An Introduction to Metamodelling and Graph Transformations

with eMoflon

Part III: Story Driven Modelling

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

The eMoflon team
Darmstadt, Germany (October 2015)

Contents

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.

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.

<div align="center">▷ Continue from Part II...</div>

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. 1.1, a box consists of a series of partitions with a strict set of rules. The contents to be memorized are written on little cards and placed in the first container. Every time the user correctly answers a card, that card is promoted to the next partition. Once it reaches the final partition, it can be considered memorized, and no longer needs to be practiced. Every time the user incorrectly answers a card however, it is returned to the original starting partition, and the learning process is restarted.
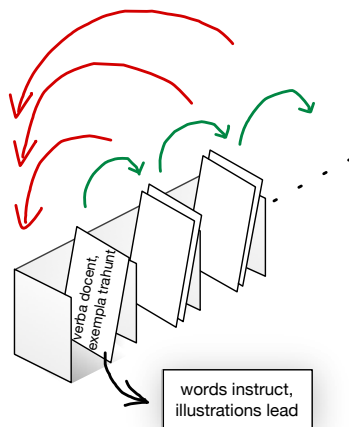
Figure 1.1: Static Structure of a Leitner's Learning Box

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

▶ To get started in Eclipse, press the `Install, configure and deploy Moflon` button and navigate to "Install Workspace". Choose "Handbook Example (Part 3)". (Fig. 1.2).

▶ 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.

---

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

Figure 1.2: Choose a cheat package

▶ In order to start working with the cheat package, you have to generate code by (i) opening the `.eap` file in `LeitnersLearningBoxVisual`, (ii) exporting it using Enterprise Architect, (iii) refreshing 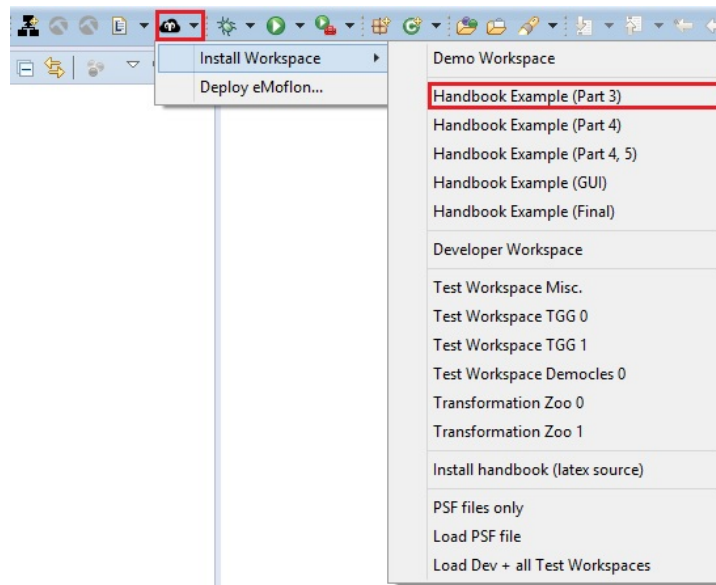the project containing the `.eap` and (iv) rebuilding the `LearningBoxLanguage` project. For more details on the code generation process, refer to Part I, Section 2.

▶ 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.

---

[2]To learn how to make your own instance models, review Part II, Section 4

2 Transformations explained

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

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

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

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

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

1. *Finding* an occurrence of the precondition L in G via a *match*, m

2. *Cutting* out or *Destroying* $(L \setminus R)$, i.e., the elements that are present in the precondition but not in the postcondition are deleted from G to form $(G \setminus Destroy)$

3. *Pasting* or *Creating* $(R \setminus L)$, i.e., new elements that are present in the postcondition but not in the precondition and are to be created in the hole left in $(G \setminus Destroy)$ to form a new graph, $H = (G \setminus Destroy) \cup Create$ (Fig.2.1).

$$L \xleftarrow{\hspace{2cm}} L \cap R \xrightarrow{\hspace{2cm}} R$$

$$\textcircled{1}m \Big\downarrow \quad PC \qquad \Big\downarrow \quad PO \qquad \Big\downarrow m^*$$

$$G \xleftarrow{\textcircled{2}} G \setminus \underbrace{(L \setminus R)}_{Destroy} \xrightarrow{\textcircled{3}} H$$
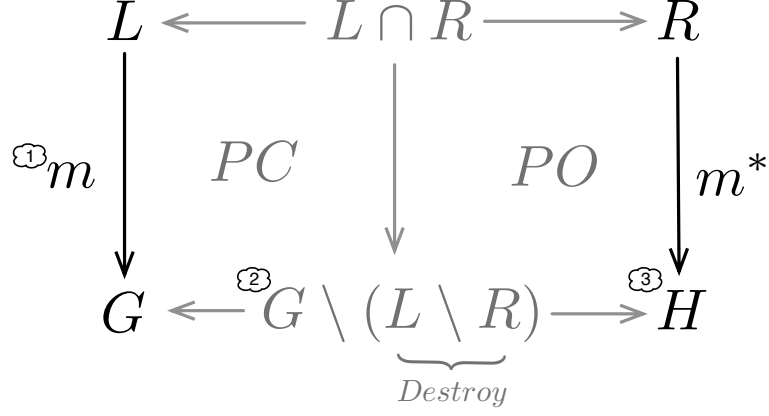
Figure 2.1: Applying a rule $r : (L, R)$ to G to yield H

Let's review this application.

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

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

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

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 $Destroy := (L \setminus R)$ from G to yield $(G \setminus Destroy)$ (deletion), and one when inserting $Create := (R \setminus L)$ in $(G \setminus Destroy)$ 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 2.2 depicts this *moveCard* rule.

As already indicated by the colours used for *moveCard*, we employ a compact representation of rules formed by merging (L, R) into a single *story*
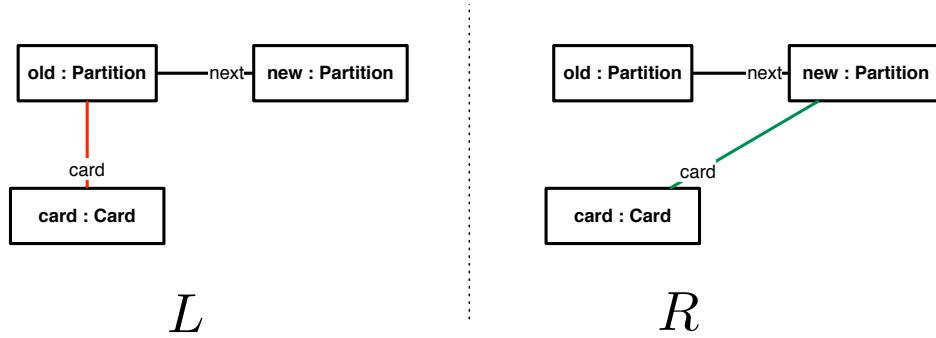
Figure 2.2: *moveCard* as a graph transformation rule

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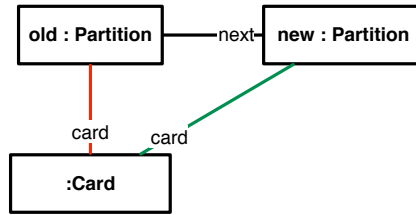

Figure 2.3: Compact representation of *moveCard* as a single *story pattern*

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

Despite being able to merge rules together to form one story pattern, the individual rules still have to be applied in a suitable sequence to realise complex model transformations consisting of many steps! This can be specified with simplified activity diagrams, where every *activity node* or *story node* *Activity* contains a single *story pattern*, and are combined with the usual imperative *Node* constructs to form a control flow structure. The entire transformation can therefore be viewed as two separate layers: an imperative layer to define the top-level control flow via activities (i.e., if/else statements, loops, etc.), and a pattern layer where each story pattern specifies (via a graph transformation rule) how the model is to be manipulated.
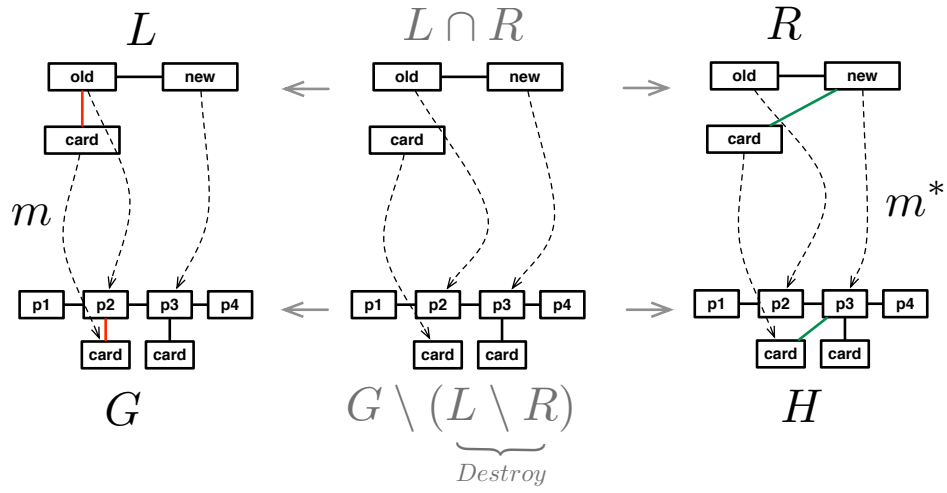
Figure 2.4: Applying *moveCard* to a learning box

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

3 Removing a card

Since we're just getting started with SDMs,[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. 3.1).
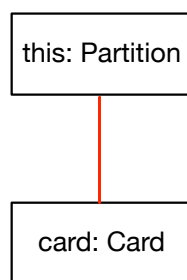

Figure 3.1: Removing a card from its partition

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

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

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

- ▶ Delete the `PartitionImpl.inject` file from your working set (Fig 3.2).

- ▶ Now select `LearningBoxLanguage` and click on the "Build" button.

- ▶ 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.

---

[3]As you may have already noticed, we use "SDM" or "Story Diagrams" interchangeably to mean both our graph transformation language *or* a concrete transformation used to implement a method, consisting of an activity with activity nodes containing story patterns.
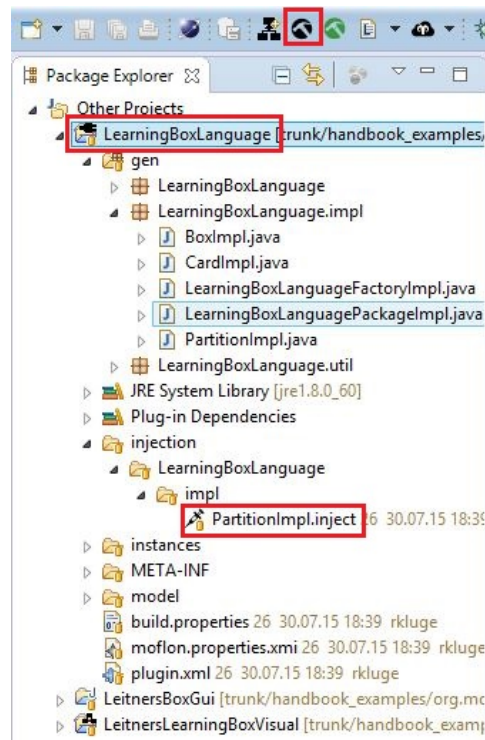
[4]Refer to Part II, Section 6

Figure 3.2: Remove injection content

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

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

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

Object Variable

---

[5]We shall learn what further conditions may be specified in later SDMs.

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

Patterns also declare *link variables* to match references in the model. Given 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. 3.3), and (3) *Double-click* on the chosen method to indicate that you want to implement it.
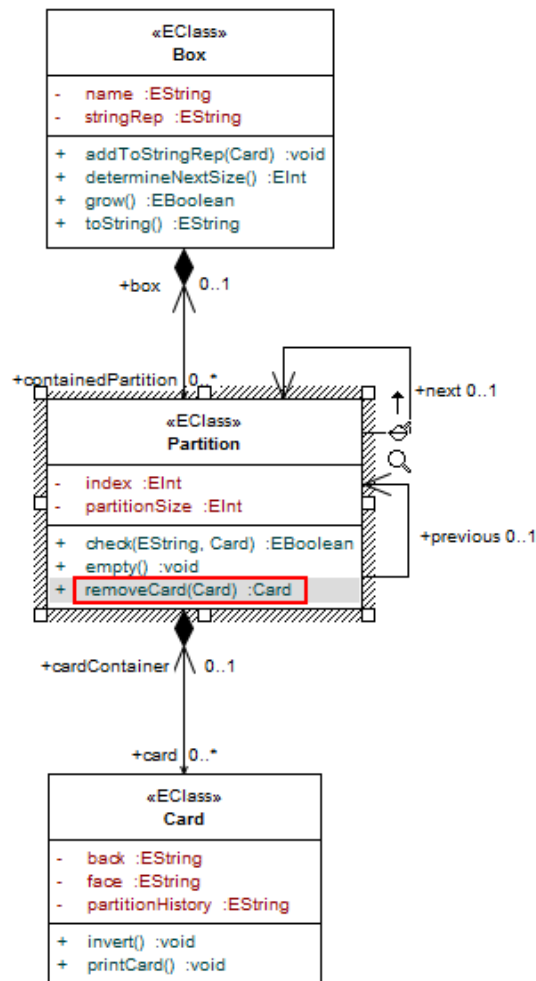

Figure 3.3: Double-click a method to implement it

▶ If you did everything right (and answered the question which popped up about creating a new SDM diagram with Yes), a new *activity diagram* should be created and opened in a new tab with a cute anchor in the corner, and a *start node* labelled with the signature of the method
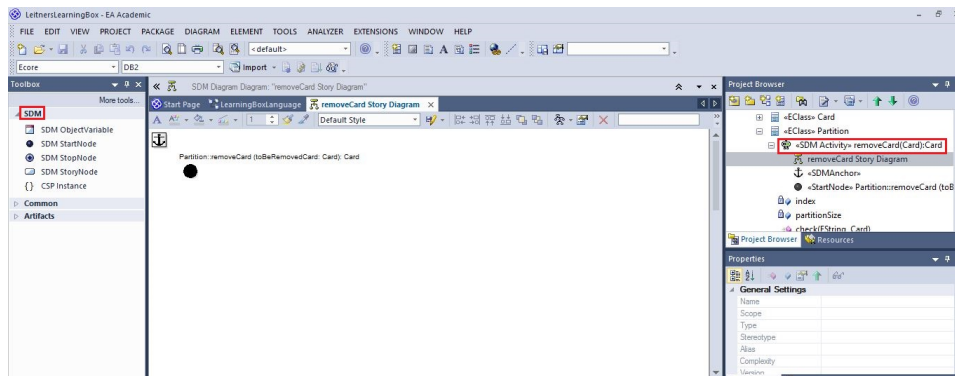
(Fig. 3.4).

Figure 3.4: Generated SDM diagram and start node

▶ This diagram is where you'll model `removeCard`'s *control flow*. In other words, this is `removeCard`'s imperative top-level diagram. We refer to the whole activity diagram simply as the *activity*, which always starts with a *start node,* contains *activity nodes* connected via *activity edges*, then finally terminates with a *stop node.* Before creating these however, let's quickly familiarise ourselves with the EA workspace.

▶ First, inspect the project browser and notice that an `<<SDM Activity>>` container has been created for the method `removeCard`. This container will eventually host every artifact related to this pattern (i.e., object variables, stop nodes, etc.). Please note that if you're ever unhappy with an SDM, you can always delete the appropriate container in the project browser (such as this one), and start from scratch.

▶ Next, note the new `SDM` toolbox that has been automatically opened for the diagram and placed to the left above the common toolbox. This provides quick access to SDM items that you'll frequently use in your diagram. You can also invoke the active toolbox in a pop-up context menu anywhere in the diagram by pressing the `space` bar.

▶ Finally, in the top left corner of the diagram, you'll notice a small anchor. Double click on this icon to quickly jump back to the meta-model. From there, double click the method again to jump back to the SDM. This is just a small trick to help you quickly navigate between diagrams.

▶ To begin, select the start node, and note the small black arrow that appears (Fig. 3.5).
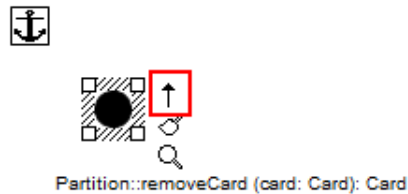
Partition::removeCard (card: Card): Card

Figure 3.5: 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. 3.6, choose `Append StoryNode` to create a *Story Node*.

