# Story Diagrams – Syntax and Semantics [1] [2] [3]

Technical Report
tr-ri-12-320

Markus von Detten, Christian Heinzemann, Marie Christin Platenius,
Jan Rieke, Julian Suck, and Dietrich Travkin
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
D-33102 Paderborn, Germany
[Markus.von.Detten|Christian.Heinzemann|Marie.Christin.Platenius|
Jan.Rieke|Julian.Suck|Dietrich.Travkin]@uni-paderborn.de

Stephan Hildebrandt
Department System Analysis and Modeling
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3
D-14482 Potsdam, Germany
stephan.hildebrandt@hpi.uni-potsdam.de

Version: 0.1

Paderborn, April 11, 2012

[1]This work was partially developed in the Collaborative Research Center 614, "Self-optimizing Concepts and Structures in Mechanical Engineering" at the University of Paderborn and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft (DFG).

[2]This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (CRC 901).

[3]This work was partially developed in the project "ENTIME: Entwurfstechnik Intelligente Mechatronik" (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the European Union, European Regional Development Fund, "Investing in Your Future".

# Contents

# Chapter 1.

# Introduction

The high complexity of modern technical systems poses great challenges to their development process. Model-based development approaches are a promising means to tackle this complexity. In such a model-based development approach, models are considered to be first-class artifacts of the development. They describe different parts of the system in development from different viewpoints and on different abstraction levels. For instance, models are used to describe the structure and behavior of a software system, improving the overall comprehensibility of the system. These models can then be employed to automatically generate code, reducing the risks of implementation errors.

Even if two models have different viewpoints and are used for different purposes, their information may overlap. Thus, models have to be translated into each other during such a development process. To translate between different models and to keep them consistent, model transformations can be applied. Model transformations are also used to define in which way models can be changed, e.g., to specify refactoring operations.

Furthermore, model transformations themselves can be employed to precisely specify the behavior of a system at run-time. If, for example, a system should react to environment changes by reconfiguration, these reconfigurations can be described by model transformations which define how to change the structure of the system. They furthermore allow a formal analysis, e.g., to prove that certain properties still hold after applying a transformation.

Story diagrams [ZSW99, FNTZ00, Zün01] are a powerful visual formalism for specifying model transformations, based on the well-known concept of graph transformation systems. They feature declarative parts to specify object patterns which are matched and altered in the source model and combine them with ideas from imperative programming to specify the control flow of the transformation execution. The concrete syntax of story diagrams is based on the concrete syntax of UML activity diagrams.

Since their introduction in 1998, story diagrams have been successfully applied in a wide range of application scenarios and are now used for diverse purposes. Furthermore, several extensions to story diagrams have been proposed. For instance, Story Decision Diagrams (SDDs) [GK06] extend story diagrams with features of first-order-logic such as quantification to allow the expression of complex properties like safety requirements. Timed Story Scenario Diagrams (TSSD) [KG07] on the other hand are a story-diagram-based notation for the specification of scenarios, integrating structural and temporal aspects.

Besides, some semantic issues have been identified in the original concept [TMG06]. In addition, the main tool for the specification of story diagrams, the FUJABA tool suite, has undergone major redesigns in the last years; these redesigns also affected the story diagram implementation. Moreover, new approaches like the Story Diagram Interpreter have emerged.

In this technical report, we seek to provide a complete reference to the syntax and semantics of story diagrams. It consolidates previous publications in a single document. We provide definitions for the abstract and the concrete syntax as well as the semantics of story diagrams.

As an example, we show how story diagrams can be used to specify refactoring operations on structural software models like class diagrams.

The following chapter introduces important foundations like graph transformations that are necessary for the understanding of story diagrams. Chapter 3 then describes the concepts used in story patterns by explaining their abstract and concrete syntax as well as the semantics. Chapter 4 gives a complex example by illustrating the specification of a refactoring operation with story diagrams. Related work is discussed in Chapter 5 while Section 6 concludes the main part of the report. Appendix A deals with the execution of story diagrams by interpretation. Finally, Appendix B contains the technical reference that documents the current metamodel for story diagrams in detail.

# Chapter 2.

# Foundations

This chapter introduces the foundations for working with story diagrams. Since story diagrams are based on graphs and corresponding graph transformations, we introduce the basics of graph transformations in Section 2.1. Story diagrams are built on an extension of this simple graph model which is called typed attributed graph transformations (cf. Section 2.2). These are based on so-called type graphs that allow to distinguish different types of nodes and edges. Based on typed attributed graph transformations, we will introduce basic concepts of model transformations in Section 2.3. Section 2.4 presents a type graph which we use in the examples in this document.

## 2.1. Graphs and Graph Transformations

Graphs consist of nodes and edges where an edge always connects two nodes. Nodes are used to represent objects and edges denote relationships between these objects. In the course of this document, we assume edges to be directed, i.e., they have a source node and a target node. In the most simple case, neither nodes nor edges have a predefined semantics [Roz97].

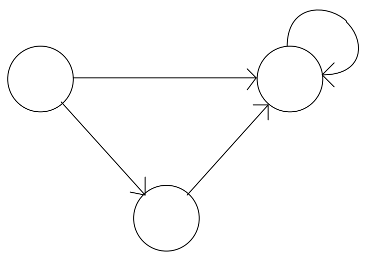


Figure 2.1.: Simple Graph

Figure 2.1 shows an example of a simple graph with three nodes and four directed edges. The nodes are visualized as circles, the edges are visualized as arrows. An edge may have the same node as a source and target node. Such an edge is called a *self-edge*.

*Graph transformation rules* specify allowed modifications of graphs. They consist of a left-hand side (LHS), a right-hand side (RHS), and a so-called rule morphism. Both, the LHS and the RHS are graphs while the rule morphism specifies which nodes of the LHS and RHS are considered to be the same. This information is required for the application of a graph transformation rule to a graph.

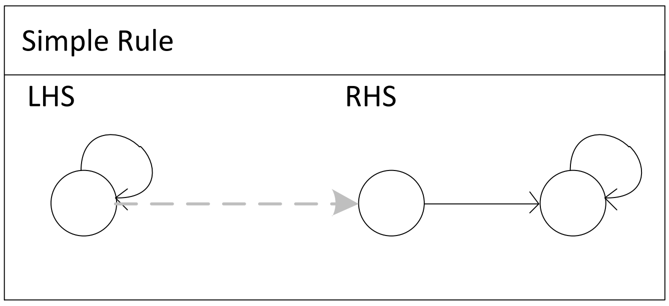


Figure 2.2.: Simple Graph Transformation Rule

Figure 2.2 shows an example of a graph transformation rule. The LHS contains only one node with a self-edge. The RHS contains two nodes connected by an edge where the right node of the RHS has a self-edge as well. The rule morphism is visualized by the gray, dotted arrow. It specifies that the node of the LHS and the left node of the RHS are considered to be the same.

The application of a graph transformation rule to a graph is called a *graph transformation* [EEPT06]. The graph on which the rule is to be applied is called the *host graph*. The application of a graph transformation rule to a graph is performed in three steps. In the first step, an occurrence of the LHS of the graph transformation rule in the host graph is searched. Such an occurrence is called a *match* of the graph transformation rule. If a match has been found, the second step is executed in which all nodes and edges that occur in the LHS but not in the RHS are deleted from the host graph. In this step, the rule morphism is used to decide which nodes do not occur in the RHS. In the third step, all nodes and edges that occur in the RHS but not in the LHS are added to the host graph. After the application of the graph transformation rule, there exists a match of the RHS into the host graph.

Figure 2.3 shows an example of a graph transformation that applies the graph transformation rule of Figure 2.2 to the graph of Figure 2.1. The matching of the LHS into the host graph is visualized by a gray, dotted line. Then, the graph transformation rule deletes the self-edge from this node. Afterwards, a new node with a self-edge is created and connected to the previously matched node by an edge. The match of the RHS into the host graph after the rule application is again shown by gray, dotted lines.

Formally, identifying a matching of a graph transformation rule in a host graph requires to identify a subgraph of the host graph which is isomorphic to the LHS. This is denoted as the *subgraph isomorphism* problem which is known to be *NP-complete* [Epp95].

In the field of algebraic graph transformations, the two most popular approaches for applying a graph transformation rule to a graph are the *double-pushout approach* [Roz97] and the *single-pushout approach* [Roz97]. The definition of story diagrams follows the single-pushout

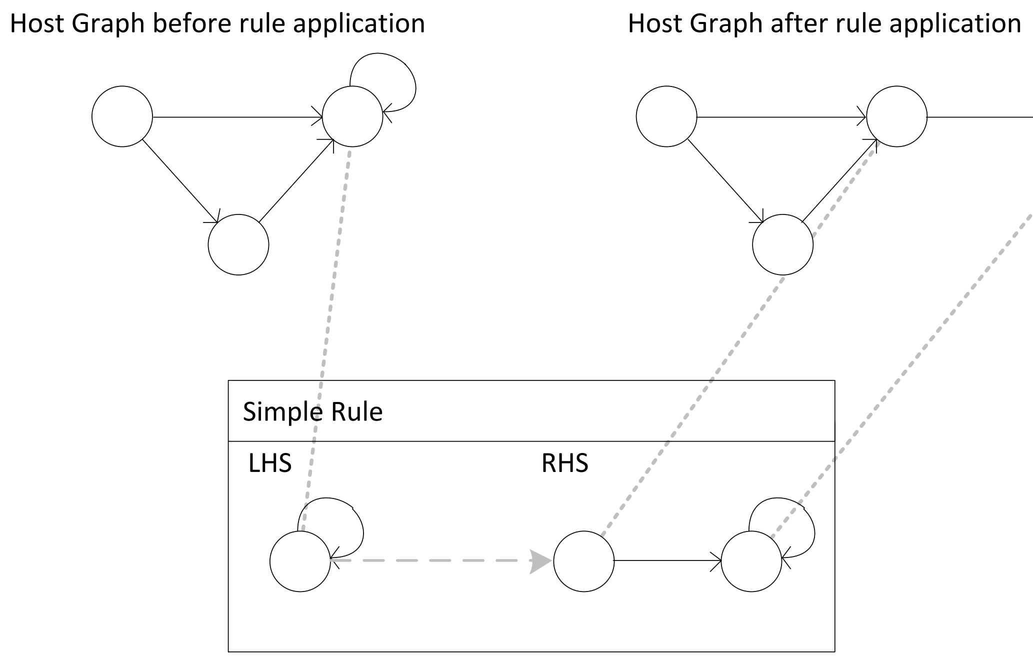

Host Graph before rule application
Host Graph after rule application
Simple Rule
LHS
RHS

Figure 2.3.: Application of a Graph Transformation Rule

approach. Besides the more theoretical differences, the two approaches differ in the handling of two special situations that might occur upon rule application.

The first situation is the following. Assume the left-hand side of a rule consists of two nodes. The first node is to be deleted and the second one is to be preserved. Both of these nodes may be matched to the same node in the host graph. In this situation, it is not clear if the node in the host graph is to be deleted or preserved. The double-pushout approach explicitly forbids the application of the rule in such situations. The single-pushout approach allows such situations and gives deletion priority over preservation.

The second situation deals with dangling edges. It occurs if a certain node is to be deleted but some of its connected edges are to be preserved. The transformation would lead to a non-valid graph in which the edges would no longer have either a source or a target node. The double pushout approach does not allow such situations and instead requires that connected edges are explicitly deleted. The single-pushout approach allows such situations and implicitly deletes edges if one of their source or target nodes is deleted.

In general, matches of graph transformation rules are homomorphisms of the LHS of the rule to the host graph. That allows to match two nodes of the LHS to the same node of the host graph leading to the first situation mentioned above. Such situations may be prevented by using isomorphisms for matching the LHS. Then, each node of the LHS must be matched to a unique node of the host graph. Thus, using isomorphic matchings prevents the first situation when using single-pushouts.

