. 99). 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.

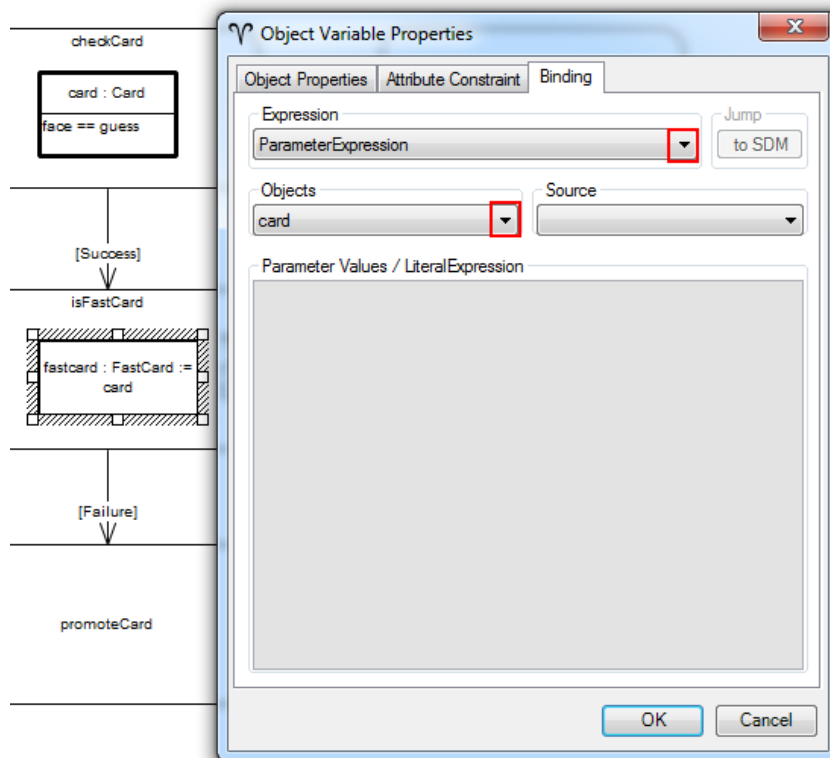


Figure 99: Create a binding for `fastcard`

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. 100 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. 100 to Figs. 40 and 41, 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.

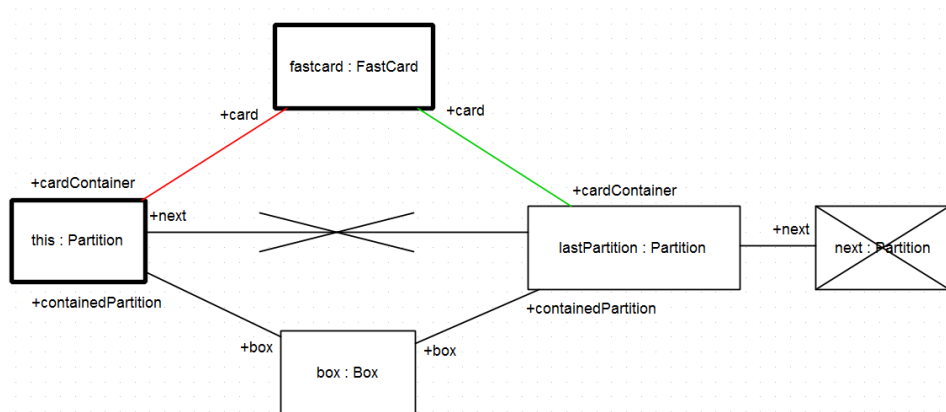


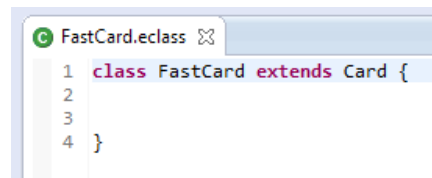
Figure 100: Story pattern for handling fast cards.