Partition::removeCard (card: Card): Card
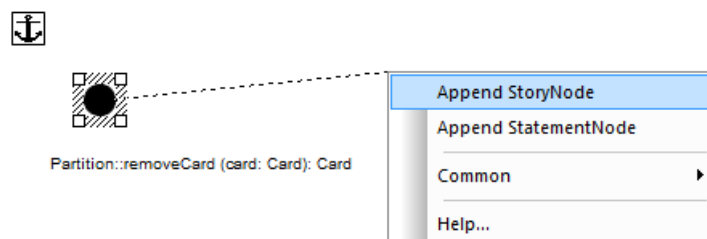
Append StoryNode

Append StatementNode

Common ▶

Help...

Figure 3.6: Create new activity node

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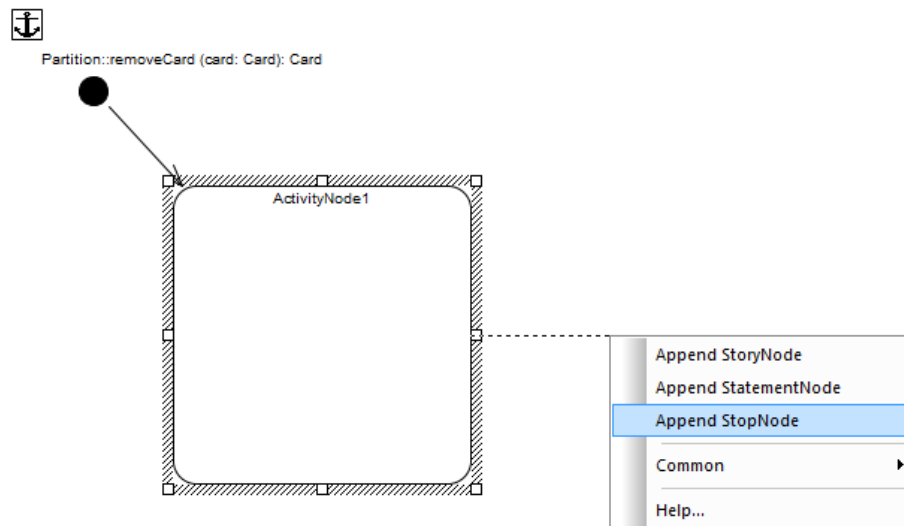
---

[6]This was discussed in Part II, Section 2.5

Figure 3.7: Complete the activity with a stop node

▶ If everything is correct, you should now have a fully constructed activity that models the method's control flow.

▶ While a *stop node* is rather self explanatory, you may be wondering about the differences between the other two menu options, the *story node* and *statement node*. Since not all activity nodes can contain *Story Node* story patterns (e.g., start and stop nodes), those that *can* are called *Statement* story nodes. Statement nodes cannot and are used instead to invoke *Node* 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. 3.8. Enter `removeCardFromPartition` as the name of the story node, and select `Create this Object`. Click `OK`. The activity node now has a single *bound object variable*, `this`.

▶ To create a new object variable, choose `SDM ObjectVariable` from the toolbox then click inside the activity node (Fig. 3.9). A properties window will automatically appear (Fig. 3.10).
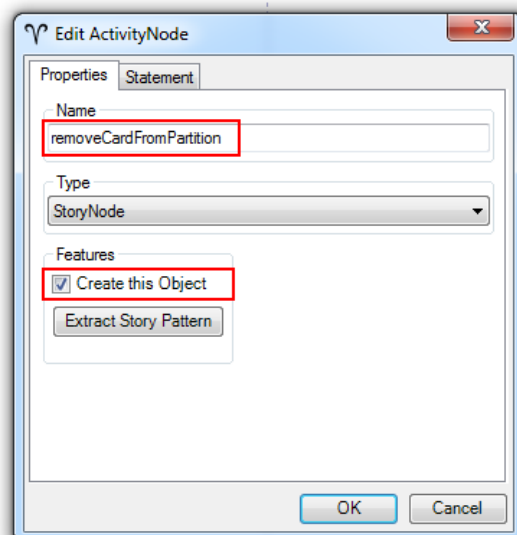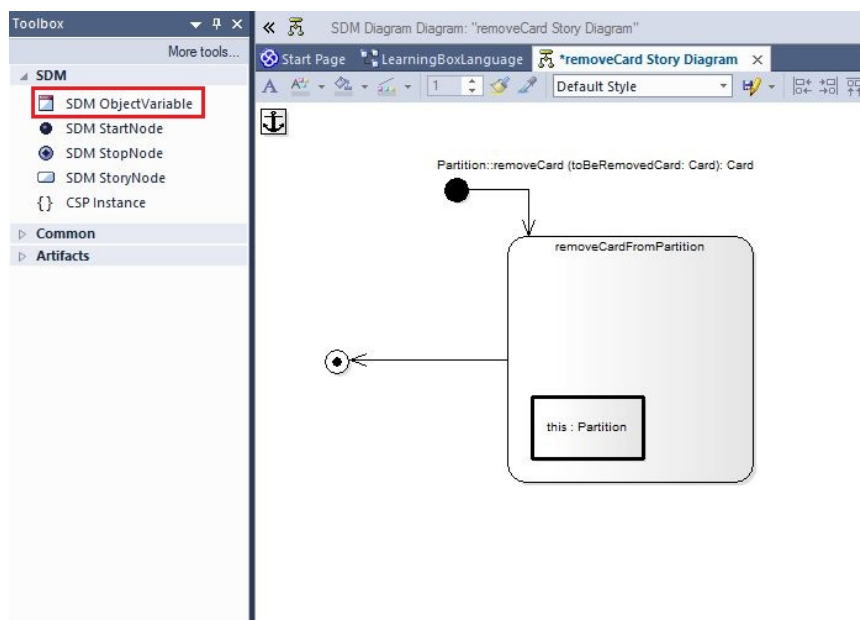
Figure 3.8: Initializing a story node

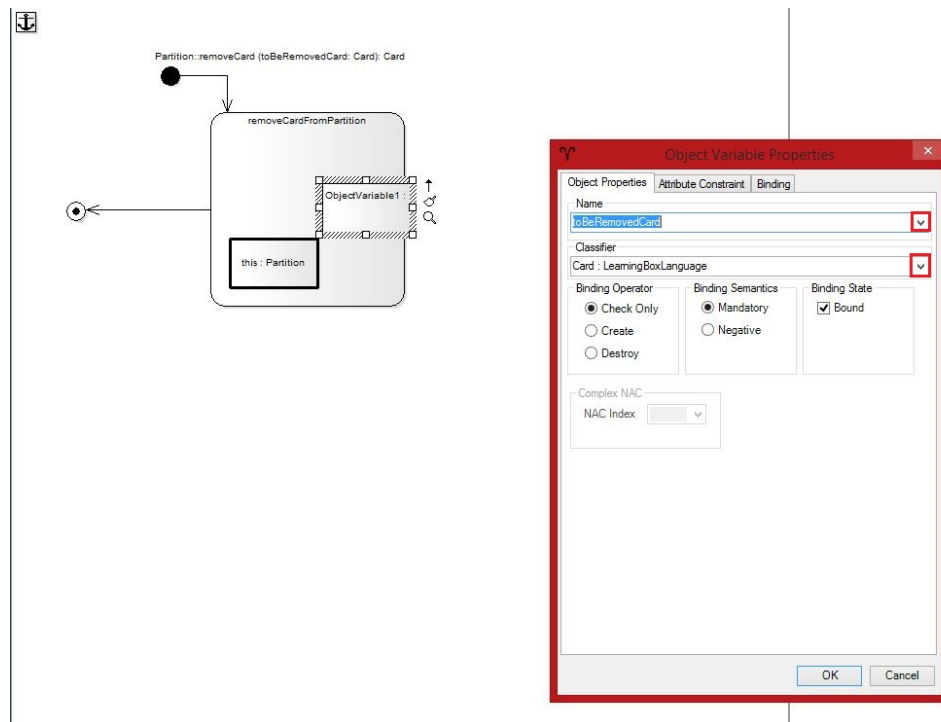Figure 3.9: Add a new object variable from the toolbox

Figure 3.10: Specify properties of the added object variable

▶ Using the drop-down menus, choose `toBeRemovedCard` as the name of the object, and set `Card` as its type. Since `card` is a parameter of the method, it is offered as a possible name which can be directly chosen to avoid annoying typing mistakes.

▶ In this dialogue, note that the `Bound` option is automatically set. We have now seen two cases in this activity for bound object variables: an assignment to `this`, and an assignment to a method parameter. Setting `toBeRemovedCard` to bound means that it will be implicitly assigned to the parameter with the same name.

▶ To create a *link variable* between the current partition and the card to be removed, choose the object variable `this` and quick link it to `toBeRemovedCard` (Fig. 3.11).

▶ According to the metamodel, there is only one possible link between a partition and card. Select this and set the `Binding Operator` to `Destroy` (Fig. 3.12). The reference names will automatically appear in the diagram.
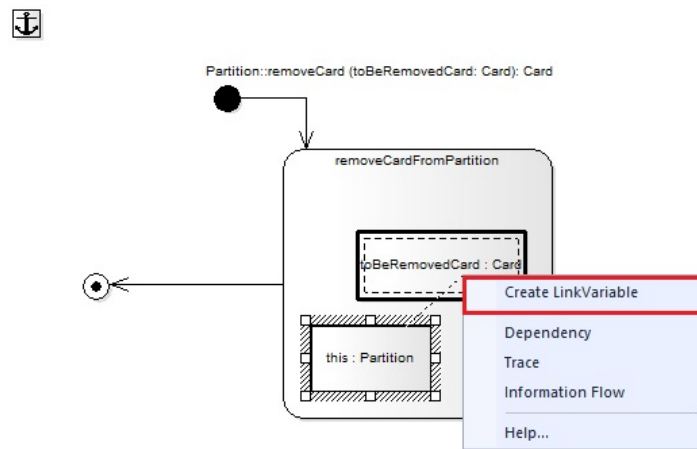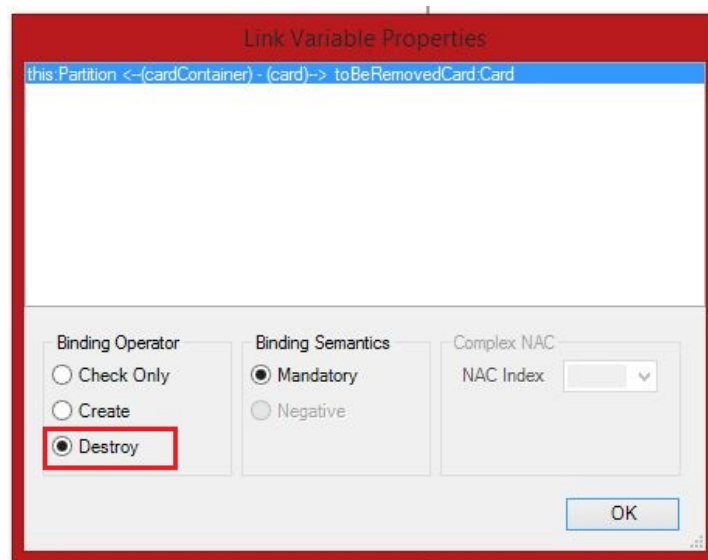
⚓

Figure 3.11: Create a link variable

Figure 3.12: Specify properties for created link variable

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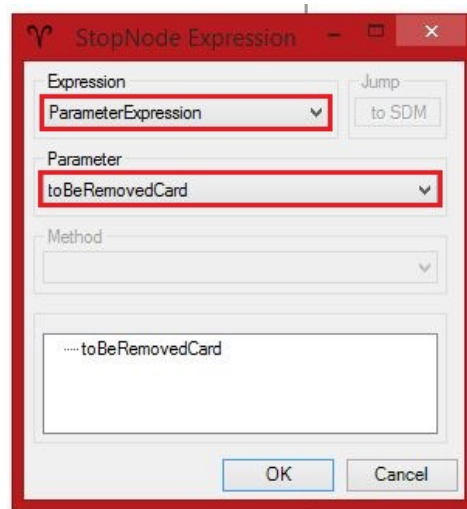
Figure 3.13: Adding a return value to the stop node

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

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

▶ Returning to the activity, if you've done everything right, your first SDM should resemble Fig. 3.14, where `removeCard`'s entire pattern layer is modeled inside the sole *activity node*. The method's return value is now indicated below the stop node.

▶ Don't forget to save your files, validate and export your pattern to the Eclipse workspace,[7] then build your metamodel's code from the package explorer.

---

[7]Go to "`Extensions`" and select `Add-In Windows` to activate eMoflon's console. If you're unsure how to validate, export, or use this window, review Part I, Section 2.1.
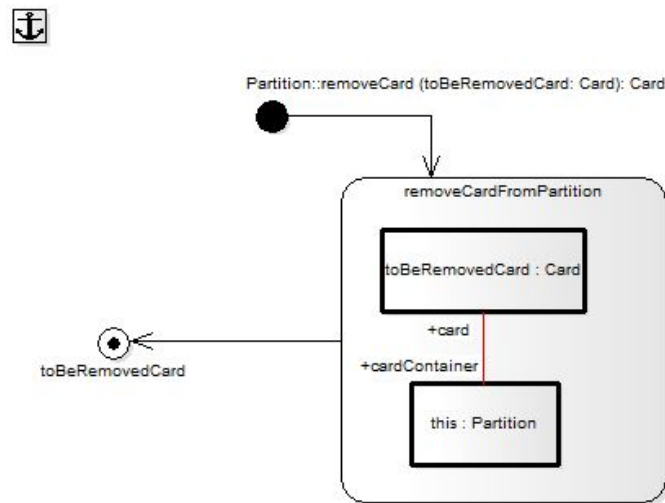
Figure 3.14: Complete SDM for `Partition::removeCard`

▶ If you're unable to export or generate code successfully, compare your SDM carefully with Fig. 3.14 and make sure you haven't forgotten anything.

Removing SDMs

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

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

▶ In the project browser, navigate to the method of which you want to remove the SDM. This can be accomplished by right-clicking the method in the Ecore diagram and then selecting "Find in project browser" (Fig. 3.15), or by navigating manually through the project browser.

▶ Inside of the EClass "Partition", you find the one entry for the method ("removeCard(Card)") and one entry for its SDM ("≪SDM Activity≫ removeCard(Card)") (Fig. 3.16).

▶ Select and remove the SDM Activity to get rid of the SDM associated with "removeCard(Card)" either by using the context menu entry
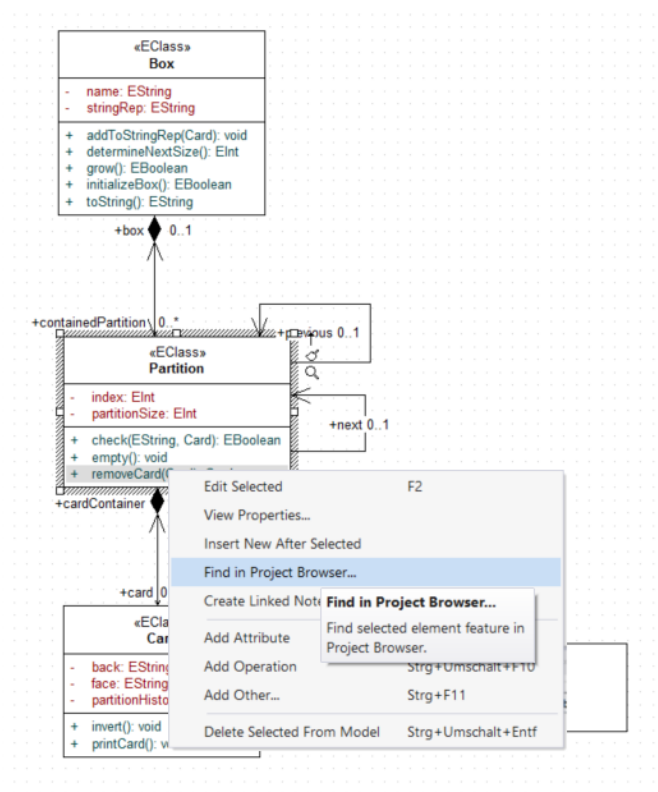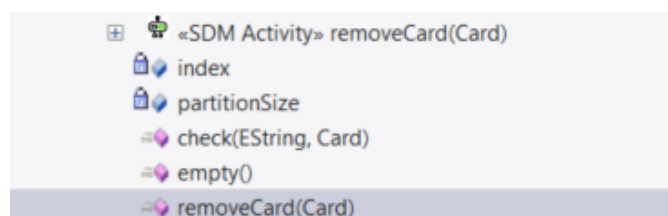
Figure 3.15: Navigate to method in project browser

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

"Delete '≪SDM Activity≫ removeCard(Card)'" or via *Ctrl+Delete.*
Important: This action cannot be undone!