In addition to LHS and RHS a graph transformation rule may specify negative application conditions (NAC, [Roz97]). A NAC is an additional condition for a successful match. If the subgraph specified by the NAC can be matched into the host graph, then the graph transformation rule must not be applied.

## 2.2. Typed Attributed Graph Transformations

Graphs and according graph transformations as introduced in Section 2.1 are a very basic approach for describing structures and their modification. When using graph transformations for modeling behavior for object-oriented software or as a foundation for defining the semantics of modeling languages, it is necessary to distinguish different types of nodes and edges in a graph in order to give them semantics.

Therefore, *typed attributed graph transformations* [EEPT06] have been defined. Typed attributed graph transformations introduce a *type graph* and node attributes. The type graph defines different types of nodes and edges and it defines which types of edges are allowed to be used in combination with which types of nodes. Additionally, nodes may carry attributes like, e.g., objects in an object-oriented programming language. Moreover, the type graph specifies inheritance relations between types of objects, a concept that is also known from object-oriented programming languages. Thereby, a type graph specifies the structure of all possible graphs.

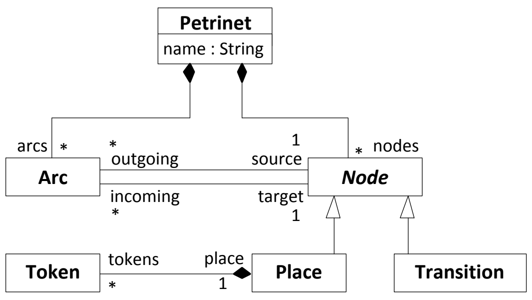


Figure 2.4.: An Exemplary Type Graph – Metamodel for Petrinets

As an example, Figure 2.4 shows a simple type graph for petrinets. A Petrinet consists of Nodes and Arcs. An Arc connects two nodes, which it refers as source and target. As usual, there exist two types of nodes: Places and Transitions, where Places may contain a number of Tokens.

The type graphs can be created, e.g., by using an arbitrary metamodeling language. Examples include Ecore [SBPM08], MOF [Obj11a], and UML [Obj10b].

The modifications of a typed attributed graph are specified by typed attributed graph transformation rules. In these rules, all nodes and edges are typed over the type graph. The matching needs to respect these types.

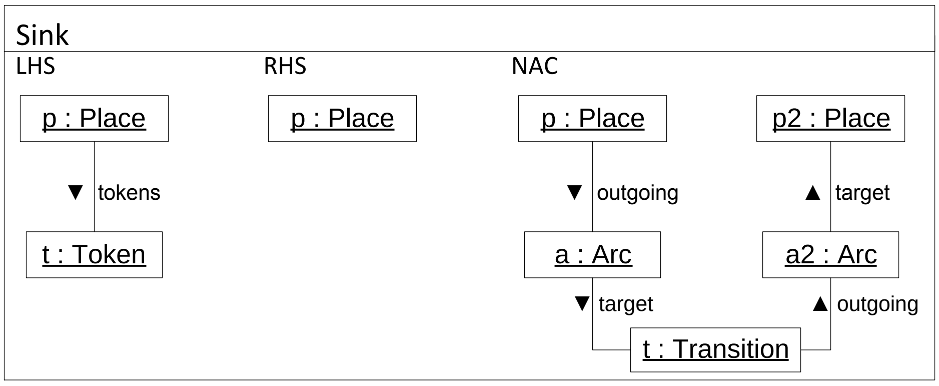


Figure 2.5.: Typed Attributed Graph Transformation Rule

Figure 2.5 shows an example of a typed attributed graph transformation rule that specifies the behavior for a sink using a concrete syntax as used by Ehrig et al. [EEPT06]. In a petrinet, a sink is a transition that has no outgoing place. Then, the transition consumes tokens without producing new tokens. The rule matches a Place with a Token in its LHS. Upon application, the Token is deleted. The rule, however, may only be applied if the place p is connected to a sink, which is ensured by the NAC. The NAC specifies a subgraph where p is connected to a Transition that has an outgoing Arc to a Place. If such subgraph can be matched, the rule is not applicable. The rule morphisms is indirectly specified by the names of the nodes, as proposed by Ehrig et al. [EEPT06].

## 2.3. Model Transformations

A model transformation modifies or translates different kinds of models. We use graph transformations to specify model transformations formally. In the context of model transformations, the host graph of a graph transformation is called *instance model*[1] (or simply model). Instance models are specified in a certain language, often a domain-specific language (DSL)[2]. Thus, the type graph represents the *abstract syntax* of the language used and is part of the corresponding *metamodel* [Küh06] for this language. By replacing the metamodel we are able to specify transformations for models specified in various languages.

The type graph or the abstract syntax of the language used to describe the instance models is, in our case, described by a set of classes and their relations which define all potential instance models. These classes and relations constitute a so-called *type model* (the actual type graph). An instance model always contains *objects* and *links* (nodes and edges) that are instances of the classes and relations defined in the corresponding type model.

If a graph transformation transforms a model based on a given type graph into a model based on the same type graph (modification of the model), the transformation is called *en-*

[1]Thomas Kühne calls it *token model* [Küh06].
[2]The language can be textual or graphical.

*dogenous* (also known as *in-place* transformation). Otherwise, i.e., the transformation transforms a model based on a given type graph into a model based on another type graph (translation), the transformation is called *exogenous* (also known as *model-to-model* transformation). In principle, story diagrams are endogenous graph transformations. By combining the type graphs of the source model and the target model, story diagrams may also be used to specify exogenous graph transformations.

## 2.4. The Type Graph in The Running Example

The type graph used in the examples in this report describes the structure of an abstract syntax tree for programming languages. In particular, it is an updated and slightly simplified version of the generalized abstract syntax tree (GAST) metamodel developed in the QBench project [QBe06]. The GAST was developed to provide a unified syntax tree model for different programming languages like Java, C, and C++. Figure 2.6 shows an excerpt of that metamodel. Some specialized subclasses have been omitted for clarity reasons.

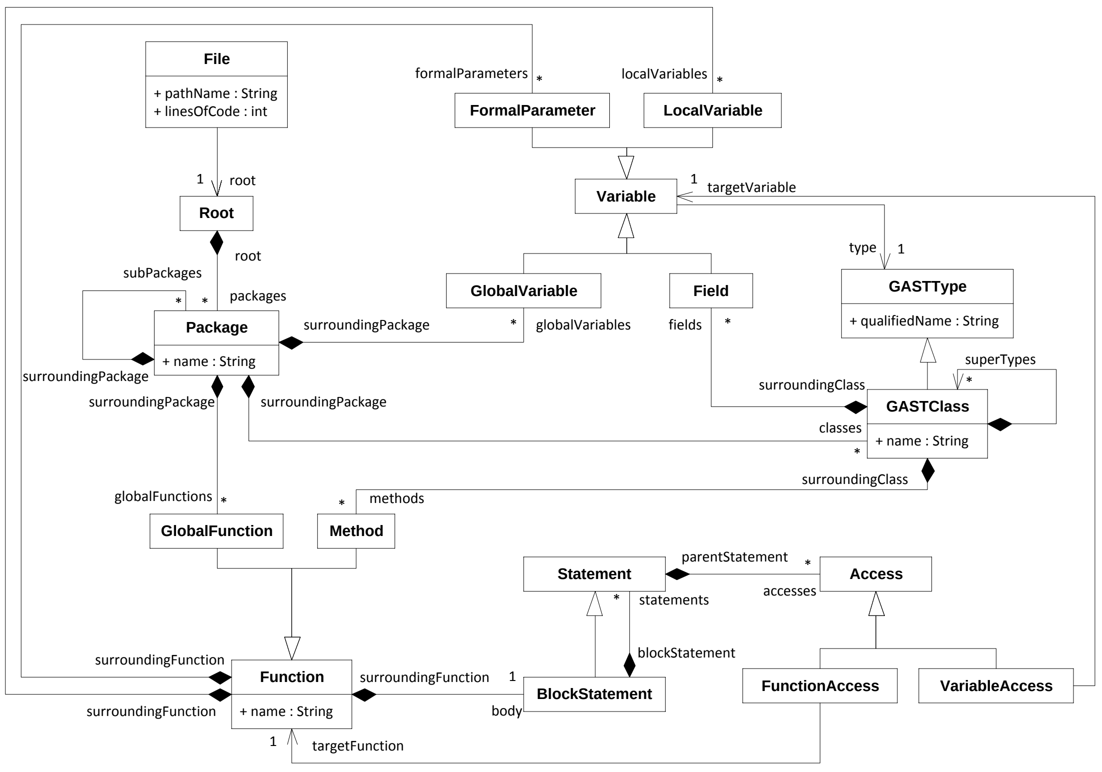


Figure 2.6.: Type Graph of a Generalized Abstract Syntax Tree (GAST) for Object-Oriented Programming Languages [QBe06]

The following description of the classes in the type graph is based on [Tra11].

**Root** The Root element is the central element of every GAST model. All other elements are reachable from the Root node via composition relations.

**File** Elements of the GAST, e.g., classes and packages, can be assigned to files in the file system. A File element holds references to those classes and packages and a String containing the path to the file.

**Package** Similar to packages in Java, the Package element provides name spaces and visibilities. A Package element can contain other packages, classes, global variables, and functions.

**GASTType** The GASTType element represents data types like primitive data types and classes. The attribute qualifiedName contains the unique, fully qualified name of the type.

**GASTClass** Classes are represented by the element GASTClass in the GAST and are a sub type of the GASTType. A GASTClass holds references to its methods, attributes, and inner classes. A GASTClass can be assigned to a Package.

**Function** Function is the super type for all executable operations. In addition to a name attribute, a Function can have a number of local variables and formal parameters. The return type of a Function is determined by its DeclarationTypeAccess, a sub class of Access. A Function always contains a block statement which, in turn, can contain other statements.

**GlobalFunction** A GlobalFunction element represents a globally accessible operation, i.e., an operation that does not belong to a class. It can be assigned to a name space defined by a package. For example, C functions are represented by GlobalFunctions.

**Method** Functions that belong to a class are represented by Method elements, a sub type of Function.

**Variable** Variable is a super type for all kinds of variables. A Variable always has a name and a type.

**LocalVariable** LocalVariables are variables that are contained in a Function.

**FormalParameter** FormalParameters are variables that represent the parameters of a Function.

**GlobalVariable** GlobalVariables are variables that are globally accessible within a given scope. The scope is determined by the package in which the GlobalVariable is contained.

**Field** The Field element represents class variables. Therefore it is contained in a GASTClass.

**Statement** A Function consists of a number of Statements. There are multiple sub classes of Statement which represent the different kinds of statements. Most of them are omitted here. A Statement can contain a number of Accesses.

**BlockStatement** The BlockStatement is a special kind of statement which can contain other Statements. It is the root element of all Statements contained within a Function.

**Access** An Access represents the use of a Variable or a Function. It always belongs to a certain Statement.

**FunctionAccess** A FunctionAccess represents the use of a Function in a Statement and therefore references the accessed Funtion element.

**VariableAccess** A VariableAccess represents the use of a Variable in a Statement and therefore references the accessed Variable element.

# Chapter 3.
# Concepts

This chapter presents the concepts used in story diagrams. It begins with a short presentation of the general ideas of story diagrams and story patterns. Section 3.2 then proceeds to explain story patterns in detail. It is followed by Section 3.3 which describes how story patterns are used in story diagrams and also presents calls between different story diagrams. Section 3.4 briefly covers the expressions that can be used in story diagrams.