- Inspect Fig. 105 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. 101.



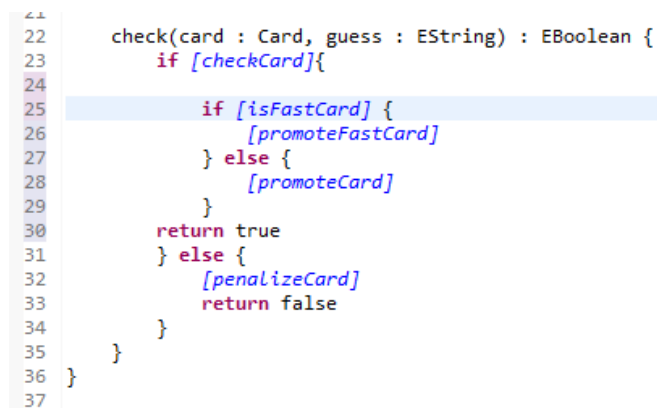
```

1 class FastCard extends Card {
2
3
4 }

```

Figure 101: 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. 102.



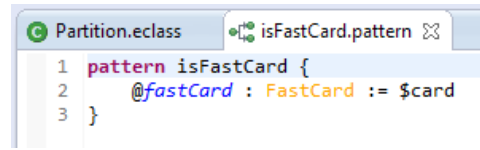
```

22 check(card : Card, guess : EString) : EBoolean {
23     if [checkCard]{
24
25         if [isFastCard] {
26             [promoteFastCard]
27         } else {
28             [promoteCard]
29         }
30     }
31     return true
32 } else {
33     [penalizeCard]
34     return false
35 }
36 }
37

```

Figure 102: 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. 103.



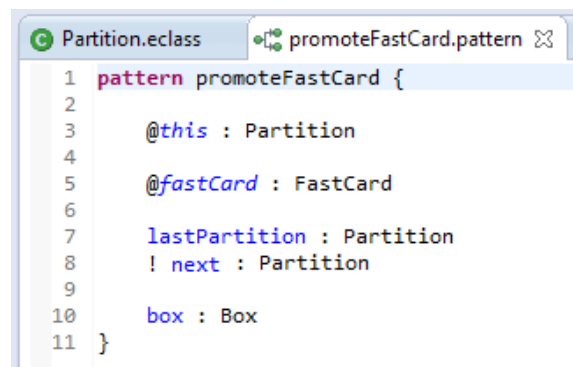
```

1 pattern isFastCard {
2   @fastCard : FastCard := $card
3 }

```

Figure 103: 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. 104.



```

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

```

Figure 104: 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.²⁶
- Your final pattern should resemble Fig. 105. As you can see, this pattern is remarkably similar to the original movement patterns, `promoteCard` and `penalizeCard` (Fig. 49). This of course makes sense – the target pattern is only now always the last partition in the box and no longer the current `next`.

²⁶If 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 105: 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. 100 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. 106). As you can see, it has all the same attributes as a standard **card**.

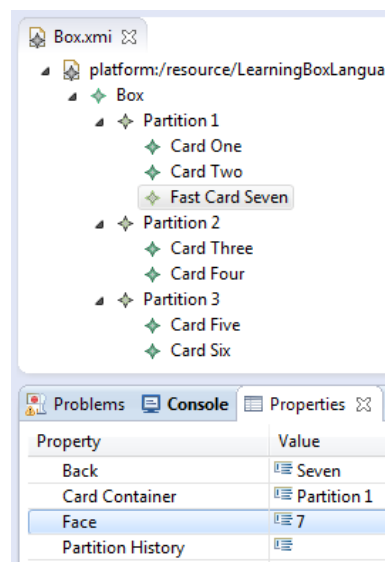


Figure 106: Creating and editing a new **FastCard** element

- 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 DSL²⁷ 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.

²⁷A 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 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. 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.

References

- [1] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 2005.
- [2] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Berlin, 2006.