▶ After a successful export and rebuild of your project, you may add an injection for "Partition::removeCard".

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.remove-Card(c)` is invoked for a partition `p`, with a card `c` as its argument, the specified pattern will *match* only if that card is contained in the partition. After determining matches for all variables, the link between the partition and the card is deleted, effectively "removing" the card from the partition. If the card is *not* contained in the partition, the pattern won't match, and nothing will happen. In both cases, the card that's passed in is returned.

▶ If your code generation was successful, navigate to "LearningBox-Language/gen/LearningBoxLanguage/impl/PartitionImpl.java" to the `removeCard` declaration (approximately line 347). Inspect the generated implementation for your method (Fig. 3.17). Notice the null check that is automatically created - only a very conscientious (and probably slightly paranoid) programmer would program so defensively!
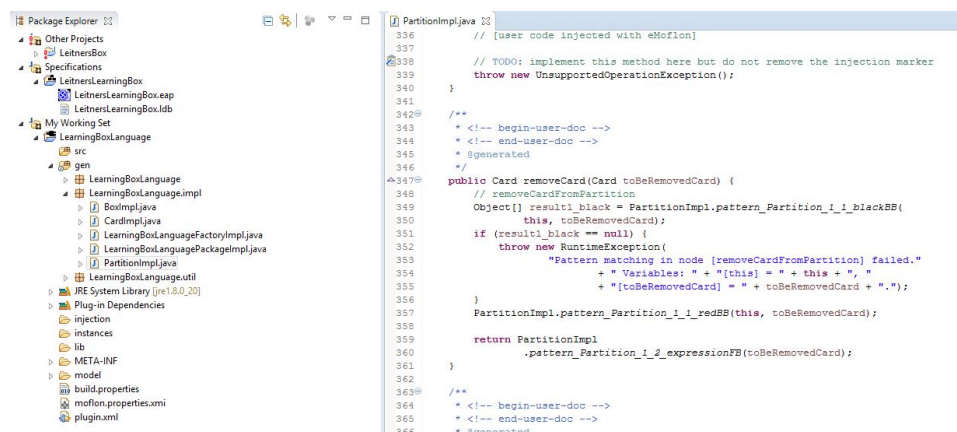
Figure 3.17: Generated implementation code

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,[8] then go to any partition and select `Remove Card` (Fig. 3.18). It should immediately refresh, and you'll no longer be able to see the card in either the GUI or in the `Box.xmi` tree in the "instances" folder. Pretty cool, eh?
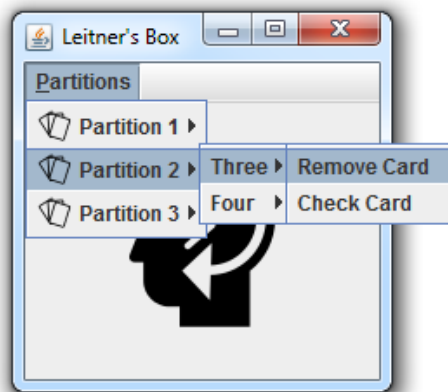
Figure 3.18: Testing `removeCard`

---

4   Checking a card

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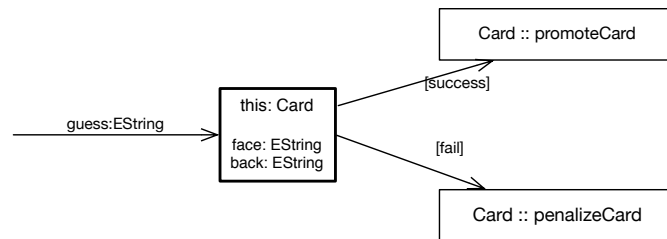

Figure 4.1: Checking a card with a guess

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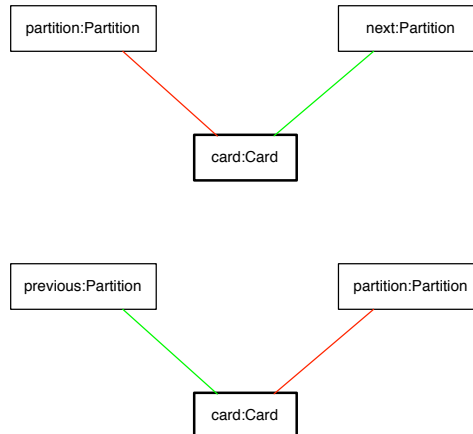
Figure 4.2: Promote (above) and penalize (below) card patterns

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

Attribute Constraint

4.1 Implementing check

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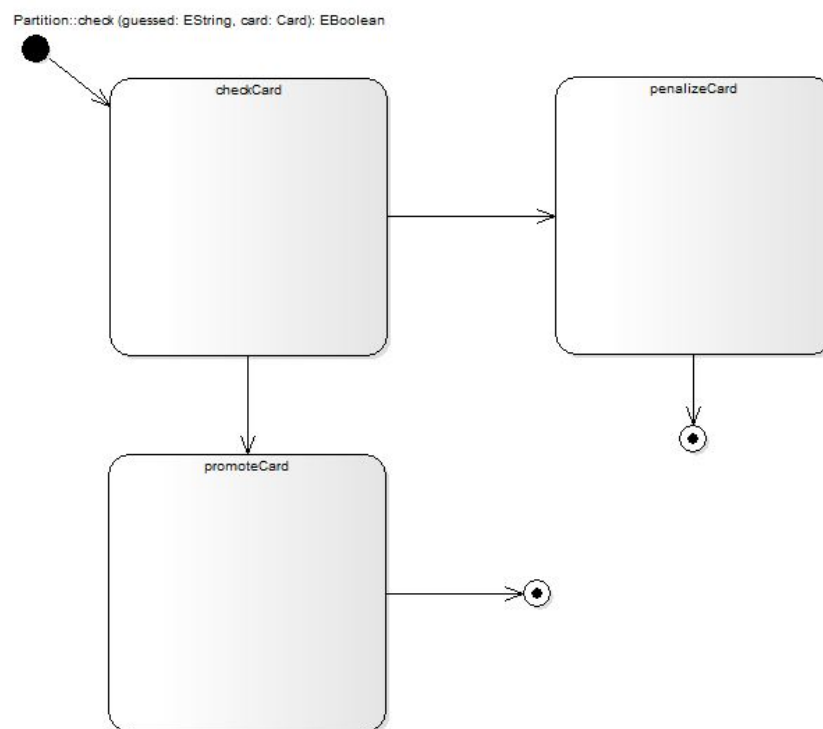
Figure 4.3: Activity diagram for `Partition::check`

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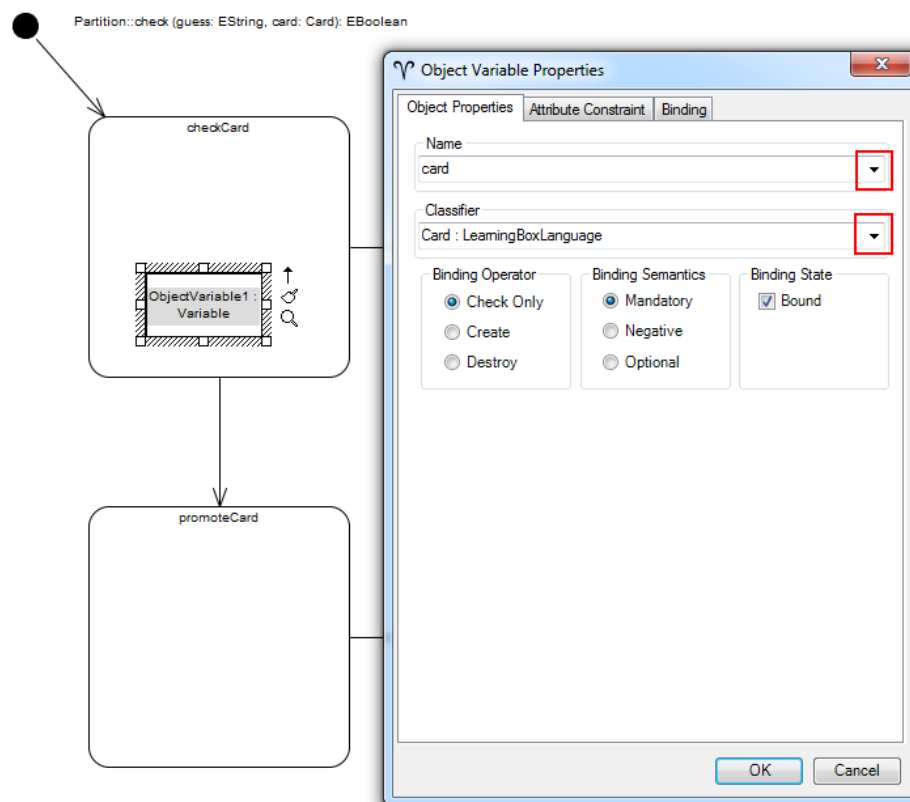
Figure 4.4: Creating the card variable

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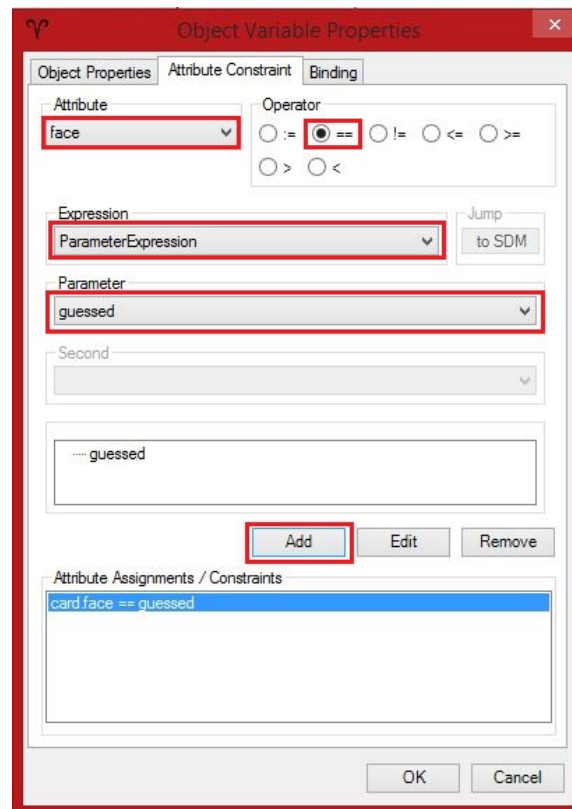
Figure 4.5: Creating an attribute constraint

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

Before building the other two activity nodes, let's quickly return to the control flow. Currently, the pattern branches off into two separate patterns after completing the initial check, and it is unclear how to terminate the method. As the code generator does not know what to do here, this is flagged as a validation error (you're free to press the validation button and take cover). We need to add *edge guards* to change this into an *if/else* *Edge* construct based on the results of the *attribute constraint*. *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. 4.6). Repeat the process for the `Success` edge leading to `promoteCard`.
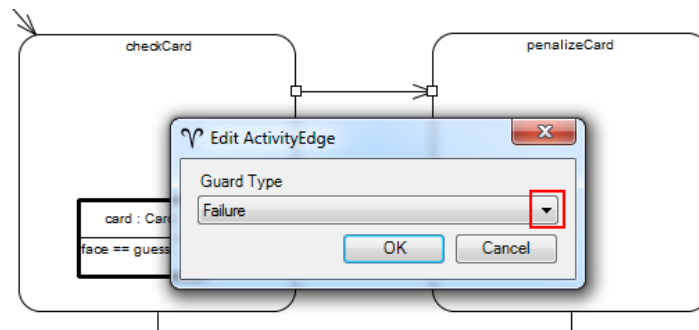
Figure 4.6: Add a transition with a guard

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

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

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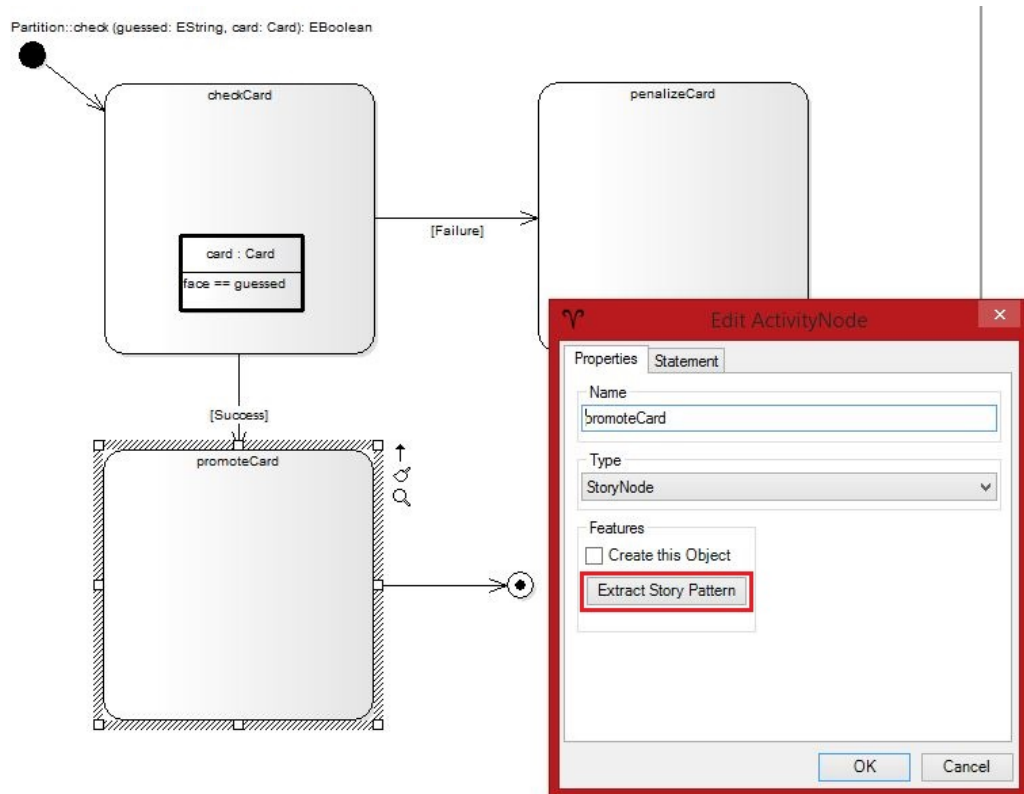

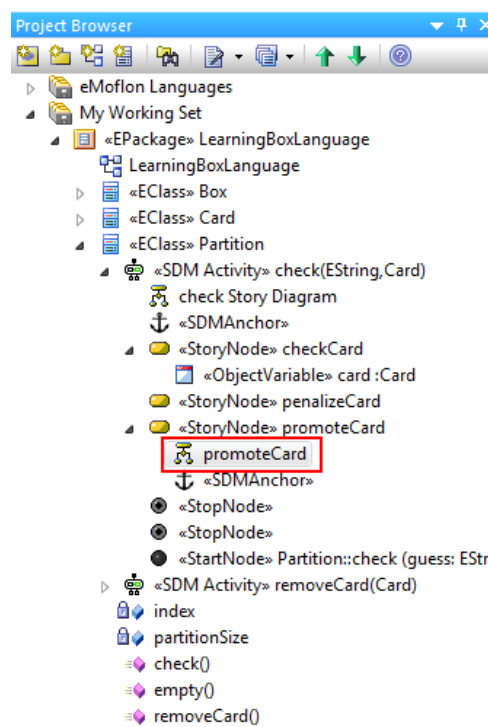Figure 4.7: Extract a story pattern for more space and a better overview

Figure 4.8: A new subdiagram is created automatically

Another EA gesture[9] 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.[10]

▶ To put this into practice, create a new `Card` object variable by drag-and-dropping the class from the project browser into the new (extracted) pattern diagram (Fig. 4.9).
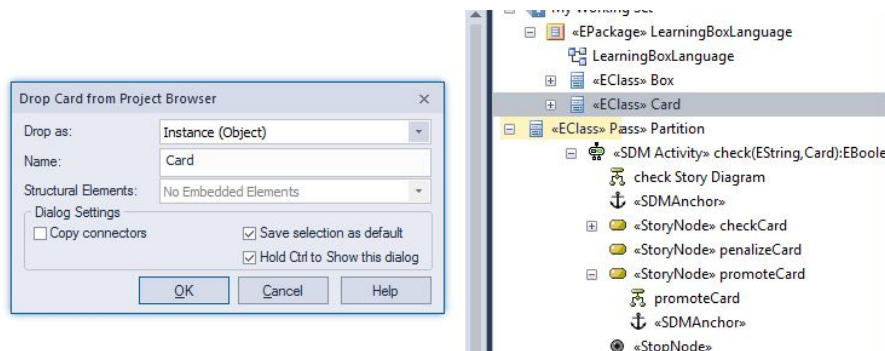
Figure 4.9: Add a new object variable per drag-and-drop

▶ 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 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.