## 3.1. Story Diagrams and Story Patterns in a Nutshell

In model-driven software development, a software model is the key artifact of the development. It describes the software's structure as well as its behavior and can be translated into executable source code or be interpreted to be executed. The UML offers notations for the description of the software structure and behavior, besides others class diagrams and activity diagrams. However, since UML activity diagrams use natural language in the activity nodes to describe the particular activities, they are not automatically executable. Thus, a formal behavioral specification is needed. For that purpose, *story diagrams* have been developed [FNTZ00, Zün01]. They are based on UML 1.5 activity diagrams [Obj03] and replace the natural language with a formal language to specify behavior and, thus, can be automatically executed.

A story diagram specifies a *model transformation*. In terms of the classification of model transformation languages proposed by Czarnecki and Helsen [CH06], story diagrams are an endogenous, in-place transformation language (see also Section 2.2).

Story diagrams describe the control flow similar to UML activity diagrams by means of activity nodes and activity edges. The behavior of each activity node is described using a graph transformation language called *story patterns*. Each activity node embeds one story pattern. A story pattern uses a graphical notation to specify modifications of object structures in object-oriented software systems. The modifications are basically creations and removals of objects and their interconnections (links).

Using a simple and familiar notation, story patterns are similar to UML object diagrams (see the embedded story pattern in Figure 3.2). A story pattern represents an object structure that is to be modified. It includes annotations specifying which objects and links are to be removed and created. Story patterns are a declarative language since they only specify what to remove and create but not how to do it and in which order. This way, the complexity of the behavioral specifications is reduced. In contrast to the deterministic control flow specified by activity nodes and edges which determine the order of story pattern executions, the order of creations and deletions specified by a story pattern is non-deterministic.

Story patterns are based on the well-known formalism of *graph transformation systems* and the corresponding theory [Roz97]. Thus, precise analyses of the operations described by story patterns are possible, e.g., it can be checked if certain properties of the object structure to be modified remain after the structure's modification [Sch06, Mey09].

A story pattern specifies a *graph transformation rule* [Roz97] (see Section 2.1). Given a so-called *host graph*, i.e. the graph to be modified, a graph transformation removes and creates nodes and edges in the given host graph. The host graph is a typed attributed graph where the graph specified in the graph transformation is searched for (*graph matching*) and modified afterwards (see Chapter 2 for more details).

In case of story patterns, the host graph is the object structure or model to be modified, i.e. the run-time data of the executed software. Thus, we call the host graph's nodes and edges *objects* and *links* while the host graph itself is called *instance model* (or simply model) in the remainder of the report. The type graph is a set of classes and their relations which define all potential instance models at run-time. These classes and relations constitute a so-called *type model* (see Section 2.3). Furthermore, we call the nodes and edges in story patterns *object variables* and *link variables* since these represent and are matched to objects and links in the instance model. The types of these variables are determined by types in a type model which is a prerequisite to story patterns.

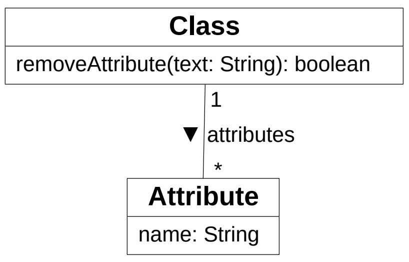


Figure 3.1.: Exemplary Type Model

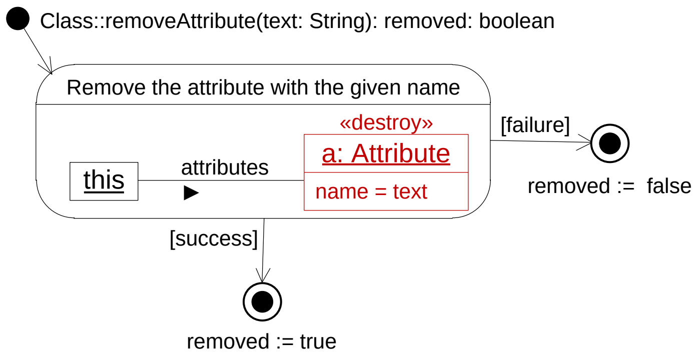


Figure 3.2.: Exemplary Story Diagram

For example, the class diagram in Figure 3.1 defines the types Class and Attribute as well as their relations, attributes, and operations. The corresponding story diagram in Figure 3.2 defines the behavior of the removeAttribute method defined in the class diagram. Here, the story diagram specifies that a class's attribute with the name given by the parameter text is to be found in the instance model and in case of success this attribute is to be removed («`destroy`»).

Graph matching includes the *subgraph isomorphism* problem which is known to be *NP-complete* [Epp95]. To reduce the problem in the average case, the implemented graph matching approach for story patterns uses a subgraph isomorphism for at least one node as input, i.e. at least one node in the left-hand side of a graph transformation rule (object variable in a story pattern) is already matched to a certain node in the host graph (object in an instance model).

In summary, a story diagram is a special, formally defined UML activity diagram that embeds graph transformation rules, so-called story patterns, in its activity nodes to precisely describe run-time behavior by means of graph transformations.

## 3.2. Story Patterns

In this section, we introduce story patterns in more detail. We start by giving the general idea of story patterns in Section 3.2.1. Thereafter, we describe the basic concepts of story patterns, namely object variables, link variables, and their respective binding semantics in Sections 3.2.2 to 3.2.4. Finally, we show the use of object attributes in a story pattern in Section 3.2.5.

### 3.2.1. General Idea

Story patterns are typed attributed graph transformation rules with inheritance on object types (cf. Section 2.2) that can be embedded into an activity node of a story diagram (cf. Section 3.3). By using a type model as introduced in Section 2.2, story patterns enable polymorphism for matching object and link variables. This allows for specifying graph replacement rules for object-oriented models.

Object and link variables are matched to the objects and links of the instance model. In contrast to typed attributed graph transformations, story patterns explicitly require to use isomorphic matchings, i.e., two object variables of a story pattern may not be matched to the same object of the instance model.

For enabling a concise notation of the graph transformation, story patterns apply a short-hand notation depicting the left-hand side (LHS) and the right-hand side (RHS) in a single, annotated graph using stereotypes. In the short-hand notation, we use binding operators for defining the LHS and the RHS. Object and link variables representing objects and links not to be changed by the story pattern carry no stereotype. Object and link variables representing objects to be created (or deleted) are annotated with «`create`» (or «`destroy`», respectively). Consequently, the LHS consists of all object and link variables that carry no stereotype or stereotype «`destroy`». The RHS consists of all object and link variables that carry no stereotype or the stereotype «`create`». The deletion of objects and links follows the single-pushout approach (cf. Section 2.1).

Figure 3.3 shows an example of a single story pattern that redirects a method call from an old method to a new method. In the example, the object variables parentClass, oldMethod, and newMethod are bound variables, i.e., they already refer to objects of the instance model (cf. Section 3.2.4.1). The object variables anyMethod and c are unbound. When applying the

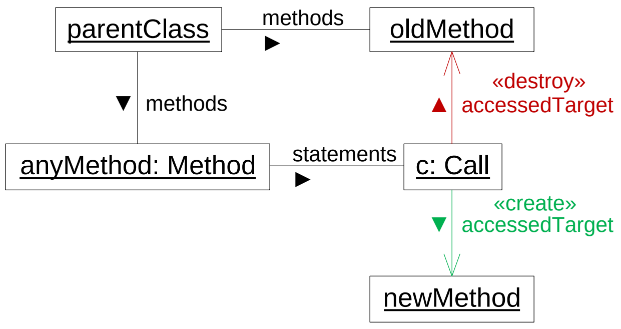


Figure 3.3.: Example of a Story Pattern

story pattern, first a match for anyMethod and c is searched in the instance graph. A possible match will be any method in parentClass which contains a call to oldMethod. If the matching is successful, the link from c to oldMethod will be deleted and the link from c to newMethod will be created.

In the concrete syntax of story patterns, the object and link variables representing objects and links not to be modified by the story pattern are visualized in black. Object and link variables representing objects and links to be destroyed are annotated with «destroy» and visualized in red. Object and link variables representing objects and links to be created are annotated with «create» and visualized in green. An unbound object variable is labeled with its name and the name of the corresponding type. For bound object variables, we omit the name of the type (e.g. parentClass in Figure 3.3).

In general, the matching process is executed as a three step process: first, a matching is searched which uses the bound variables of the story pattern as a starting point. The matching associates objects and links of the instance model to all object and link variables of the story pattern. The matching is performed as defined for typed attributed graph transformations and considers all object and link variables of the LHS. If a matching can be obtained, the story pattern is applicable and the execution proceeds. Otherwise the execution of the story pattern is aborted. In the second step, all objects and links matched to object and link variables annotated with «destroy» are deleted. Finally, objects and links are created in the instance model for all variables annotated with «create».

Story patterns aim to reduce the computational complexity of the matching process (cf. Section 2.1) by using bound variables. We require at least one bound object variable in each story pattern which is used as a starting point for the matching process.

## 3.2.2. Objects and Object Variables

Object variables in a story pattern represent the objects in an instance model to be matched. The variables are uniquely identified by their name. The objects are instances of classes of the underlying type model (cf. Section 2.2). Thus, the object variables are typed by classes from this model.

The story pattern in Figure 3.3 contains five object variables with the names parentClass, oldMethod, anyMethod, c and newMethod. The type of an object variable is only visualized if the variable is unbound or maybe bound (cf. Section 3.2.4.1). For example, the object variable anyMethod has the type Method.

Object variables have binding states, binding operators and binding semantics which are described in Section 3.2.4.

### 3.2.3. Links and Link Variables

Link variables represent connections between objects and are used to connect different object variables. A link variable is typed over a reference of the underlying type graph.

Like object variables, link variables also have binding operators and binding semantics (cf. Section 3.2.4), but no binding state.

### 3.2.4. Binding of Variables

Object variables have binding states (unbound, bound, maybe bound), binding semantics (mandatory, negative, optional), and binding operators (check only, create, destroy). Link variables have binding operators and binding semantics. Their meaning is described in the following.

#### 3.2.4.1. Binding States

An object variable or a link variable can be declared as *bound*, *unbound*, or *maybe bound* (i.e. it is unknown if the variable is bound or not). This is defined by its binding state. An unbound variable is matched during the execution of the containing story pattern. A bound variable must have been matched previously. For a variable that is specified as maybe bound, a new match will only be determined if it has not been bound before. Otherwise it will be treated as a bound variable. This is useful, if the same pattern should be used in different contexts, i.e., the bound variable of the pattern differs depending on the context but otherwise the patterns are identical. Without maybe bound variables, different patterns would have to be modeled that only differ in which variable is the bound variable of the pattern. With maybe bound variables, all variables can be set to maybe bound and the caller specifies a binding for one of them depending on the context.

Unbound object variables are visualized with an underlined label of the form “name: Type” (cf. Figure 3.4 a)). For bound object variables the type is hidden, as depicted in Figure 3.4 b). Maybe bound object variables are represented like unbound object variables, but are marked by a question mark after the name (cf. Figure 3.4 c)).

In a valid story pattern, each connected component[1] must contain at least one bound object variable or created variables only. This is necessary to avoid a search over the whole underlying instance model which requires a long runtime in most cases (cf. Section 3.2.1).

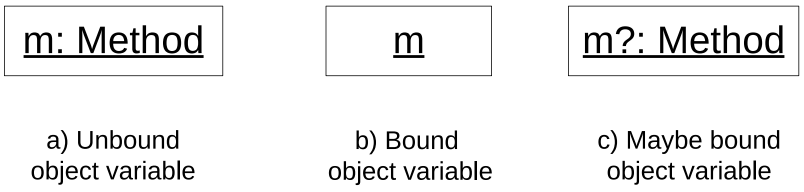


Figure 3.4.: Binding States for Object Variables

#### 3.2.4.2. Binding Semantics

Object variables and link variables have binding semantics that determine if a variable is mandatory, negative or optional. A match for mandatory variables must exist in the given instance model, otherwise the pattern matching fails. In contrast, negative variables constitute so-called negative application conditions (NACs) and must not exist in the instance model. If a variable defined as negative can be matched during the execution of the story pattern, the pattern matching fails. Matches for optional variables may exist. An optional variable will be bound if possible, but the story pattern may also be matched successfully otherwise.

Negative object variables are visualized crossed-out (cf. Figure 3.5 b)) and optional object variables are visualized with a dashed border (cf. Figure 3.5 c)). The same holds for negative and optional link variables (cf. Figure 3.5 e) and f)).

Negative as well as optional object and link variables are not part of a connected component. This means, the graph has to be still connected when ignoring optional and negative parts. However, optional and negative object variables must be reachable from a connected component. Consequently, regarding the rule that each connected component must contain at least one bound object variable (cf. Section 3.2.4.1), there are situations in which the use of negative or optional object variables is not allowed. Figure 3.6 shows these situations. Case a) is allowed but case b) is not because, in the latter case, the graph without the negative and optional elements is not a connected component anymore. Case c) is allowed because the object variables a and c are bound which means that each connected component has at least one bound object variable. Accordingly, case d) is allowed, too, because a and b are both bound. Case e) is not allowed while Case f) is. Case g) is not allowed because the semantics is the same as in Case a) due to the single-pushout approach of story patterns.

[1] With "connected component" we mean a subgraph in which each object variable is reachable from at least one bound object variable via directed link variables.

Similar to the application of negative object variables, Figure 3.7 shows some examples for the application of optional object variables. While case a) is allowed, case b) is not allowed because in this case the shown graph is not connected anymore. However, case c) and d) are allowed because each connected component contains at least one bound object variable. Cases e) and f) are also allowed.

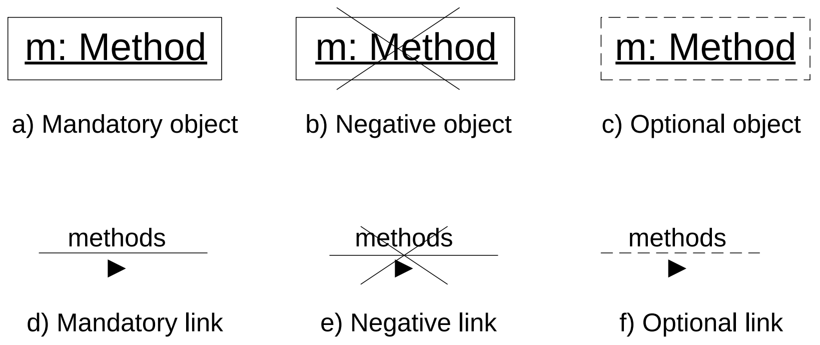


Figure 3.5.: Binding Semantics for Object and Link Variables

### 3.2.4.3. Binding Operators

Binding operators define whether an object or link is to be created, deleted, or just matched in the instance model. After all elements that are defined to be deleted or just matched have been matched, the model is modified by deleting and creating the elements as defined (see Section 3.2.1).

Objects and links to be created are marked with the stereotype «`create`» (cf. Figure 3.8 b) and e)) and objects and links to be deleted are marked with the stereotype «`destroy`» (cf. Figure 3.8 c) and f)).

Since no objects and links exist for variables marked with «`create`», they also do not belong to a connected component (like negative or optional variables).

### 3.2.4.4. Feasible Binding Combinations

Binding states, binding semantics and binding operators can be arbitrarily combined, but only certain combinations are feasible. Table 3.1 lists all feasible binding combinations for object variables. As shown there, bound and maybe bound object variables must not have negative or optional binding semantics. As well, the combination of the binding states bound or maybe bound and the binding operator create is not allowed.

The feasible combinations of binding semantics and binding operators for link variables are given in Table 3.2. Link variables have no binding state.

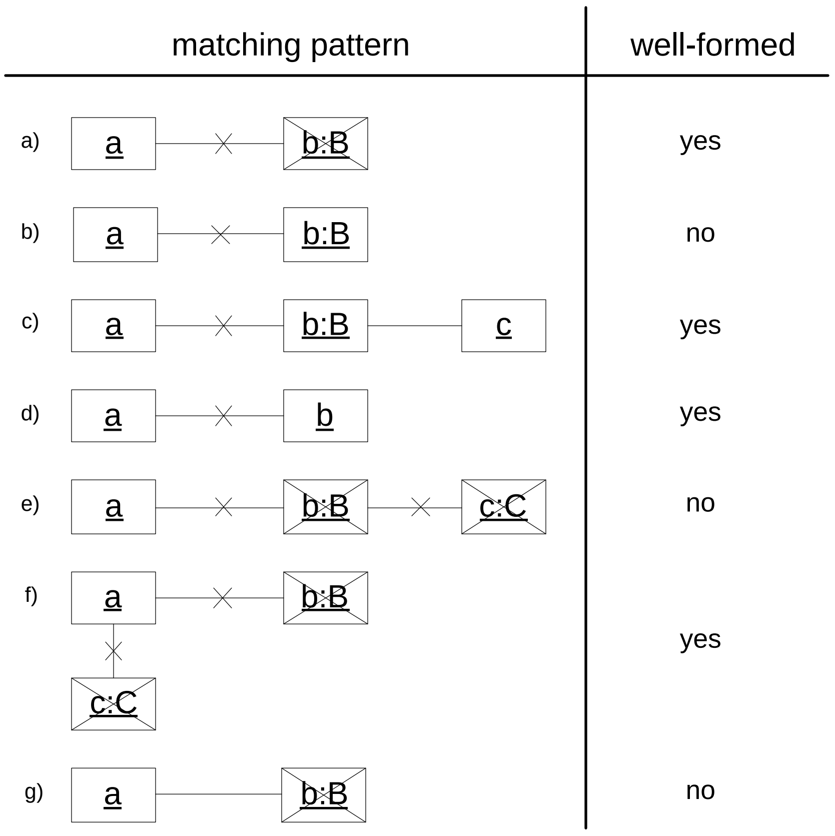


Figure 3.6.: Negative Application Conditions

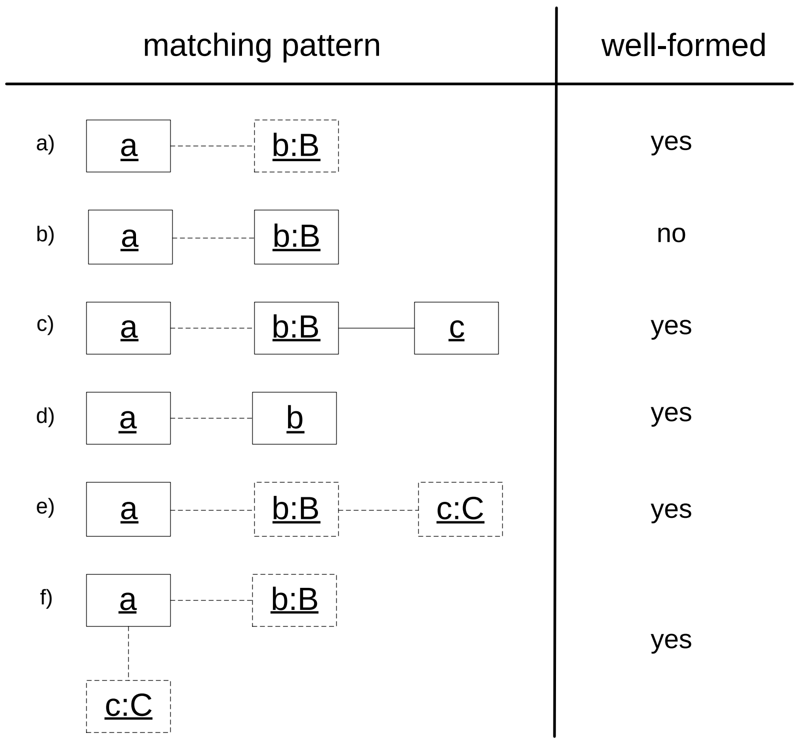


Figure 3.7.: Optional Object and Link Variables

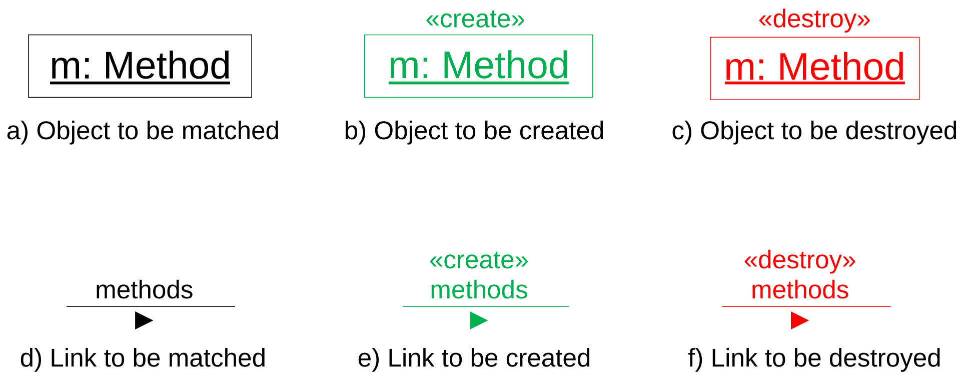


Figure 3.8.: Binding Operators for Object and Link Variables

Table 3.1.: Feasible Combinations of Binding States, Binding Semantics, and Binding Operators for Object Variables

| Binding State | Binding Semantics | Binding Operator | Feasible |
|---:|---:|---:|---:|
| UNBOUND | MANDATORY | CHECK_ONLY | yes |
| UNBOUND | MANDATORY | CREATE | yes |
| UNBOUND | MANDATORY | DESTROY | yes |
| UNBOUND | NEGATIVE | CHECK_ONLY | yes |
| UNBOUND | NEGATIVE | CREATE | no |
| UNBOUND | NEGATIVE | DESTROY | no |
| UNBOUND | OPTIONAL | CHECK_ONLY | yes |
| UNBOUND | OPTIONAL | CREATE | yes |
| UNBOUND | OPTIONAL | DESTROY | yes |
| BOUND | MANDATORY | CHECK_ONLY | yes |
| BOUND | MANDATORY | CREATE | no |
| BOUND | MANDATORY | DESTROY | yes |
| BOUND | NEGATIVE | CHECK_ONLY | no |
| BOUND | NEGATIVE | CREATE | no |
| BOUND | NEGATIVE | DESTROY | no |
| BOUND | OPTIONAL | CHECK_ONLY | no |
| BOUND | OPTIONAL | CREATE | no |
| BOUND | OPTIONAL | DESTROY | no |
| MAYBE_BOUND | MANDATORY | CHECK_ONLY | yes |
| MAYBE_BOUND | MANDATORY | CREATE | no |
| MAYBE_BOUND | MANDATORY | DESTROY | yes |
| MAYBE_BOUND | NEGATIVE | CHECK_ONLY | no |
| MAYBE_BOUND | NEGATIVE | CREATE | no |
| MAYBE_BOUND | NEGATIVE | DESTROY | no |
| MAYBE_BOUND | OPTIONAL | CHECK_ONLY | no |
| MAYBE_BOUND | OPTIONAL | CREATE | no |
| MAYBE_BOUND | OPTIONAL | DESTROY | no |

Table 3.2.: Feasible Combinations of Binding Semantics and Binding Operators for Link Variables

| Binding Semantics | Binding Operator | Feasible |
|---|---|---|
| MANDATORY | CHECK_ONLY | yes |
| MANDATORY | CREATE | yes |
| MANDATORY | DESTROY | yes |
| NEGATIVE | CHECK_ONLY | yes |
| NEGATIVE | CREATE | no |
| NEGATIVE | DESTROY | no |
| OPTIONAL | CHECK_ONLY | yes |
| OPTIONAL | CREATE | yes |
| OPTIONAL | DESTROY | yes |

### 3.2.5. Using Object Attributes

The objects of our instance model carry attributes. During the application of a story pattern, these attributes can be used twofold. First, attribute constraints can be specified to restrict the attribute values to a certain range, thereby restricting the possible matches of a story pattern. Second, attribute values can be changed during the graph rewriting step after a successful matching.