---

[9]The other two gestures we have learnt are "Quick Link" and "Quick Create"

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

▶ After creating the object, the object properties dialogue will open. Set the `Name` to `card` and confirm its `Binding State` is `Bound` (Fig. 4.10).
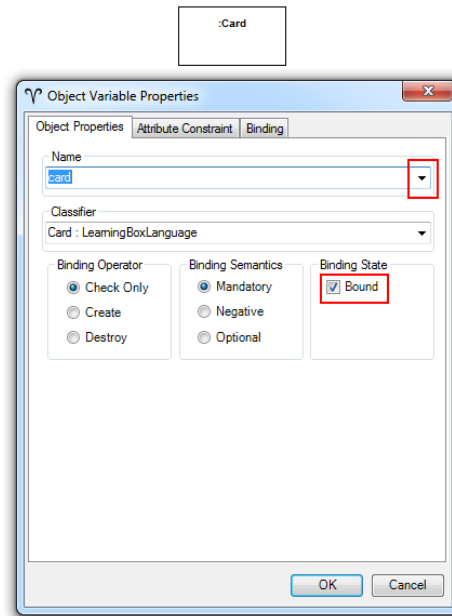
Figure 4.10: Object variable properties of the new card

▶ 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. 4.11.

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

▶ To specify this, quick link from `this` to `nextPartition` (or vice-versa) to establish `next`, as shown in Fig. 4.12. As you can see, there are
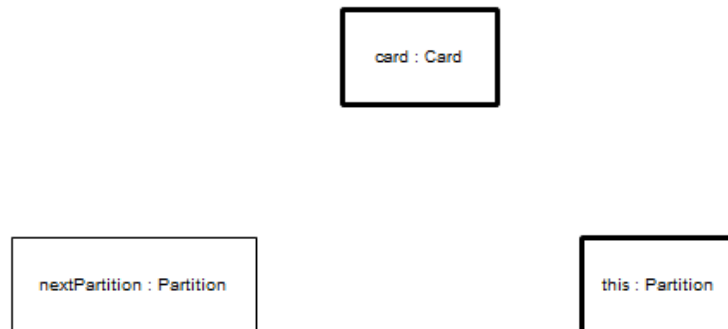
Figure 4.11: All object variables for story pattern `promoteCard`

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. 4.13.
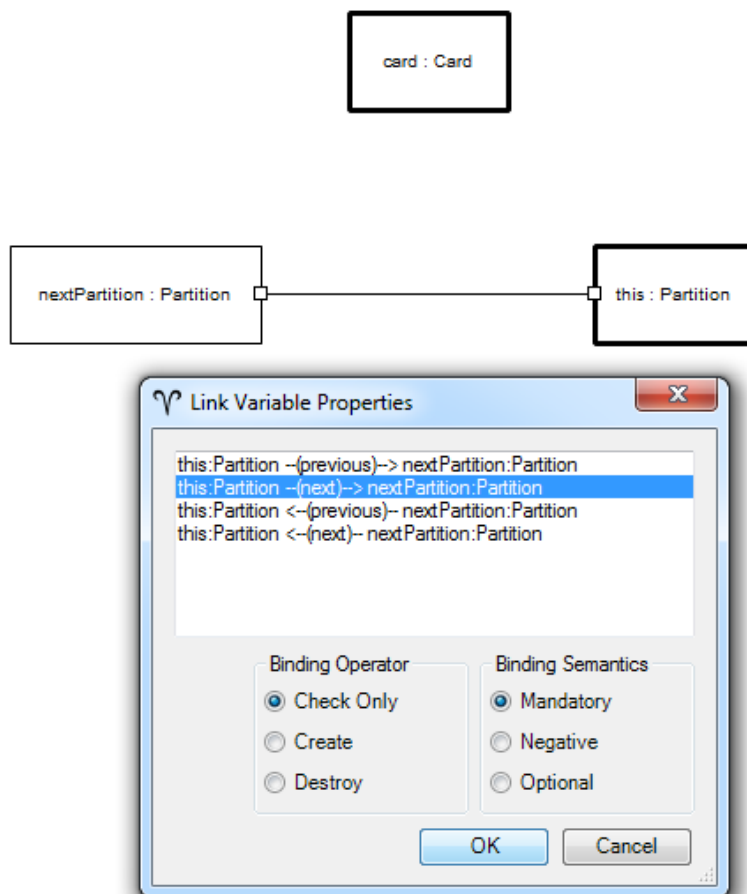
Figure 4.12: Possible links between `this` and `nextPartition`
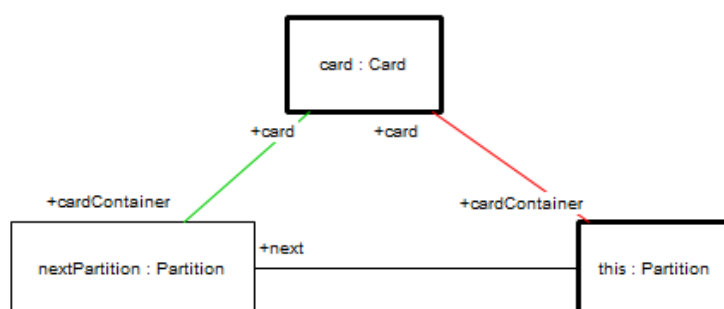
Figure 4.13: Complete story pattern for `promoteCard`

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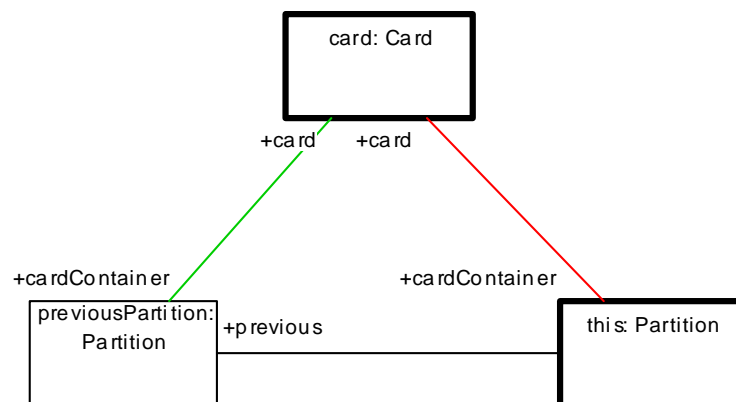
Figure 4.14: Complete story pattern for `penalizeCard`

To complete the `check` activity, we need to signal (as a return value) the result of the check - was the card promoted or penalized? We have no object to return so instead, we need to edit the stop nodes so they return a *LiteralExpression*. This expression type can be used to *LiteralExpression* specify arbitrary text, but should really only be used for true literals like 42, "foo" or `true`. It can be (mis)used for formulating any (Java) expression that will simply be transferred "literally" into the generated code, but this is obviously really dirty[11] 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. 4.15). Change the value in the window below to `true`. Press `OK`, then finish the SDM by returning `false` after `penalizeCard` in the same manner. Note that it is also possible that `promoteCard` or `penalizeCard` fails[12]. This is handled by placing parallel `Success`

---

[11]It defeats, for example, any attempt to guarantee type safety

[12]For example, if the actual partition doesn't have a "proper" successor or predecessor – in this case, unfortunately, we can't give a real reward or penalty for the guess, which means that we still return `true` or `false`.

and `Failure` edges after the `promoteCard` and `penalizeCard` activities. After doing these, your diagram should resemble Fig. 4.16.
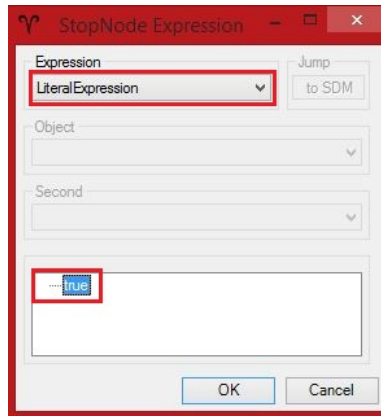
Figure 4.15: Add a return value with a literal expression

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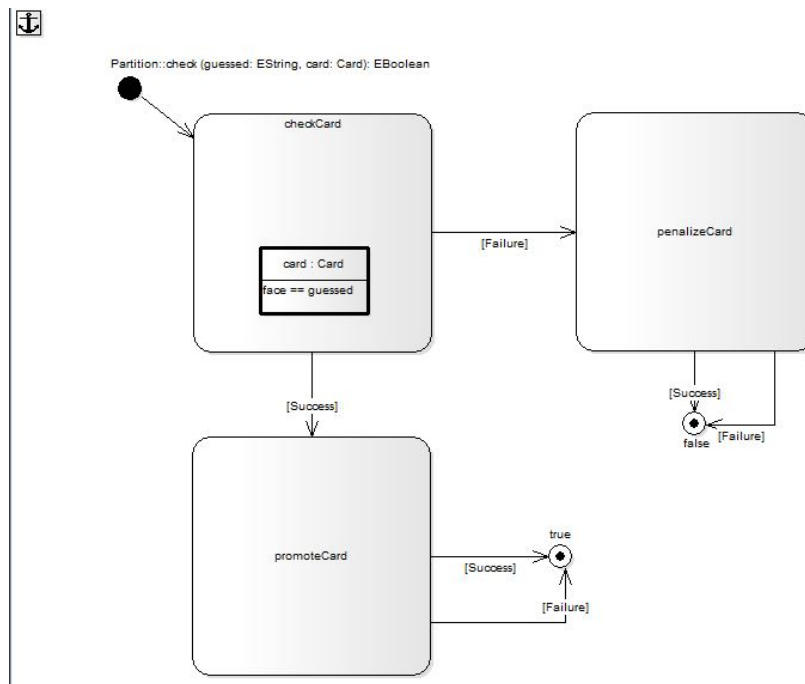
Figure 4.16: Complete SDM for `Partition::check`

5 Running the Leitner's Box GUI

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

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

▶ 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 `LeitnersBoxControl-`
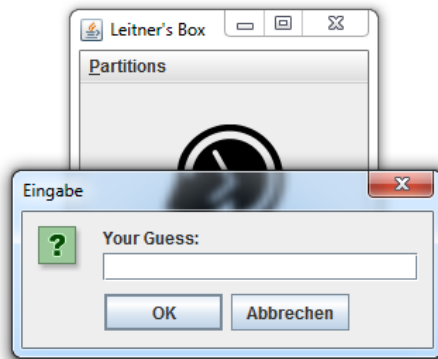
Figure 5.1: Enter your guess

`ler.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,[13] we obviously need some construct for repeatedly deleting each card in the partition (Fig. 6.1).
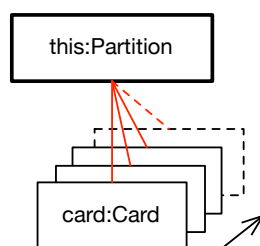
Figure 6.1: Emptying a partition of every card

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

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

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

The answer is no, the pattern would yield the same result, regardless of whether or not the link is explicitly destroyed! This is due to the transformation engine eMoflon uses.[14] It ensures that there are never any *dangling* *Dangling* *edges* in a model. Since deleting just the `card` would result in a dangling *Edges* 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?

---

[13] If there was only one, we would just invoke `removeCard`

[14] CodeGen2, 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. 6.2). Press `OK`, and you'll see that a *for each* node is represented as a stacked node to indicate the potential for repetition.
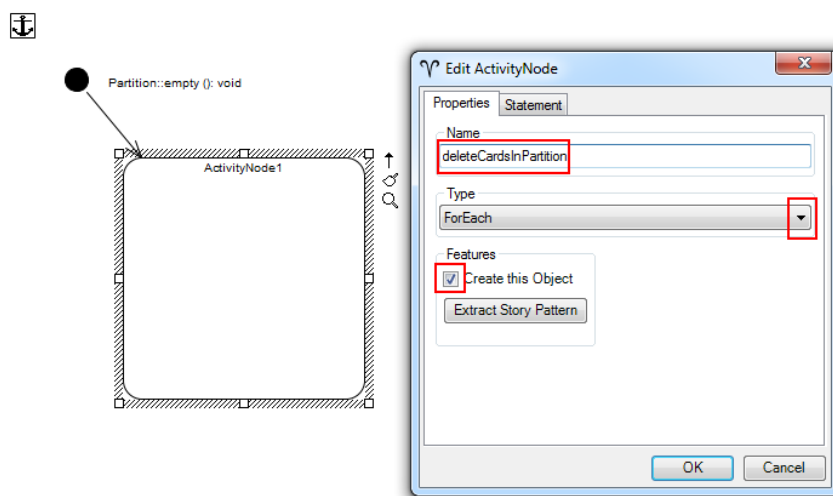
Figure 6.2: Creating a looping story node

▶ Now create the `card` object variable needed to complete this SDM. Unlike `removeCard` (Fig. 3.14) however, the goal of `emptyCards` is not just to remove the link between the selected partition and card, we want the matched `card` to be *completely* deleted. This means in the properties tab, after setting the name and binding state, you'll need to set the `Binding Operator` to `Destroy` (Fig. 6.3).

▶ Complete the story pattern as indicated in Fig. 6.4. Notice that the guard that terminates the looping node has an `[end]` edge guard. Indeed, a *for each* story node *must* execute an `end` activity when all matches in the pattern have been handled. `empty` is defined as a `void` method, so don't worry about setting any return value in the stop node.
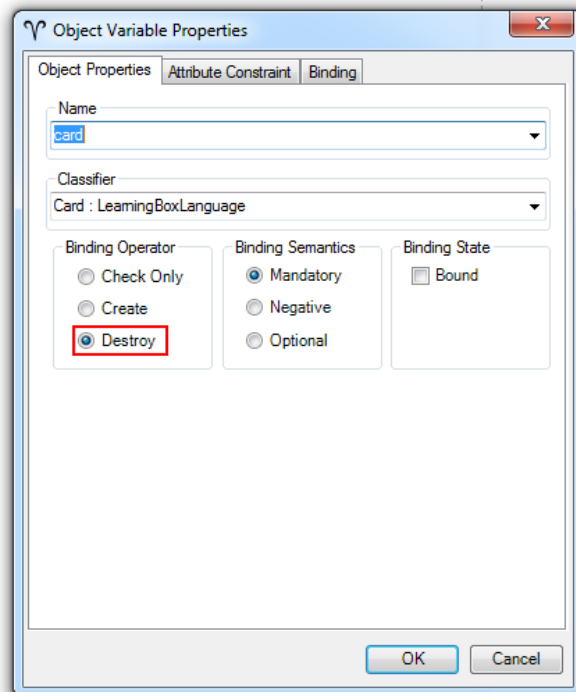
Figure 6.3: Editing `card` so it gets destroyed
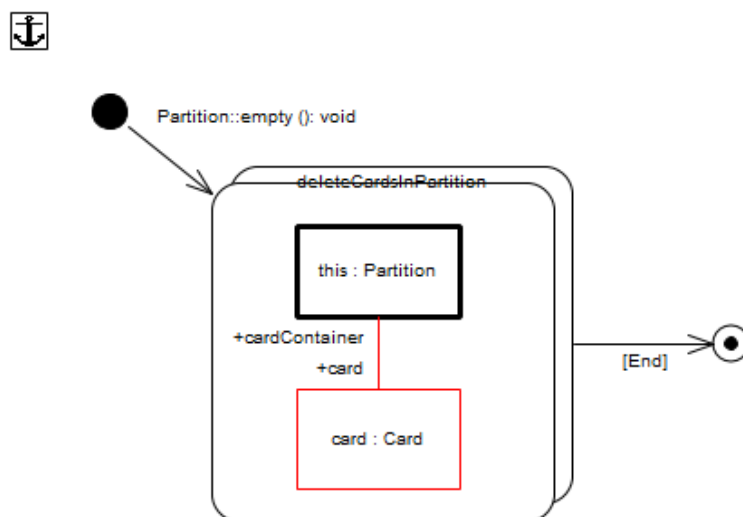
Figure 6.4: Completed `empty` story pattern

▶ Done! You've now learnt that in order to create a repeating action, all you need to do is change a standard story node into a `for each` node, and use appropriate *edge guards*.