We use *attribute constraints* to restrict the matching of object variables to objects of the instance model that have specific attribute values. Thus, attribute constraints are considered to be part of the LHS and do not change the instance model. Figure 3.9 shows an example.

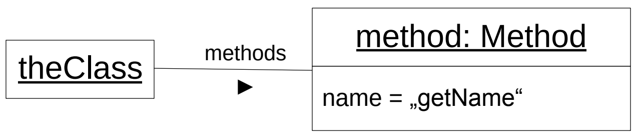


Figure 3.9.: Matching Pattern with an Attribute Constraint

In the example, we match a method being contained in the class represented by the object variable theClass. The match is restricted to a method which has the name "getName".

The values of attributes that are not restricted by an attribute constraint are not considered during the matching. Thus, they may have an arbitrary value. In the current version of story patterns, attribute constraints need to be specified using OCL [Obj10a]. Besides equality checks, all comparative operations on the attributes of an object supported by OCL can be used as object constraints.

Beside attribute constraints, *attribute assignments* can be used to change the value of an attribute during the application of a story pattern. Thus, attribute assignments are considered to be part of the RHS. When using attribute assignments, the value of the attribute is not considered while matching the LHS to the instance model. Figure 3.10 shows an example.

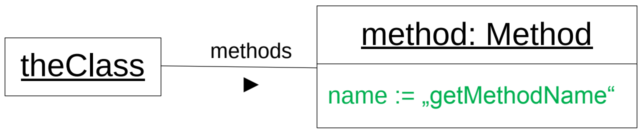


Figure 3.10.: Using an Attribute Assignment

In the example, the story pattern matches a method with an arbitrary name in the class theClass. Then, the name of the method is changed to *"getMethodName"*.

The concrete syntax of an attribute assignment is

```
<attributeAssignment> ::= #Attribute.name ':=' Expression
```

The expression is to be specified using OCL. The type of the return value of the OCL expression must be assignable to the type of the attribute. Since the attribute value is changed as part of the RHS, the assignment is visualized in green color.

The OCL statements we allow for attribute constraints and attribute assignments must not traverse the references of the object variables. Both may only use the attributes of object variables in the same story pattern and arbitrary arithmetic, comparing, and logical operations on them.

## 3.3. Story Diagrams

After explaining the concept of story patterns in Section 3.2, a prerequisite for this section, we explain the story diagrams themselves. We give an overview of the general idea in Section 3.3.1 and go on with explaining the language constructs in the following sections.

### 3.3.1. General Idea

The main idea behind story diagrams is to formalize UML activity diagrams to better support model-driven software development. This is done by not only modeling the software structure, but also completely modeling its behavior and, thus, making the software model executable. For that purpose, graph transformations were chosen to formally specify behavior and have been combined with UML activity diagrams. The result, story diagrams, is a mixture of two languages: an imperative, deterministic language for the description of control flow, namely UML activity diagrams, and a declarative, non-deterministic, object-oriented, graph-transformation-based language for the description of model modifications, so-called story patterns (see Section 3.2). Both languages are graphical, formally defined, and use a familiar notation based on UML 1.5 activity diagrams[2] and UML object diagrams with minor modifications.

An exemplary story diagram is illustrated in Figure 3.11. This story diagram takes a graph-based representation (a model of a so-called abstract syntax graph, see Section 2.4) of an object-oriented program, e.g., in Java, and replaces calls of a given method (oldMethod) with calls of another given method (newMethod).

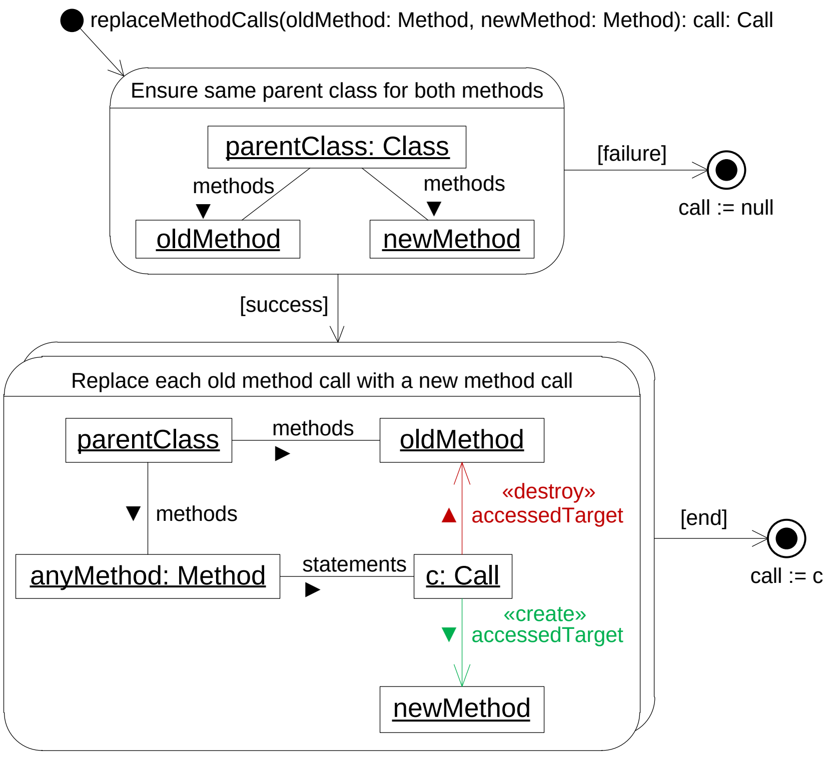


Figure 3.11.: Exemplary Story Diagram – Replace Method Calls

Like UML activity diagrams, story diagrams model control flow by means of activity nodes and activity edges. Each activity node embeds a story pattern to formally specify the behavior for this node[3]. The activity edges can carry guards. These are either specified by boolean expressions, e.g., checking attribute values of a matched object, or by keywords used to specify decisions on whether a story pattern could be matched or not[4]. In Figure 3.11, the used guards are [success] (successful execution of a story pattern), [failure] (failed to completely execute a story pattern), and [end] (activity edge points to the first activity node to be executed after a loop).

[2]Actually, we use the notation of UML 1.5 activity diagrams, but already use the terminology of the UML 2.
[3]There are some exceptions like activity call nodes which do not contain story patterns to specify the behavior.
[4]A story pattern is successfully matched if for each object and link variable in the pattern corresponding objects and links are found in the instance model (host graph) and all specified constraints are satisfied.

In contrast to ordinary UML activity diagrams, story diagrams, so far, do not model concurrent execution. Thus, the language constructs *fork* and *join* are currently not supported in story diagrams. We plan to include these concepts in future versions of story diagrams.

Basically, there are two different ways of using story diagrams in a model-driven software development process.

Originally, story diagrams were used in object-oriented software development to formally specify the behavior of methods that are defined in classes. Calling such a method means to execute the story diagram that represents the method's behavior. If there is a story diagram that models the behavior for each method specified in a class model, the software model completely covers the software's structure and behavior and, thus, can be analyzed and executed. In this case, story diagrams specify the behavior of objects whose properties are defined by classes. For that reason, those story diagrams have a *this* variable – similar to the keyword *this* in Java – representing the object (a class instance) that they belong to (a self reference). This variable can be used as a starting point for the graph matching specified in a story diagram.

For example, the class diagram in Figure 3.12 defines a method findAttribute for all Class objects. This method's behavior is specified by the story diagram in Figure 3.13. The matching of the object structure specified in the contained story pattern, in this case, starts with the this object variable of the type Class which is already bound to the Class object that the findAttribute method belongs to. Thus, this method tries to find an attribute a in the same class with the name given by the method's parameter text.

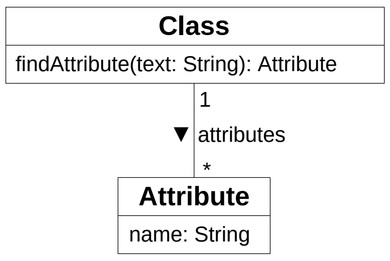
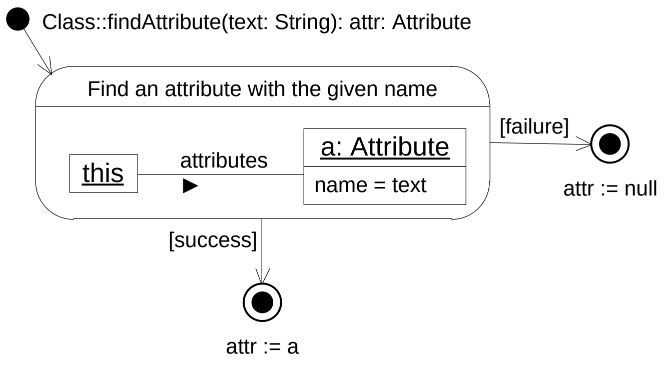


Figure 3.12.: Type Model for the Story Diagram in Figure 3.13



Class::findAttribute(text: String): attr: Attribute
Find an attribute with the given name
this
attributes
▶
a: Attribute
name = text
[failure]
attr := null
[success]
attr := a

Figure 3.13.: Exemplary Story Diagram With *this* Object Variable

Another more flexible way of using story diagrams is to specify any kind of model transformation or operation in a story diagram without attaching this behavior to a certain class. In contrast to the previous case, there is no *this* variable that could be used as a starting point for the graph matching. All starting nodes for the graph matching have to be provided as arguments of the story diagram call. For this purpose, in contrast to the story diagram in Figure 3.13, the story diagram in Figure 3.14 has an additional parameter c. The corresponding arguments of a story diagram call are assumed to be known (bound object variables) and are

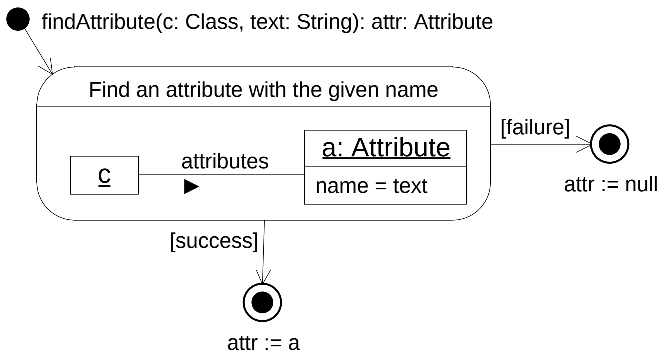

findAttribute(c: Class, text: String): attr: Attribute
Find an attribute with the given name
c
attributes
▶
a: Attribute
name = text
[failure]
attr := null
[success]
attr := a

Figure 3.14.: Exemplary Story Diagram Without *this* Object Variable

used as starting points for the graph matching. This way, the operations or transformations defined by story diagrams can be used from within any other part of the developed software, like a software library would be used. Typically, model-to-model transformations, consistency checks, or more generally speaking, recurring and object-independent operations are defined this way.

In both cases, story diagrams can be used to generate executable source code or be executed using an interpreter. Besides execution, the formally defined story diagrams can also be analyzed to guarantee certain behavioral properties [Mey09, Zün09]. For example, model checking can be used to check whether a certain invariant holds (e.g. that all accessible variables are still accessible after a refactoring operation) or if a critical state can ever be reached (e.g. if an attribute or method has no parent class after a refactoring operation which would result in an incorrect program).

A complete description of the story diagrams' abstract syntax is given in the Appendix B. There is also a grammar that determines all feasible story diagrams by constraining their structure. The latest version of this grammar can be found in Thomas Klein's diploma thesis [Kle99].

### 3.3.2. Activities, Activity Parameters and Return Values

Since story diagrams can be seen as special UML activity diagrams, we reused the class names defined by the UML 2. Thus, similar to UML activity diagrams, a story diagram is represented by a so-called activity (class Activity).

Each story diagram can have parameters. We distinguish *in* and *out* parameters, i.e. parameters representing arguments given when a story diagram is called (*in*) and parameters representing return values (*out*). Parameters can be *in* and *out* parameters at the same time. The story diagram in Figure 3.14 has two *in* parameters c and text as well as an *out* parameter attr of the type Attribute. If there are more than one *out* parameter, these are comma-separated.

If a story diagram defines the behavior of a method, the parameters are defined by the corresponding method's signature. In this case, the number of *out* parameters is limited to one single parameter and represents the only *return* value of the method and story diagram. Besides these parameters, there is another implicitly defined parameter this which – similar to Java's this keyword – represents the object that the story diagram belongs to.

In case a story diagram is not defining a method's behavior, it defines its own signature explicitly with according *in* and *out* parameters. The number of *out* parameters is allowed to be arbitrary in this case and there is no *this* parameter.

The values or objects returned after execution of a story diagram are defined by expressions in the *stop* activity nodes. For example, the object matched to the object variable a is returned by the story diagram in Figure 3.14 in case of a successful execution. This is specified by the expression attr := a which represents an assignment of the value of object variable a to the *out* parameter attr. Otherwise, an empty reference is returned which is specified by the keyword null. This notation is taken from Matthias Meyer [Mey09].

## 3.3.3. Activity Nodes, Activity Edges

A story diagram's control flow is defined by activity nodes and activity edges, similar to UML activities. Except for the cases where an activity node represents a call of another story diagram, each node embeds a story pattern to specify the corresponding behavior. Such activity nodes are called *story nodes*. Executing a story node results in executing the embedded story pattern.

In contrast to single story patterns, story patterns contained in story nodes of a story diagram have a different scope. Here, you can reuse all object variables declared in the story patterns of preceding story nodes. For example, in Figure 3.11 (p. 23), the object variable parentClass is reused in the second story pattern by specifying the variable as a bound variable, i.e. the variable does not have to be matched anymore.

Executing a story node means executing the corresponding story pattern which, in turn, means finding a subgraph with the specified properties (e.g. finding objects of a certain type, with certain attribute values, and with certain connections) and performing specified modifications of the found subgraph (e.g. creating or removing objects and links or changing their attribute values).

We distinguish two kinds of story nodes: *modifying story nodes* and *matching story nodes*. A matching story node contains a story pattern that only matches a specified object structure, but does not change it. A modifying story node also performs modifications of the matched object structure. For static analyses of model transformations described with story diagrams, it is helpful to know which tranformations do not modify the instance model.

## 3.3.4. Decision Nodes, Guards, and Loops

The behavior of a story node is defined by its story pattern. Therefore, since trying to find a subgraph defined by a story pattern can fail, each execution of a story node can also fail. To distinguish the cases of a successful story node execution and its failure, the outgoing activity edges can be provided with the guards [success] and [failure] (see Figure 3.15 a)). The control flow is following the activity edge with the *success* guard in case of a successful story node execution, i.e. a successful matching of the corresponding story pattern. Otherwise it follows the activity edge with the *failure* guard. If an outgoing activity edge has no guard, it covers both cases, success and failure. The guards [success] and [failure] can only be used pair-wise (exactly two outgoing activity edges with exactly these two guards). The first story node in Figure 3.11 (p. 23), for example, uses these guards.

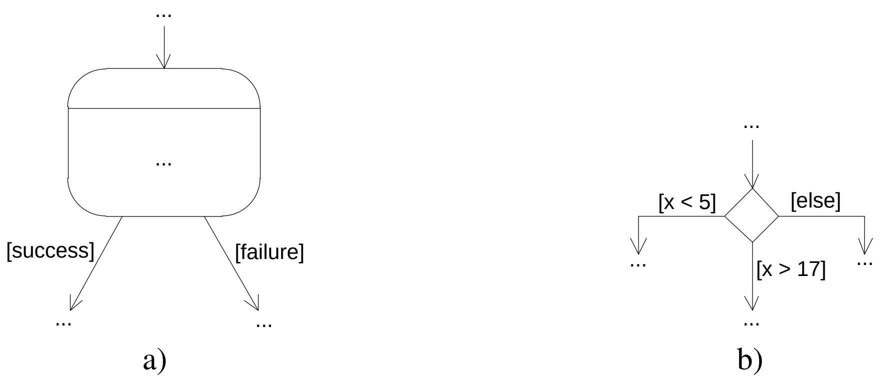


Figure 3.15.: Examples For Decisions

Besides [success] and [failure], boolean expressions can be used as guards (see Figure 3.15 b)). We use the *junction node* – depicted as a diamond – for decisions that do not depend on a previous story node. In this case, the boolean expression of the outgoing activity edge is evaluated to *true* or *false*. There can be arbitrarily many outgoing activity edges with boolean guard expressions. The boolean expressions have to mutually exclude each other and the corresponding guards have to be combined with an outgoing activity edge with the guard [else]. I.e., if there is a guard with a boolean expression, there is also an activity edge with the guard [else].

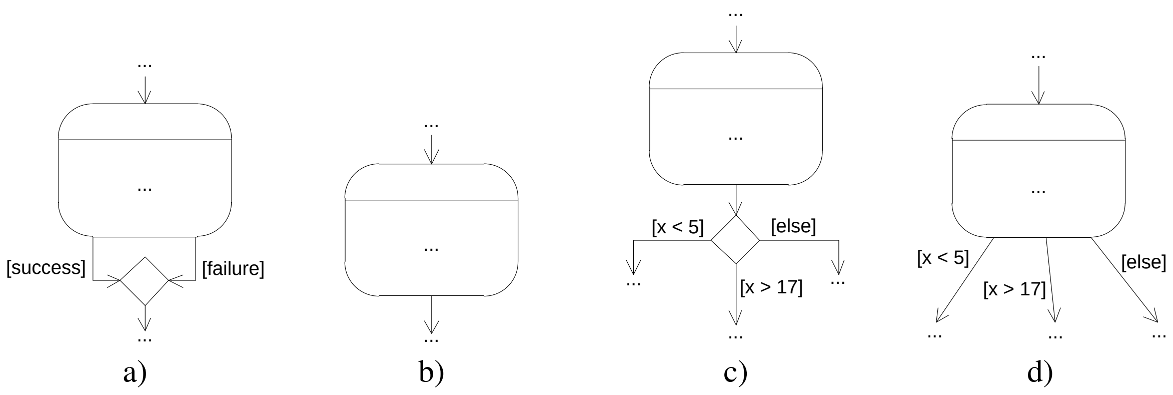


Figure 3.16.: Examples For Control Flow Simplifications: case a) is semantically equivalent to b), case c) is semantically equivalent to d)

The junction node can also be used to merge several control flows into one (several activity edges point to a junction node which has only one outgoing activity edge, see Figure 3.16 a)).

In order to simplify the control flow as shown in Figures 3.16 a) and c), we also allow shorthand notations as illustrated in Figures 3.16 b) and d). The control flow in case b) is semantically equivalent to that in case a). The control flows in cases c) and d) are also equivalent.

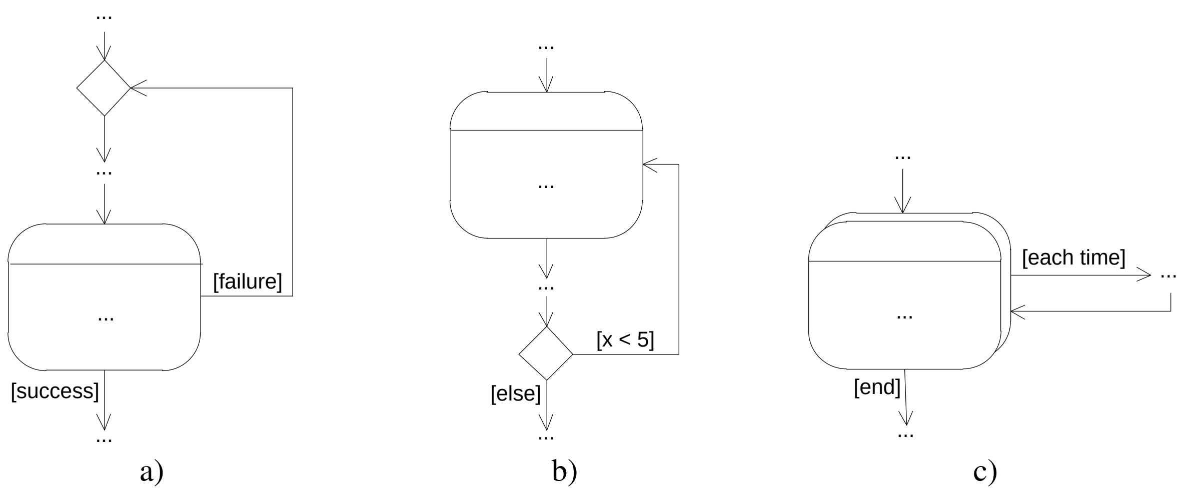


Figure 3.17.: Examples For Loops

Activity edges and guards can be used to model loops as illustrated in Figures 3.17 a) and b). There is an additional construct to model loops which allows to perform the same operations with each occurrence of a certain object structure. For that purpose, we use a special activity node that we call *for-each* activity node and special guards [each time] and [end].

The for-each activity node is depicted by a cascaded activity node (see Figure 3.17 c)). The second activity node in Figure 3.11 (p. 23), for example, is a for-each activity node. Such a node represents a loop where the contained story pattern is executed as often as new subgraphs can be matched that differ from the previously matched graphs by at least one other matched object. The story pattern in the for-each activity node in Figure 3.11 is matched for each existing pair of a method (object variable anyMethod) and corresponding call object (object variable c). Besides the matching itself, all *destroy* and *create* steps are also executed for each of these matched subgraphs. In general, after execution of the story pattern in the for-each activity node, the control flow follows the activity edge with the guard [each time], if available (see Figure 3.17 c)). This edge is optional and can be omitted like in Figure 3.11. The [each time] edge leads to the activity node (or a sequence of such nodes) that is to be executed after each successful execution of the for-each activity node. After that, the control flow returns to the for-each activity node in order to match and process the next object structure that can be matched by the for-each activity node. If there is an activity edge with the guard [each time], there has also to be such an activity edge leading back to the for-each activity node. This constitutes a loop. Finally, the control flow is guided by the activity edge with the guard [end] which leads to the activity node to be executed after the loop. Each for-each activity node must have such an outgoing [end] activity edge.

## 3.3.5. Story Diagram Calls

Story diagram calls are special nodes in a story diagram which are used to invoke other story diagrams. Similar to method calls, this reduces redundancy and promotes reuse.

As described in Section 3.3.2, a story diagram can have an arbitrary number of in and out parameters. When calling a story diagram, concrete arguments have to be assigned to the in parameters. Consequently, if an object variable named n is bound somewhere in the story diagram, the identifier n can be used to pass this object variable as an argument to a call. If the called story diagram has out parameters, those are bound explicitly by assignments at the stop activity node. They can be used in the calling story diagram by specifying object variables whose names match those of the out parameters.

For in parameters, we use a call-by-reference semantics. If an object that is passed as an in parameter is modified in the called story diagram, those modifications remain after the called story diagram has terminated. The object in question can be used in the calling story diagram after the call but the call may have modified its attributes or its links.