▶ As always, save and build your metamodel.

▶ Although the Learning Box GUI does not have an explicit action that invokes this SDM, feel free to extend it and see your SDM in action!

7 Inverting a card

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

card:Card
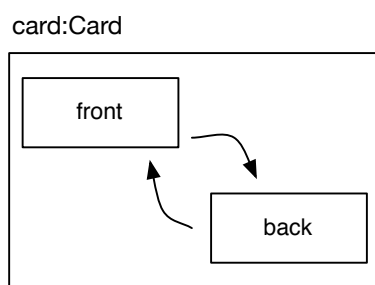
Figure 7.1: Inverting the attributes of a `Card`

Something new that we'll use in this SDM are *assignments* to set the at- *Assignments* tributes of a `temp` object variable with `card`, then again to actually swap the `card` values. An assignment is simply an attribute constraint[15] with a ':=' operator. Though it may be slightly confusing to refer to an assignment as a constraint, if you think about it, *everything* can be considered as a constraint that must be fulfilled using different strategies.

With `invert`, a successful match is achieved not by searching as you would with a comparison (`==, >, <, ...`), but by *performing* the above assignment. If the assignment cannot be completed, the match is invalid. Similarly, non-context elements (set to create or destroy) can be viewed as structural constraints that are fulfilled when the corresponding element is created or destroyed. A constraint is therefore a unifying concept similar to "everything is an object" from OO, and "everything is a model" from metamodelling. If you're interested in why *unification* is considered cool, check out [1].

---

[15]Which we first encountered in `check`

7.1   Implementing invert

▶ If you've completed all the work so far, you've got to be *really* good at SDMs now. Model the simple story diagram depicted in Fig. 7.2.
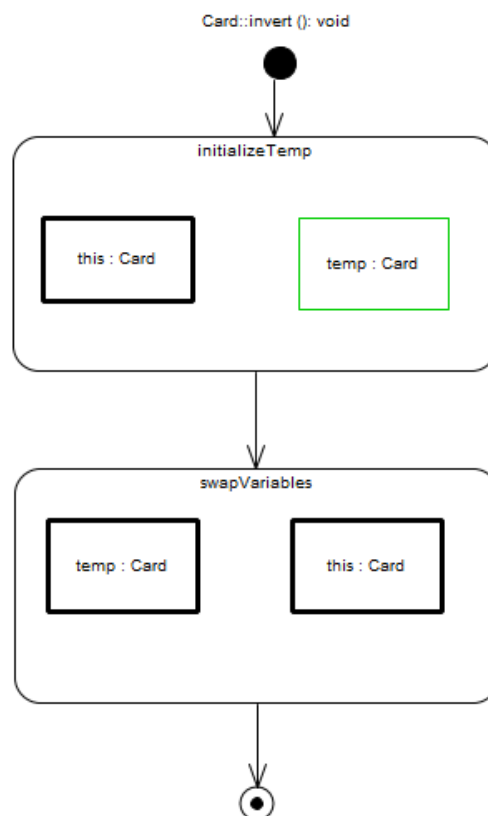
Figure 7.2: Imperative control layer for inverting a card

▶ Note that the binding operator on the first `temp` object variable is set to `create` (thus the green border). This means that we actually *create* a new object, and do not pattern match to an existing one in our model.

▶ This activity will need four assignment constraints - two in `initia-lizeTemp` (to store the "opposite" values), and two in `swapVariables` (to switch the values). Create your first assignment constraint by going to the created `temp` card and using the ':=' operator to set the `temp.back` value to `this.face` (Fig. 7.3).
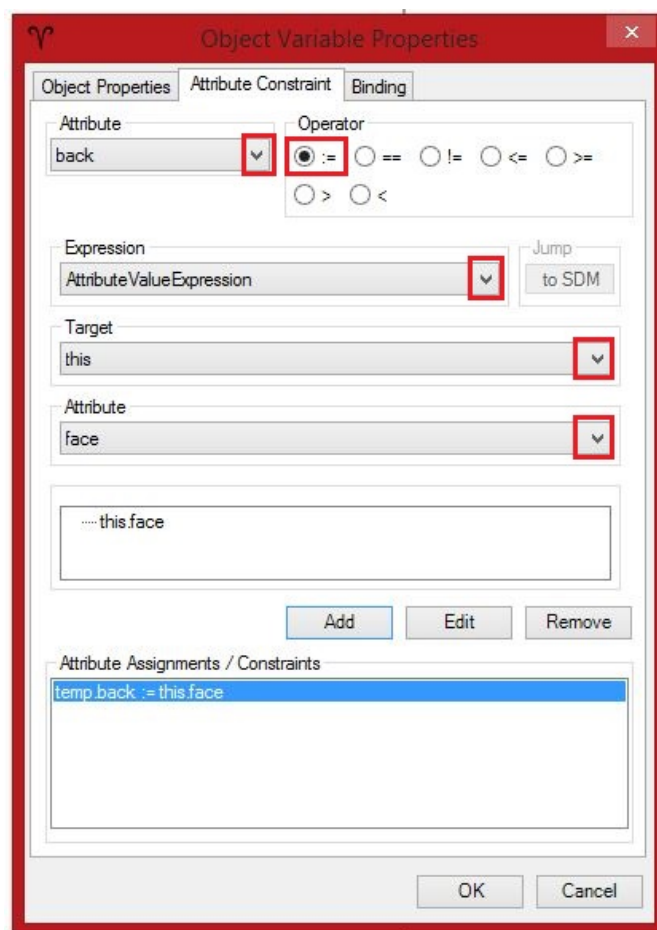
Figure 7.3: Store the `back` and `face` values of the card in `temp`

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

Card::invert (): void

initializeTemp

this : Card

temp : Card

back := this.face
face := this.back

swapVariables

temp : Card

this : Card

back := temp.back
face := temp.face
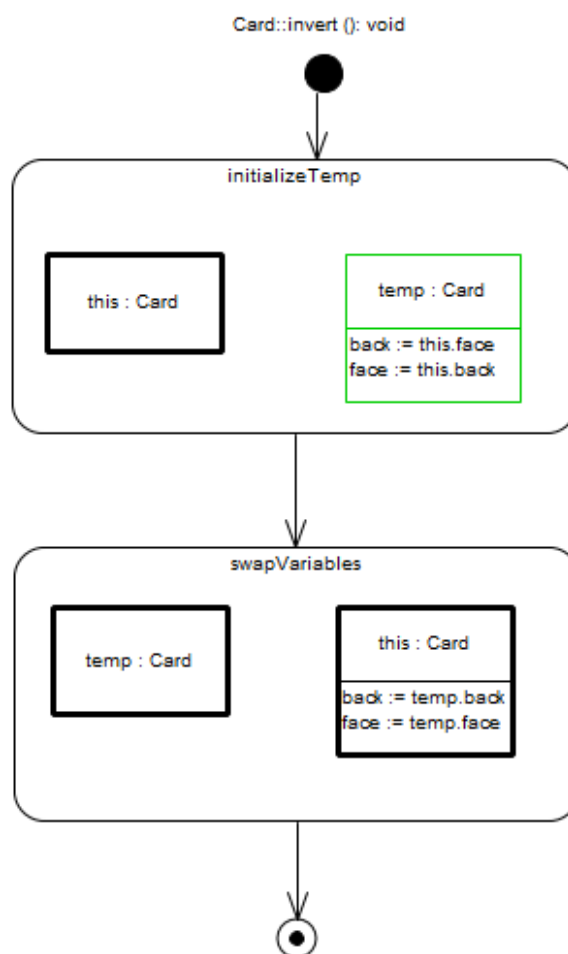
Figure 7.4: Swap back and face of the card

▶ Believe or not, that's it! You don't *have* to export and build to Eclipse, but it's always nice to confirm your work is error free.

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`?[16] 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!

---

[16]See Fig. 7.4

8 Growing the box

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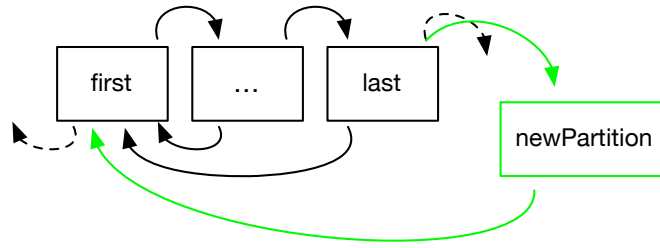
Figure 8.1: Growing a box by inserting a new partition

SDMs provide a declarative means of identifying specific partitions via *Negative Application Conditions*, simply referred to as NACs.[17] NACs express 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,[18] namely only single negative elements (object or link variables). *NAC*

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

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

---

[17]Pronounced \ˈnak\
[18]To be precise, in CodeGen2 from Fujaba

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

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

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

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

Figure 8.2: Implementation of `removeCard`

▶ Save the file, then right-click on it, either in the package explorer or in the editor window, and choose "eMoflon/ Create/Update Injection for class" from the context menu.

▶ Confirm the update in the new `BoxImpl.inject` file's partial class. `determineNextSize` is now ready to be used by your metamodel!

8.1 Implementing grow

▶ Start by creating the simple story pattern depicted in Fig. 8.3. This matches the box and *any* two partitions.[19]
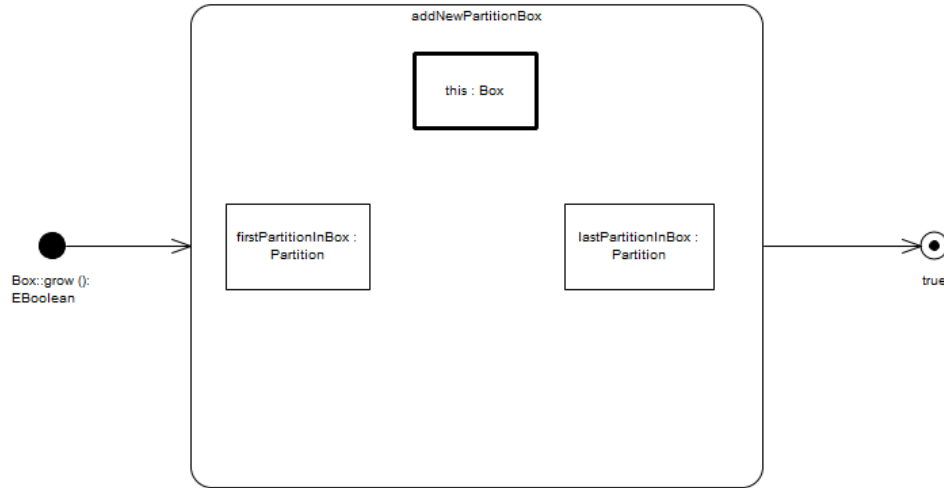
Figure 8.3: Context elements for SDM

▶ To create an appropriate NAC to constrain the possible matches for `lastPartitionInBox`, create a new `Partition` object variable `nextPartition` and set its *binding semantics* to `negative` (Fig. 8.4). The object variable should now be visualised as being cancelled or struck out.

▶ Now, quick link `nextPartition` to `lastPartitionInBox`. Be sure to choose the link type carefully! The `nextPartition` should play the role of `next` with respect to `lastPartitionInBox`. This combination (the negative binding and reference) tells the pattern matcher that if the (assumed) last partition has an element connected via its `next` reference, the current match is invalid.

▶ Great work – the first NAC is complete! In a similar fashion, create the NAC for `firstPartitionInBox`. Name the negative element `previousPartition`, and again, be sure to double-check the link variable.

▶ Finally, complete the pattern so that it closely resembles Fig. 8.5.

---

[19]Remember, the *pattern matcher* is non-deterministic.
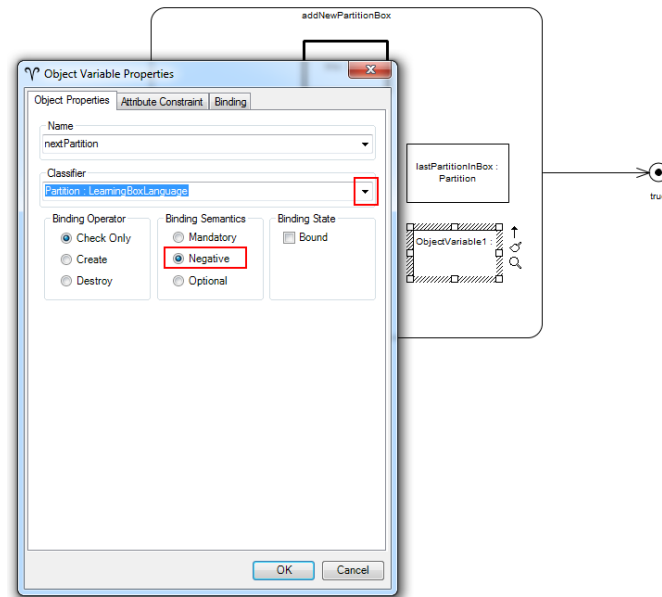
Figure 8.4: Adding a negative element
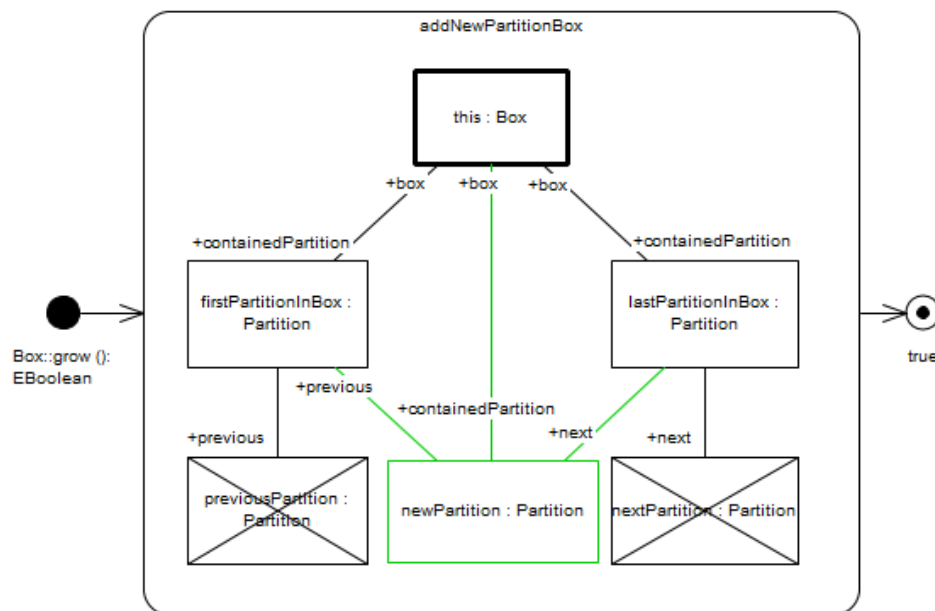
Figure 8.5: Determining the first and last partitions with NACs

► Notice how the created partition `newPartition` is 'hung' into the box. It becomes the next partition of the current *last* partition, and its previous partition is automatically set to the first partition in the box (as dictated by the rules set in Fig. 1.1). In other words, the new partition is appended onto the current set of partitions.