An example of a story diagram call is shown in Figure 3.18.

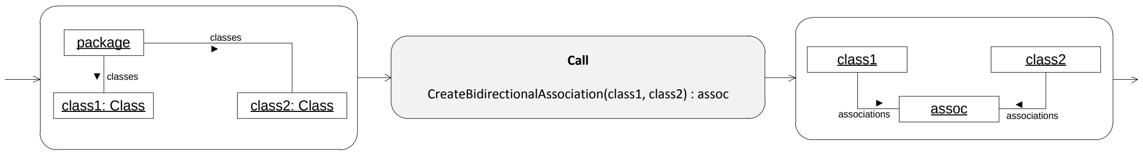


Figure 3.18.: Example of a story diagram call

The first story pattern in Figure 3.18 shows the bound object variable package. Two new object variables class1 and class2 are bound in that pattern. The next node with the grey background is a story diagram call which is also signified by its label Call. Beneath the label, the name of the called story diagram is given, in this case CreateBidirectionalAssociation. Assume that the called story diagram has two in parameters of the type Class and one out parameter of the type Association. The two classes that were bound in the first story pattern, class1 and class2 are passed to the call as arguments. They can be used in the story node after the call without passing the back as out parameters. The modifications carried out by the called story diagram (i.e. the creation of the assoc object and its connection to class1 and class2) are retained after the call terminates. The result of the call is bound to the object variable assoc. The type of this variable is determined by the out parameter type, i.e., in this case the type Association.

If a story diagram has no out parameters, the keyword void follows the colon instead of the out parameter's names (see Figure 4.2 for an example).

## 3.4. Expressions

Story diagrams and story patterns use a mainly graphical syntax. Though, some things can compactly be described by text, e.g., restrictions of attribute values to a certain range. For this purpose we added a small textual language for expressions to story diagrams to cover value comparisons, value assignments, simple arithmetic expressions, etc. In this first version of this technical report we do not describe expressions in detail.

Besides our small textual language for certain expressions, we support embedding OCL expressions, for example, to determine a value to be assigned to an attribute[5]. In future, this will be extended to also cover arbitrary other textual expressions that have to be interpreted by a given interpreter.

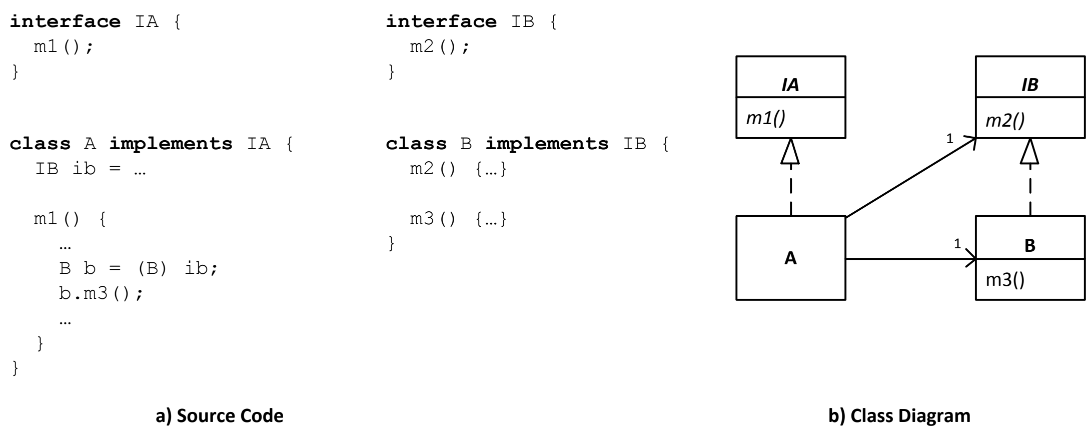
[5]Currently, the OCL tools in Eclipse (http://www.eclipse.org/modeling/mdt/?project=ocl), as far as possible, comply with the OMG OCL standard 2.3 (http://www.omg.org/spec/OCL/2.3/Beta2/PDF). We use these tools to interpret the OCL expressions. More details about this issue can be found here:
http://www.eclipse.org/projects/project-plan.php?planurl=http://www.eclipse.org/modeling/mdt/ocl/project-info/plan_indigo.xml&component=Eclipse

# Chapter 4. Complete Example

This chapter presents a complete example of a transformation with story diagrams. The setting of the example is explained in Section 4.1. The next section then presents several complex story diagrams that specify the desired transformation.

## 4.1. Motivation of the Example

A well-known principle of object-oriented programming says *"Program to an interface, not an implementation."* [GHJV95]. By only accessing interfaces instead of concrete classes from a given class, that class remains independent of concrete implementations. The accessed classes can be exchanged transparently without breaking the program. If this principle is neglected, accidentally or intentionally, this is known as an *interface violation*.



```
interface IA {
  m1();
}

class A implements IA {
  IB ib = …

  m1() {
    …
    B b = (B) ib;
    b.m3();
    …
  }
}
```

```
interface IB {
  m2();
}

class B implements IB {
  m2() {…}

  m3() {…}
}
```

a) Source Code

b) Class Diagram

Figure 4.1.: Example of an Interface Violation

In Figure 4.1, a simple example of an interface violation is depicted. The classes A and B implement the interfaces IA and IB, respectively. Following the design principle "Program to an interface, not an implementation", the classes are expected to interact through their interfaces. However, A calls the method m3() from B because m3() is not provided by the interface IB. A down-casts the object ib to the concrete type B in order to access m3(). This intentional bypassing of the interface IB is an interface violation.

There are several possibilities to remove an interface violation from a program. A trivial solution would be to delete the downcast and the call from the implementation of m1. This would, of course, remove the interface violation but also change the program behavior. A more sensible solution which will be used in this chapter is the extension of the interface IB to contain the method declaration of m3. By adding this declaration to IB, the class A can call m3 via the interface. The downcast becomes unnecessary and can be removed. At the same time, the behaviour of m1 is preserved.

To this point, the refactoring is very similar to the *Extract Interface* refactoring described by Fowler [Fow99]. Extending an existing interface, however, is a little more complicated as there may already be other classes that implement IB. If m3 is added to IB, those other implementing classes all have to be extended by a (possibly empty) method implementation of m3 in order to remain compilable.

A refactoring that removes an interface violation by extending an interface as described above is modeled with story diagrams and presented in the following section.

## 4.2. Story Diagram: Remove Interface Violation

Figure 4.2 shows the story diagram to remove an interface violation. The underlying type graph is the GAST metamodel that was introduced in Section 2.4. The story diagram consists of five story nodes and two activity calls. This section explains the story diagram step by step.

The story diagram has five in-parameters: call, interface, method, castStmt, and accessedMethodOwner. call represents the statement that calls the method in the concrete class (the call of m3 in m1). interface is the interface that will be extended (IB in the example). method is the method that is currently not declared in the interface (m3()). castStmt refers to the statement that down-casts the interface type to the concrete class type (i.e., the statement B b = (B) ib;). Finally, accessedMethodOwner is the class that contains the called method (B in the example).

The first story node (after the start node) creates a method declaration in the interface (methodDecl). This new method declaration is declared as public (attribute assignment visibility := PUBLIC) and abstract (attribute assignment abstract := true). The declaration receives the same name as the formerly called method (attribute assignment name := method.name, m3 in the example). The new method declaration is added to the methods of the interface by creating a method link between interface and methodDecl. The target accessed by the call is changed by deleting the link between call and method and recreating it between call and methodDecl. The return type of the method is set by creating a new object typeAccessNew of the type DeclarationTypeAccess and connecting it to methodDecl. It points to the same GASTType as the old declaration type access of the method.

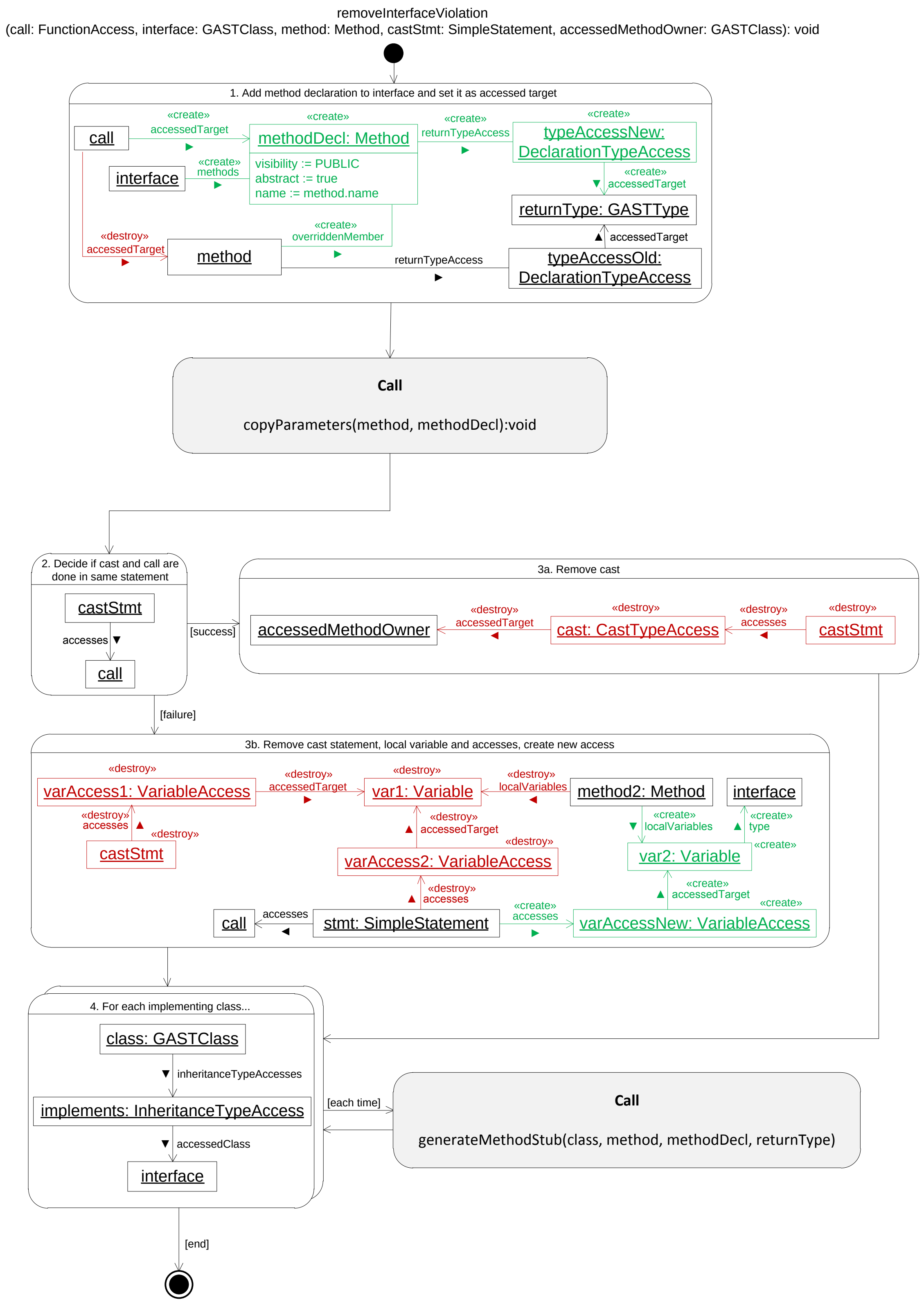


Figure 4.2.: Story Diagram: removeInterfaceViolation

The next node is an activity call node. It calls the story diagram copyParameters which is described in detail in Section 4.2.2. This story diagram is responsible for copying all the parameters of the formerly called method to the newly created declaration methodDecl.