► In order to complete `grow`, we need to set the size of the `newPartition`. Given that the new size is calculated via the helper function `determineNextSize`, we need to use a *MethodCallExpression*. Go ahead and invoke the corresponding dialogue, activate the assignment (`:=`) operator, and match your values to Fig. 8.6.
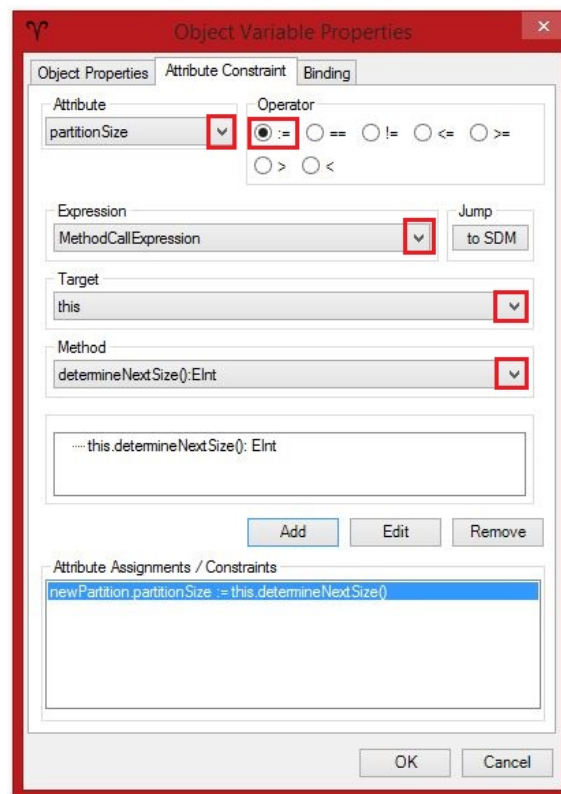
Figure 8.6: Invoking a method via a `MethodCallExpression`

► Since `determineNextSize` doesn't require any parameters, you can ignore the `Parameter Values` field this time.

► If you've done everything right, your SDM should now closely resemble Fig. 8.7. As usual, try to export, generate code, and inspect the method implementation in Eclipse.

▶ That's it – the `grow` SDM is complete! This was probably the most challenging SDM to build so give yourself a solid pat on the back. If you found it easy, well then ...I guess I'm doing my job correctly.
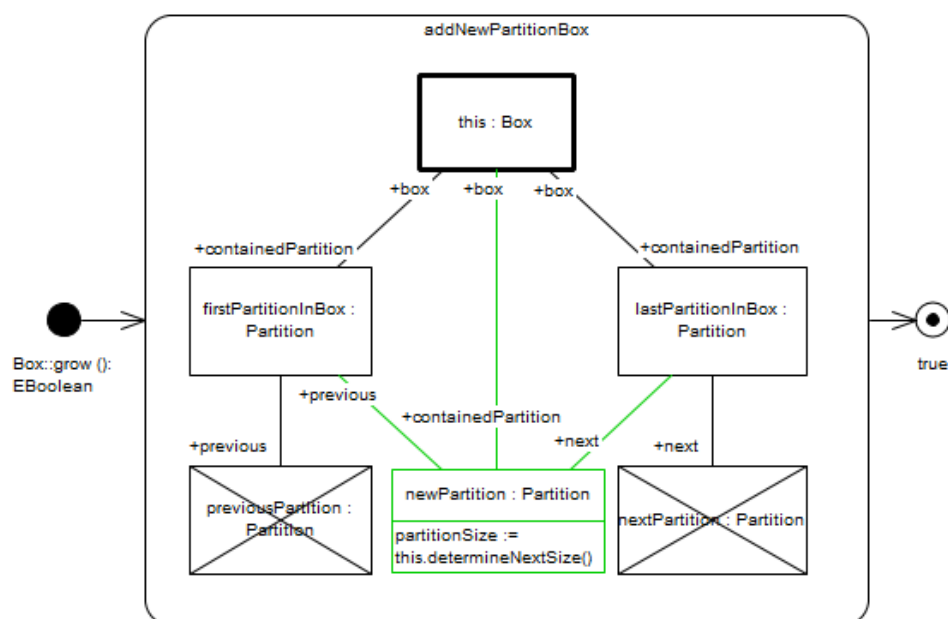
Figure 8.7: Complete SDM for `Box::grow`

9   Conditional branching

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

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

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

Statement Node

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

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

9.1 A short note on the `initializeBox` method

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

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

}
```

Figure 9.1: Code generated for branching with a statement node

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

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

Figure 9.2: The `initializeBox` declaration

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

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

9.2   Branching with statement nodes

▶ Currently, there is no method to help us initialize `box` from its pristine state (no partitions). Create one by editing your metamodel (the `LearningBoxLanguage` diagram) and invoking the `Operations` dialogue by first selecting `Box`, then pressing `F10`.[20]

▶ Name the new method `initializeBox` and, recalling the one rule of conditional branching, set its return type to `EBoolean`.

▶ Save and close the dialogue, then re-open the `grow` SDM and *Quick Create* a new story node from `addNewPartition`.

▶ This will be the node we'll use to invoke our helper method. Double click the node to invoke its properties editor and switch the `Type` to a `StatementNode`. Name it `initialize` (Fig. 9.3a).

▶ Before closing the dialogue, switch to the `Statement` tab, and create a `MethodCallExpression` to invoke your newest method (Fig. 9.3b). We want to access the `Box` object (`this`) and its `initalizeBox` method. It doesn't require any parameters, so leave the values field empty.
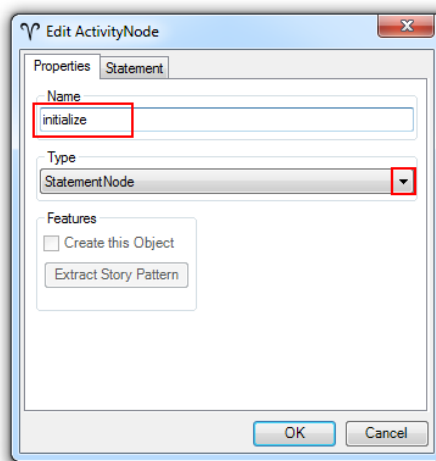
(a) Create a new *StatementNode*   (b) Edit the `MethodCallExpression`
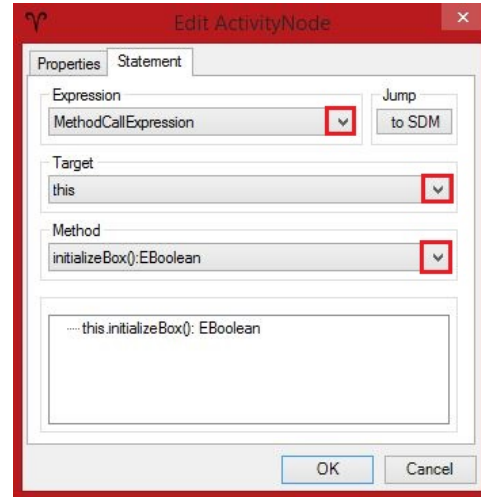
Figure 9.3

▶ Now we need to update the edge guards stemming from `addNewPart-itionInBox`. Given that we only want to call `initializeBox` if the pattern fails, change the edge guard leading to your statement node to `Failure`. Similarly, update the edge guard returning `true` to `Success`.

▶ Finally, attach two stop nodes – `true` and `false` – along with their appropriate edge guards from `initialize`. These indicate that if the method execution worked, the box could be initialized. If it failed however, `box` was in an invalid state (by e.g., having only one partition) and returns `false`. Overall, the new additions to `box.grow()` should resemble Fig. 9.4.
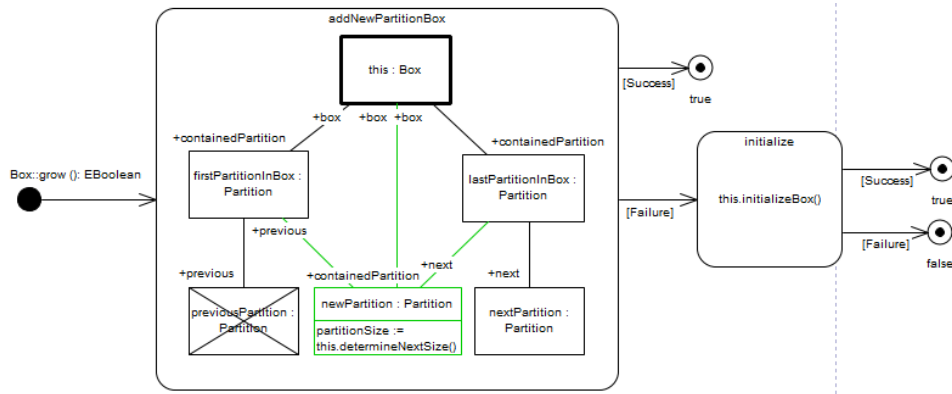
Figure 9.4: Extending `grow` with a *MethodCallExpression*

▶ To review our work up to this point, we have declared `initializeBox` and invoked it from a statement node. We have yet to actually specify the method however. Double-click the anchor to return to the main diagram and create a new SDM for `initializeBox`.

▶ Create a normal activity node named `buildPartitions` with the pattern depicted in Fig. 9.5.
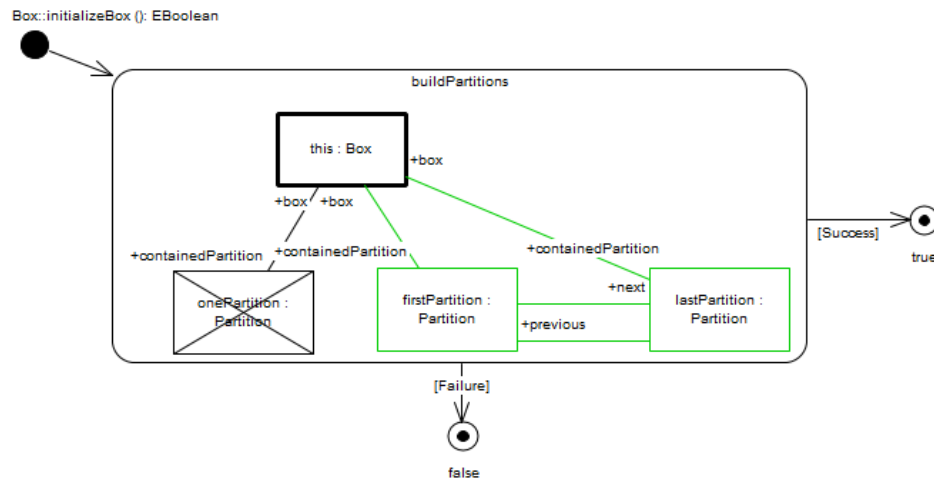
Figure 9.5: Complete SDM

▶ The NAC used here is only fulfilled if the box has absolutely no partitions, i.e., is in a pristine state and can be initialized. In other words, if `grow` is used for an empty box, it initializes the box for the first time and grows it after that, ensuring that the box is always in a valid state.

▶ You're finished! Save, validate, and build your metamodel.

10 A string representation of our learning box

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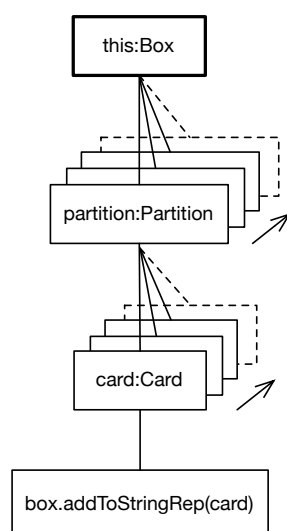
Figure 10.1: Nested *For Each* loops

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

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

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

Figure 10.2: Implementation of `addToStringRep`

10.1 Implementing toString for `Box`

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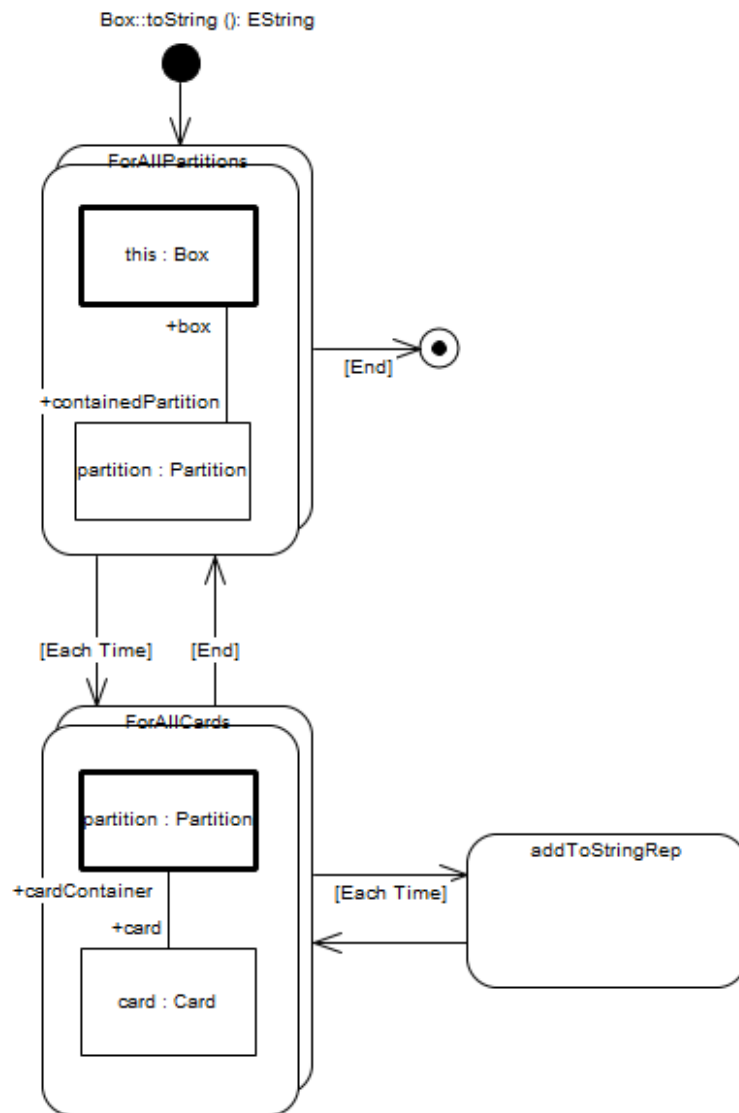
Figure 10.3: Control flow with nested loops

▶ Knowing `addToStringRep`'s default node type will not allow it to invoke our helper method, change it into a `StatementNode`. Then, analogously to how you established a *MethodCallExpression* for `grow`, have this node invoke `this.addToStringRep(Card)` (Fig. 10.4).

▶ You can see in the `Method` statement that a `Card` parameter is required. For this double click on the field below and enter the values like in Fig. 10.5. Now we included `card` so we may pass the object variable to the method.
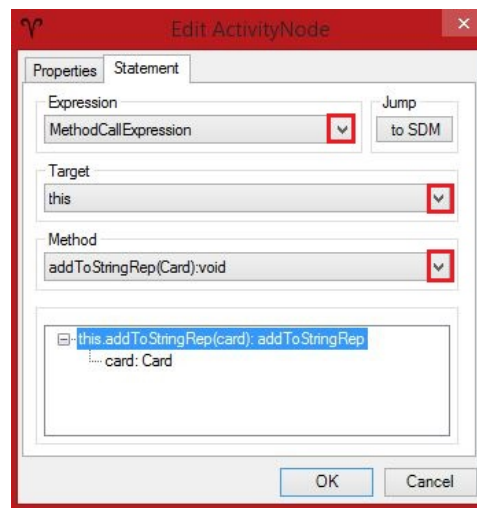
Figure 10.4: Add a *MethodCallExpression*
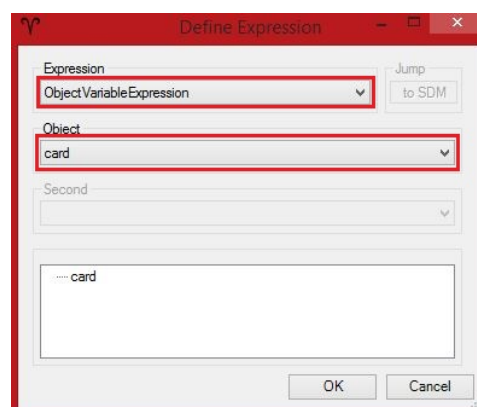
Figure 10.5: Add a parameter to the *MethodCallExpression*

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

AttributeValue-Expression
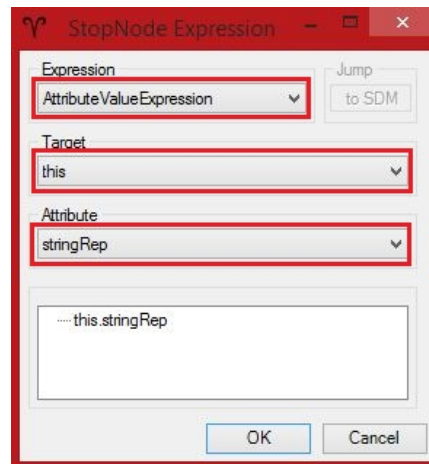
Figure 10.6: Specify a return value as an *AttributeValueExpression*

▶ Take some time to compare and reflect on the complete SDM as depicted in Fig. 10.7. The idea was to abstract from the actual text representation of the box and model the necessary traversal of the data structure. The helper method `addToStringRep` could, for example, build up something totally different.

▶ While modelling this SDM, we have seen that *for each* story nodes can be nested, and have used a *MethodCallExpression* to invoke a void helper method only for its side effects (building up the string representation of the box).

▶ As always, save, validate, and build your metamodel in Eclipse.
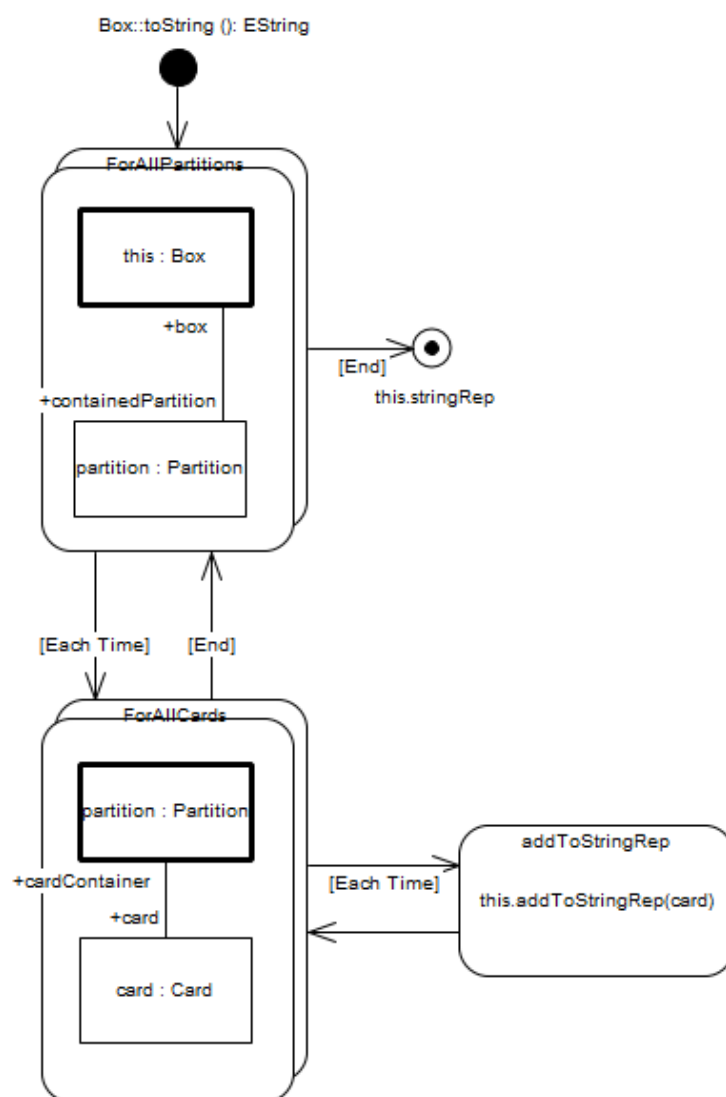
Figure 10.7: The complete SDM for Box::toString

11   Fast cards!

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

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

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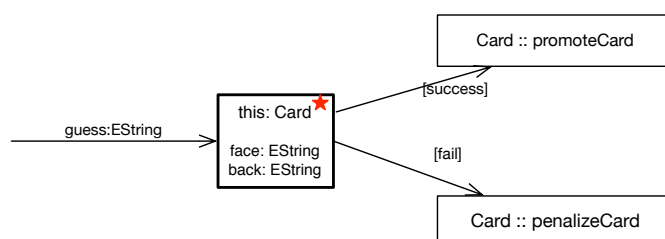
Figure 11.1: Checking a fast card against a guess

In addition to creating a new `EClass`, we also need to extend the existing `check` method to check for this special card type once a guess is determined to be correct. Now `check` needs to decide, based on the dynamic type[21] 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`.

---

[21]In a statically typed language like Java, every object has a static type (determined at compile time) and a dynamic type (that can only be determined at runtime).

11.1 Implementing FastCards

▶ To introduce fast cards into your learning box, return to the 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. 11.2). Your metamodel should then resemble Fig. 11.3.
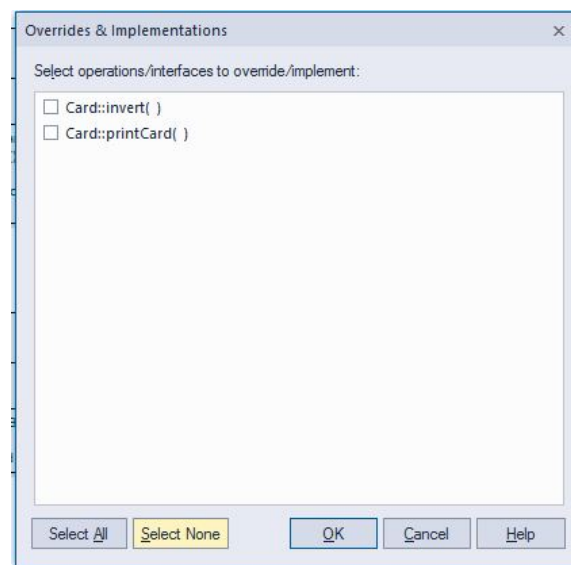
Figure 11.2: Selecting operations to override
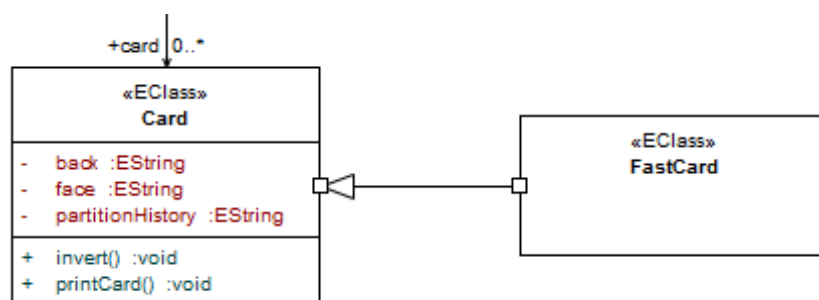
Figure 11.3: Fast cards are a special kind of card

► Now return to the `check` SDM (in `Partition`) and extend the control flow as depicted in Fig. 11.4.
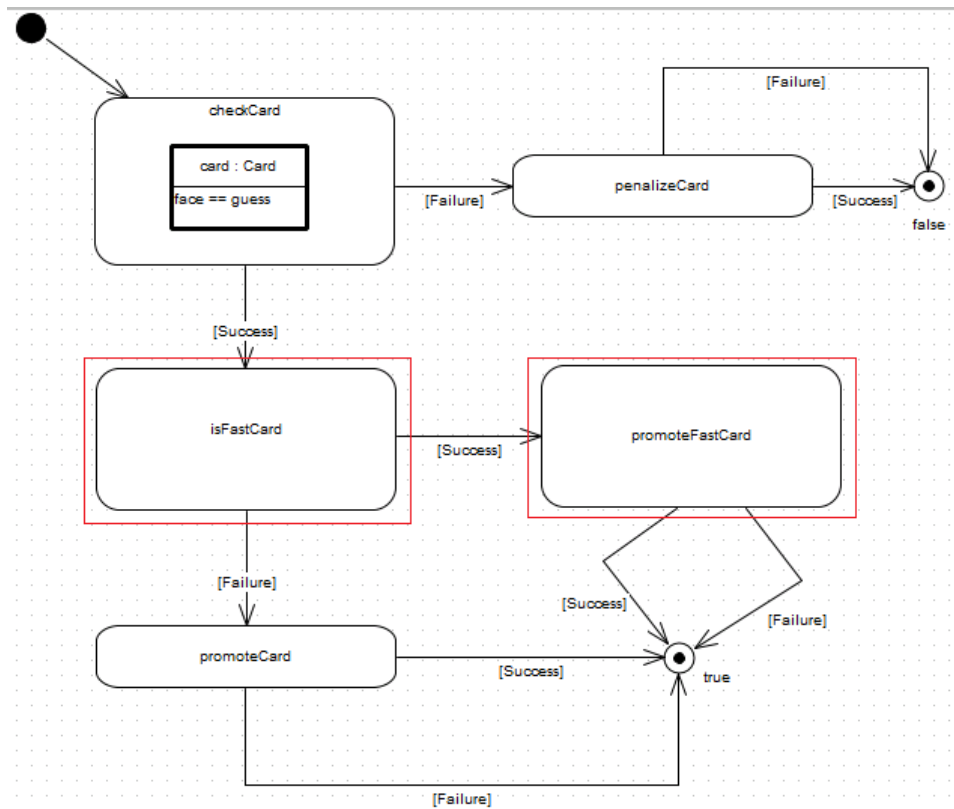
Figure 11.4: Extend check to handle fast cards.

► As you can see, you have created two new story nodes, `isFastCard`, and `promoteFastCard`.

► Next, in order to complete the newest conditional, create a bound `FastCard` object variable, named `fastcard` in `isFastCard` (Fig. 11.5).

► To check the dynamic type, we'll need to create a binding of `card` (of type `Card`) to `fastcard` (of type `FastCard`), so edit the `Binding` tab in the `Object Variable Properties` dialogue (Fig. 11.5). Please note that this tab will not allow any changes unless the `bound` option in `Object Properties` is selected. As you can see, this set up configures the pattern matcher to check for types, rather than `parameters` and `attributes` as we've previously encountered.

checkCard

card : Card

face == guess

[Success]

isFastCard

fastcard : FastCard :=
card

[Failure]

promoteCard

Object Variable Properties

Object Properties | Attribute Constraint | Binding

Expression

ParameterExpression ▼

Jump

to SDM

Objects

card ▼

Source

▼

Parameter Values / LiteralExpression

OK | Cancel
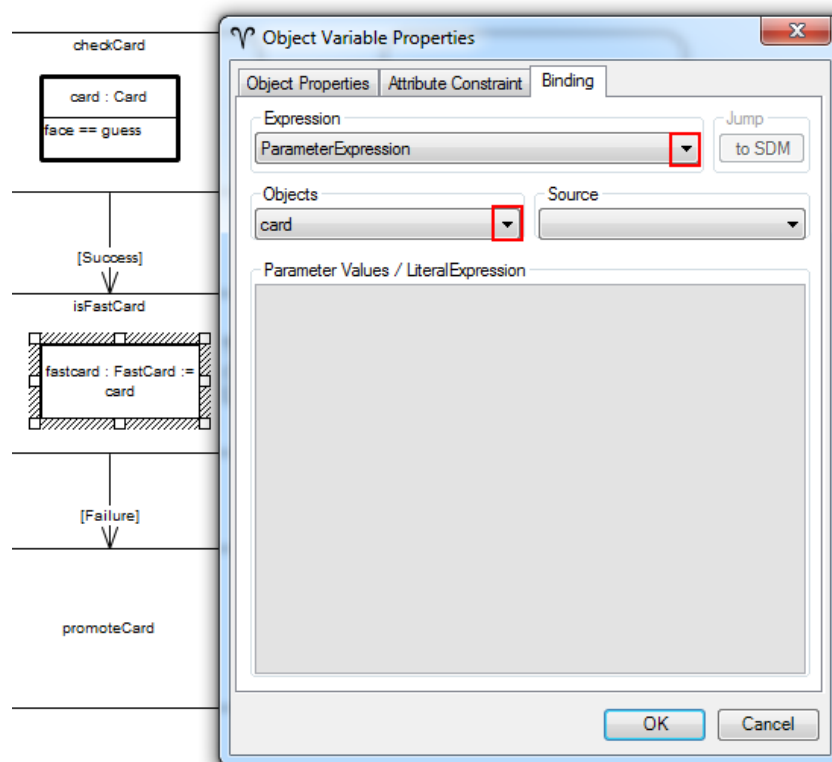
Figure 11.5: Create a binding for `fastcard`

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

▶ Update the `fastcard` binding by switching the expression to `ObjectVariableExpression`, with `card` as the target. Note that a binding could also use a *MethodCallExpression* to invoke a method whose return value would be the bound value. This is very useful as it allows invoking helper methods directly in patterns.

▶ To finalize the SDM, (i) extract the `promoteFastCard` story pattern and build the pattern according to Fig. 11.6 and (ii) create the parallel `Success` and `Failure` edges from this activity to the stop node returning `true` for the same reason as in `check` earlier. Compare the pattern in Fig. 11.6 to Figs. 4.13 and 4.14, the original promotion and penalizing card movements. As you can see, they're very similar, except `fastCard` is transferred from the current partition (`this`) immediately to the last partition in `box`, identified as having no next `Partition` with an appropriate NAC. Note that a second NAC is used to handle the case where `this` would be the next `Partition`, which is also not what we want.
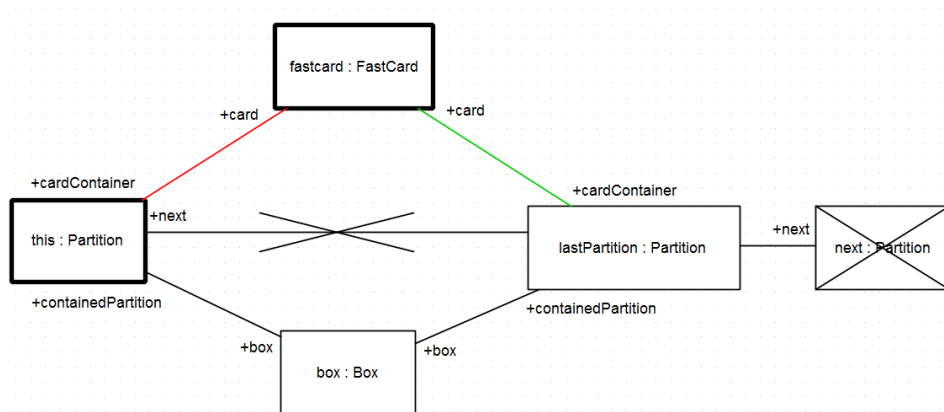
Figure 11.6: Story pattern for handling fast cards.

▶ 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`?

11.2 FastCards in the GUI

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

▶ To review the details of creating instances, read Part II, Section 3 but for now, open `Box.xmi`, right-click on your first partition, and create a new `FastCard` child element. Open the properties tab below, and edit the `back` and `face` values for testing (Fig. 11.7). As you can see, it has all the same attributes as a standard `card`.
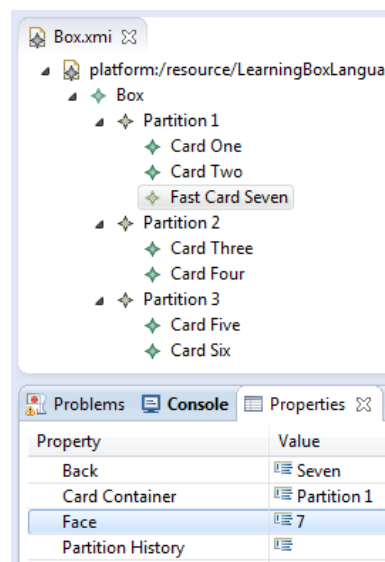
Figure 11.7: Creating and editing a new `FastCard` element

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

12   Reviewing eMoflon's expressions

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

The alternative approach to eMoflon would be to support arbitrary expressions, for example, in a script language like JavaScript or in an appropriate DSL[22] 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 *LiteralExpression*s 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.

---

[22]A DSL is a Domain Specific Language: a language designed for a specific task which is usually simpler than a general purpose language like Java and more suitable for the exact task.

ObjectVariableExpression:

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

13 Complex Attribute Manipulation

In Section 4.1 we have modeled the `Partition::check()` method. However, as you might realized, our implementation ignores the `partitionSize`, i.e., the maximal number of cards that fit in a partition. In order to prevent our implementation from "bursting" a partition we gonna use *complex attribute constraints* to extend the SDM for the `Partition::check()` method. In contrast to simple attribute constrains[23], which provide only binary operations for comparing attribute values (or literals), complex attribute constraints provides a language to specify arbitrary relations on attributes.

13.1 Using Complex Attribute Constrains

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

```
          «EClass»
          Partition

-    index: EInt
-    partitionSize: EInt
-    currentSize: EInt
```
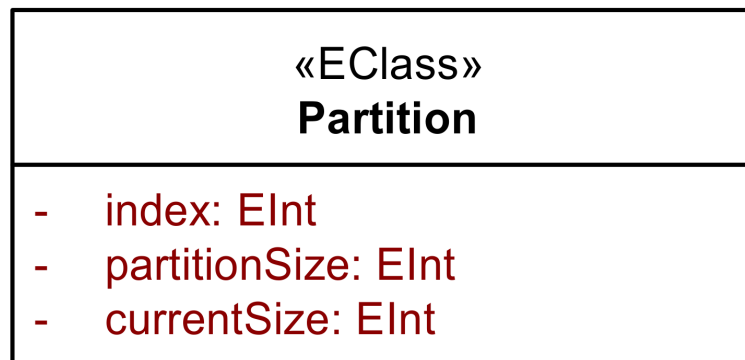
Figure 13.1: adding `currentSize:EInt` to `Partition`

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

Now we can start defining complex attribute constraints:

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

---

[23]First used in Section 4.1 to compare user's guess against the card's face value
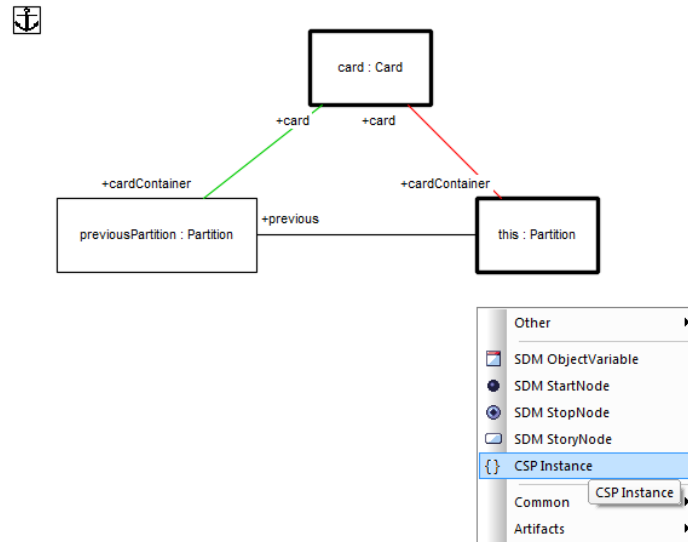
Figure 13.2: Creating a new `CSP instance`

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

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

Basically an attribute constraint is a predicate of the form

$$\texttt{SYMB(param}_1\texttt{,...,param}_n\texttt{)}$$

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

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

- A *constant* is of the form
  `value:Type`,
  as for example, `1:EInt` (first line Fig. 13.3) represents constant 1 of type `EInt`.

- An *attribute variable* of the form
  > objectVarName.attributeName,

  and refers to the current value of the attribute `attributeName` of object variable `objectVarName`, while an attribute variable of the form `objectVarName.attribteName'` refers to an attribute value after the transformation. This is necessary for, e.g., write the java expression `a.x=a.x+1` as the predicate `+(a.x', a.x, 1:EInt)`.
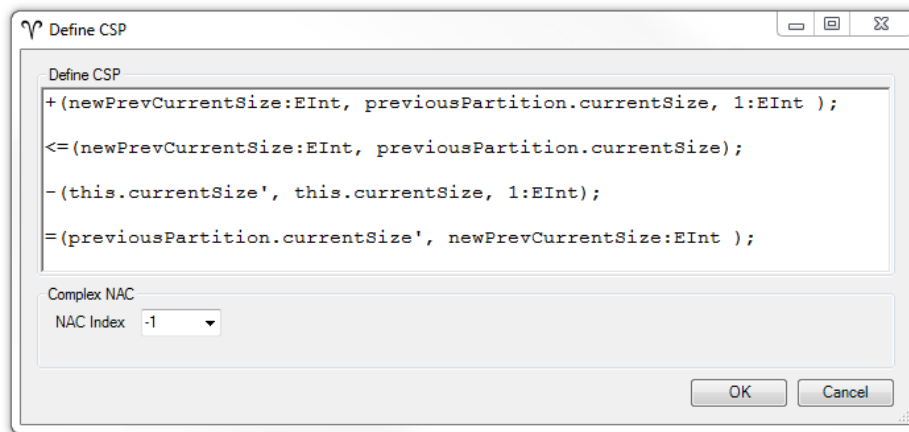
Figure 13.3: Defining complex attribute constraints

The `CSP instance` (the set of attribute constraints) shown in Fig. 13.3 consist of the constraints with meaning [24]:

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

```
<=(newPrevCurrentSize:EInt, previousPartition.currentSize),
```
meaning that:
$newPrevCurrentSize \leq previousPartition.currentSize$;

```
-(this.currentSize', this.currentSize, 1:EInt),
```
meaning that:
```
this.currentSize' == this.currentSize - 1;
```

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

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

---

[24] An complete list of build in constraints is given at the end of this chapter.

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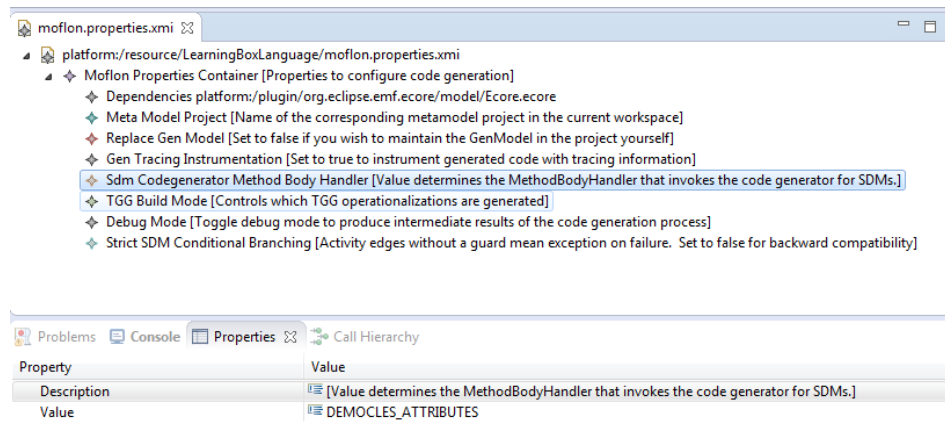
Figure 13.4: Switch SDM Codegenerator Method Body Handler to DEMO-CLES_ATTRIBUTES

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

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

Figure 13.5: LHS pattern code generated for the CSP instance

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

Figure 13.6: RHS pattern code generated for the CSP instance

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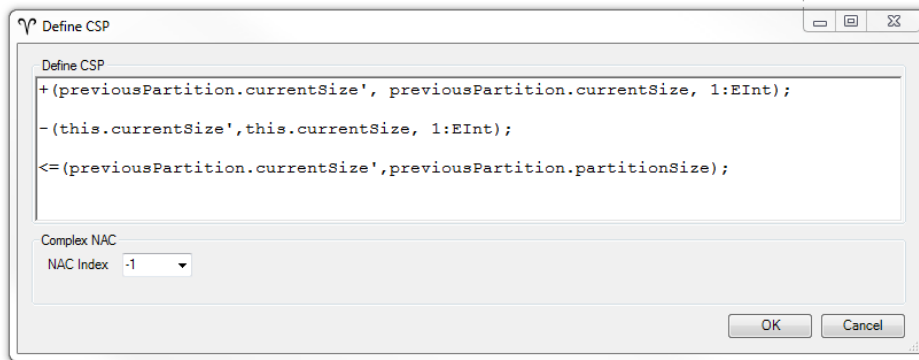
Figure 13.7: Defining complex attribute constraints

13.2   Defining Own Complex Attribute Constraints

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

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

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

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

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

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

▶ just adding the CSP instance:
  myConcat(this.stringRep', this.stringRep, card.face, card.back)
  to the ForAllCards pattern. The complete SDM should now look as shown in Fig. 13.8.

▶ Export as usual. If you are trying to build the metamodel project in eclipse you get an error message (Fig. 13.9) that informs you that the attribute constraint with signature:
  myConcat(:EString, :EString, :EString, :EString)
  is unknown. If you look at LearningBoxLanguage/lib/ there is a new file LearningBoxLanguageAttributeConstraintsLib.xmi

▶ Open the file. It contains a constraint specification myConcat (Fig.13.10 bottom) that represents the signature (myConcat(:EString, :EString, :EString, :EString)) which is derived from the information of the CSP-instance in EA. To define the meaning of myConcat we have to define operations for the constraint.

▶ Right click the operation specification group for myConcat (top of Fig. 13.10) and add as new child an operation specification.

An operation[25] is specified by an *Adornment String*, and a Specification that is a template defining the code to be generated.

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

---

[25]Double click on the operation specification to open the properties view.

Box::toString (): EString

ForAllPartitions

this : Box

+box

+containedPartition

partition : Partition

{+(this.stringRep',this.stringRep, "Partition Content:":EString);}

-1

[End]

this.stringRep

[Each Time]

[End]

ForAllCards

partition : Partition

+cardContainer

+card

card : Card

{myConcat(this.stringRep', this.stringRep, card.face, card.back); }

-1
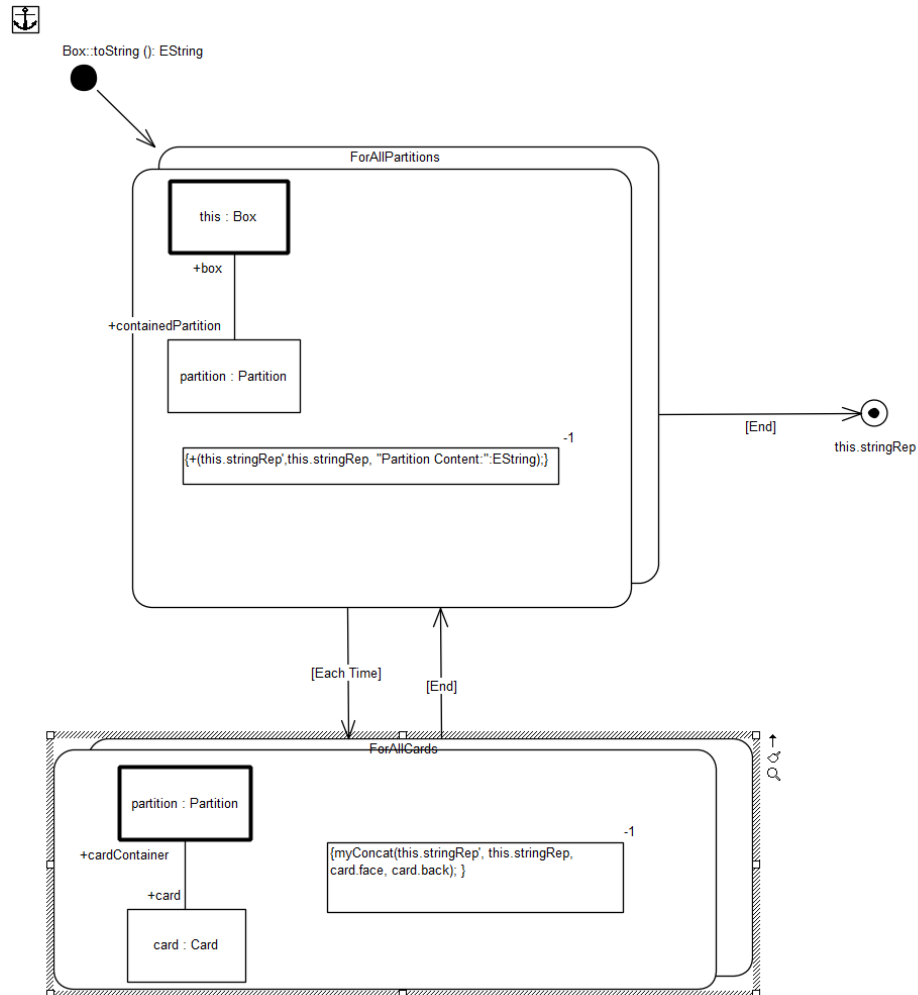
Figure 13.8: Complet SDM for `Box::toString()`

Problems ⊠   Console   Properties   Call Hierarchy

6 errors, 2 warnings, 0 others

Description

⊿ ⊗ Errors (6 items)
  ⊗ Found unknown attribute constraint with signature: "myConcat(EString,EString,EString,EString)".
  ⊗ MALFORMED_ATTRIBUTE_CONSTRAINT_SPECIFICATION: No "OperationSpecifications" in "operationSpecification
  ⊗ MALFORMED_ATTRIBUTE_CONSTRAINT_SPECIFICATION: No "OperationSpecifications" in "operationSpecification
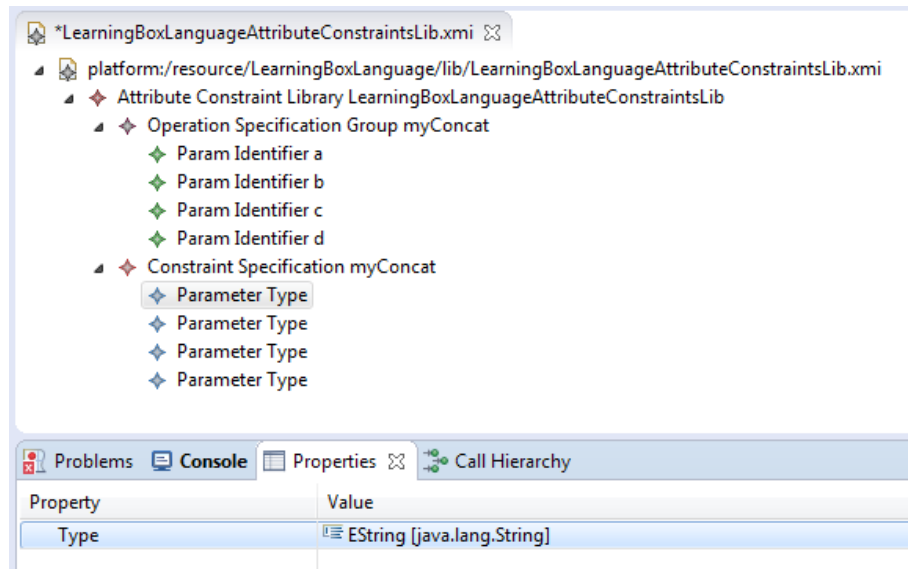
Figure 13.9: Error

Figure 13.10: The content of LearningBoxLanguageAttributeConstraintsLib.xmi
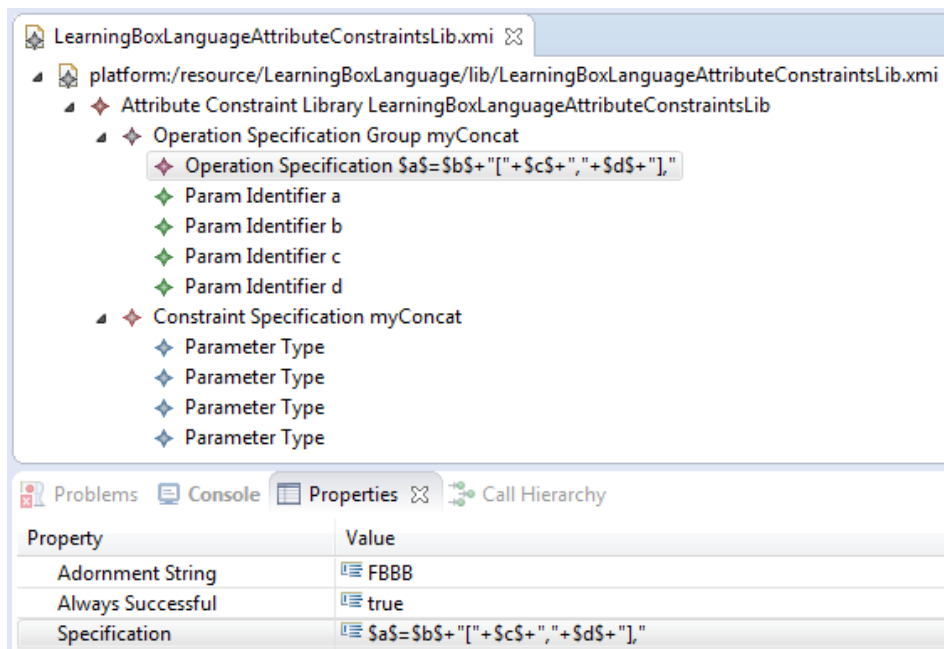
Figure 13.11: The operation specification for myConcat

13.3   Build in Complex Attribute Constraints

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

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

14 Conclusion and next steps

Congratulations – you've reached the end of eMoflon's introduction to uni-directional model transformations! You've learnt that SDMs are declared as *activities*, which consist of *activity nodes*, which are either *story pattern*s 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.