The following story node contains only the two bound, mandatory object variables castStmt and call. It's responsibility is to try and match the link accesses between those object variables. If the link exists that means that the cast and the call are part of the same statement. In that case the matching of the story node is successful and the control flow continues along the transition labelled with *[success]* to story node 3a. If the matching fails, i.e., the link does not exist and the cast and the call are therefore not part of the same statement, the story node is left via the *[failure]* transition. This distinction is necessary because the effort to remove the cast statement is much greater if the cast is not done in the same statement as the call (compare story nodes 3a and 3b).

If the cast is in the same statement as the call, story node 3a is executed: The castStmt and its access to B are deleted. If the cast is not in the same statement as the call that means that the cast is executed at some point before the call and the resulting down-cast object is stored in a temporary variable. In this case, this temporary variable can be deleted along with the accesses to it from the call and the cast statements. Instead, a new variable of the interface type (IB in the example) is created and then accessed by the call statement. In both cases, activity node 4 is executed next.

Activity node 4 is responsible for adapting all other classes that implement the now changed interface. Thus, the node is a for-each activity node that binds a class which is connected to the interface in each iteration. For each of those bindings, the node that is reachable via the [each time] transition is executed (see Section 3.3). In this case that is a story diagram call of the story diagram generateMethodStub which is explained in the following section.

When no new classes implementing the interface can be found, i.e. method stubs have been generated for all implementing classes, the story diagram terminates.

### 4.2.1. Story Diagram: Generate Method Stub

The story diagram generateMethodStub is shown in Figure 4.3. It creates a method which implements a method methodDecl in a given interface. This is accomplished by two story nodes and one story diagram call. The first node checks if the given class contains a method with the same name as the given declaration methodDecl. The check is performed by the expression 'name = methodDecl.name'. Since the object variable method is negative (crossed-out), the matching of this story node is considered successful if *no* such method exists in the class. In that case the next story node is executed. If a method of the name in question already exists, the execution of the first story node fails and the story diagram terminates.

The second story node creates a new methodStub in the given class. The visibility of this method is set to public and its name is set to the name of the method declaration as signified by the expression 'name := methodDecl.name'. The correct return type for the method is set by creating a newTypeAccess from the methodStub to the returnType that was passed to this story diagram as a parameter.

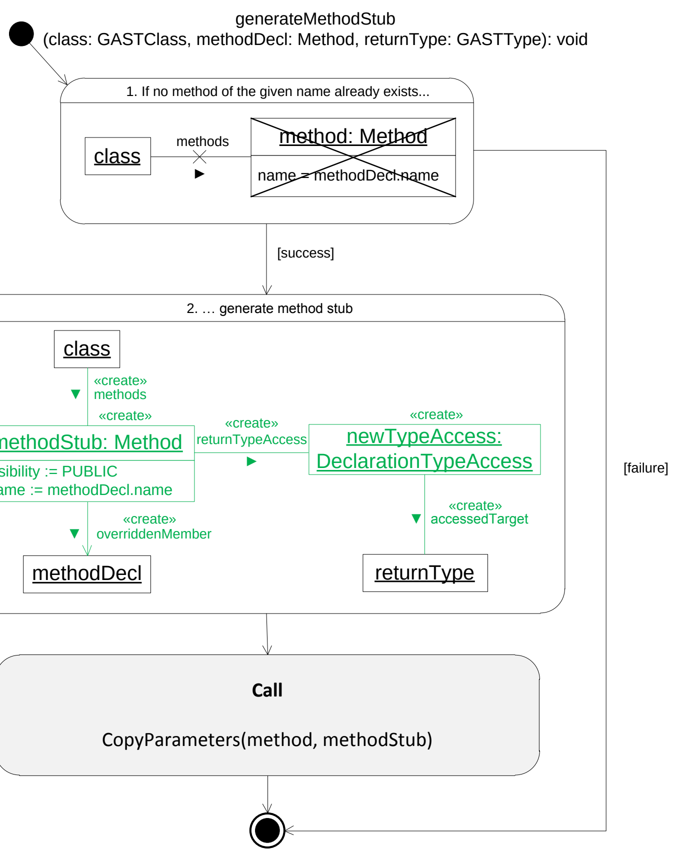


Figure 4.3.: Story Diagram: generateMethodStub

Finally, the story diagram CopyParameters is called in the story diagram call node. The method and the methodStub are passed as parameters. The called diagram then copies all parameters from the given method to the newly created methodStub. It is explained in the following section.

### 4.2.2. Story Diagram: Copy Parameters

The story diagram copyParameters (see Figure 4.4) copies all the parameters from a sourceMethod to a targetMethod. Both methods are provided as parameters. The diagram consists of two story nodes.

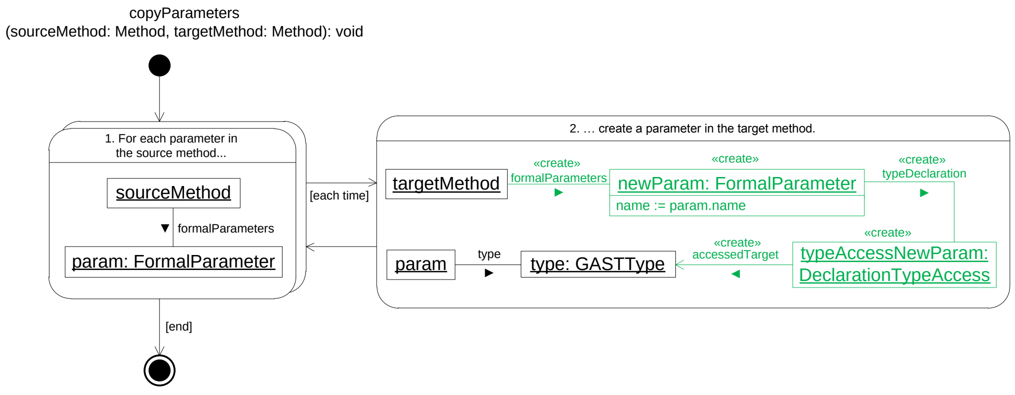

copyParameters
(sourceMethod: Method, targetMethod: Method): void
1. For each parameter in the source method...
sourceMethod
formalParameters
param: FormalParameter
[each time]
[end]
2. ... create a parameter in the target method.
targetMethod
«create» formalParameters
«create»
newParam: FormalParameter
name := param.name
«create» typeDeclaration
param
type
type: GASTType
«create» accessedTarget
«create»
typeAccessNewParam: DeclarationTypeAccess

Figure 4.4.: Story Diagram: copyParameters

The first activity node is a for-each activity node. It successively binds all formal parameters of the given sourceMethod to the object variable param. Each time a new parameter is bound, the second node is executed. There, a new formal parameter newParam is created in the targetMethod. Its name is set to the same name as the original parameter's by the expression 'name := param.name'. The type is also set accordingly by binding the type of param. Then, a new access to that type is created and connected to newParam.

# Chapter 5. Related Work

In this chapter, we give an overview about scientific publications related to story diagrams. First, we provide an extensive summary of previous work about story diagrams in Section 5.1, including their origins. In Section 5.2, we report on extensions and applications of story diagrams. Finally, we briefly describe related and similar concepts in the literature in Section 5.3.

## 5.1. Origins and Previous Work on Story Diagrams

Story diagrams have first been described by Fischer et al. [FNTZ00] and Jahnke and Zündorf [JZ98] in 1998. The foundations of story diagrams lie in the programmed graph rewriting systems PROGRES [SWZ95] which has been developed at the University of Aachen since 1989. Story diagrams (or story flow diagrams as they were called in early publications) adapt and enhance the PROGRES approach to a UML-like notation and an object-oriented data model [JZ98]. They have an easily comprehensible graphical syntax and well-defined semantics. Zündorf [Zün01] describes the syntax and semantics of story diagrams in detail. A graph grammar that formally describes the syntax of the control flow of story diagrams was defined by Klein [Kle99].

Story diagrams are embedded in a rigorous and systematic software development method called *story-driven modeling* (SDM) [Zün01, DGZ04]. While existing approaches like UML focus on the specification of the static structure of software, SDM combines, amongst others, UML class diagrams and story diagrams to allow completely specifying the structure and behavior of software systems. Furthermore, SDM describes how such a software specification can be derived from requirements. First, each use-case in the requirements is refined by a set of sample scenarios defined by so-called *story boards*. A story board is a sequence of single snap shots of graph-like object structures, describing changes in these object structures. Next, the static class structure of the system is derived from the story boards and further refined. Given the sample scenarios, the general dynamic behavior of the system is then defined using story diagrams. Finally, the implementation of the software system can be automatically generated from these formal models.

From the beginning, tool support for story diagrams was a main focus. FUJABA, an acronym for "From UML to Java And Back Again"[1], was the first tool which implemented the concept of story diagrams. In December 1997, the project started at the University of Paderborn. A first prototype was implemented in the course of a master's thesis [FNT98]. As story diagrams specify the behavior of software, the execution of story diagrams is an important requirement. For instance, Zündorf, Schürr and Winter [ZSW99] describe how story diagrams can be compiled into Java code. This code generation approach was also integrated into FUJABA.

A first public tool demonstration of FUJABA was presented at the ICSE 2000 [NNZ00], showing advanced class and story diagram modeling facilities as well as graphical debugging and simulation.

In the following, story diagrams and FUJABA have been modified and enhanced. Originally, story diagrams used expressions of the target programming language to define constraints, return values etc., i.e. if a story diagram was to be compiled into Java code, Java expressions had to be used. Stölzel, Zschaler and Geiger [SZG07] integrated OCL into story diagrams, making them more platform-independent. They connected FUJABA to the Dresden OCL toolkit [WTF11], allowing a code generation for story diagrams including the OCL constraints.

To improve flexibility for the execution of story diagrams, Giese, Hildebrand and Seibel [GHS09] present an interpreter for story diagrams. In contrast to executing generated Java code, with this approach generated story diagrams can be executed immediately. This allows, for instance, to create higher-order transformations where story diagrams are created by other story diagrams and can immediately be executed. As interpreting in general is slower than compiling, the authors implemented a new dynamic matching policy for their interpreter.

Tichy, Meyer and Giese [TMG06] identified some semantic issues in story diagrams. First, when creating more than one element in a story pattern, the order of creation is undefined. In general, this is no problem; however, in certain failure situations and when creating links in ordered associations, this may lead to non-deterministic behavior. However, defining a creation order would contradict the declarative nature of story patterns. Thus, we decided not to include an explicit creation order. However, for the failure case, an exception transition can be defined where it can be explicitly modeled how to deal with the failure. To define a link order in an ordered association, LinkConstraints can be used (see Section B.7.2.10 on Page 93). (LinkConstraints will be described in detail in later versions of this document.)

Second, when having a link between two set variables setA and setB, the intuitive semantics would be to have every set element in setA connected to every element in setB. However, this is neither supported by the tools nor allowed by the formal semantics described by Zündorf [Zün01]. We deal with sets in later versions of this document.

Third, consider there is a class with two qualified associations (to other classes) that have each other as a qualifiers. When creating one link for each of the two qualified association in one story pattern, the first association that is created is qualified by the null value although it

[1]The acronym is derived from a preceding tool called FUCABA ("From UML to C++ And Back Again") [JZ97].

could be qualified using the correct object (considering this is already bound). Again, we deal with this issue in later versions of this document.

Forth, the set of possible bindings that match in a for-each activity may be extended by this very for-each activity, i.e., the activity changes something that makes new elements match for the for-each condition. In the original work on story patterns, it was not clear how this should be handled. Thus, we define that we use a *fresh matches* semantics for for-each activities (in contrast to a *pre-select* semantics) in Section 3.3.4.

Fifth, as creations may fail, e.g., due to resource constraints, the authors propose that a story diagram should be able to react to the result of a creation. As mentioned before, an exception transition can be used to deal with such failures.

The control flow of story diagrams is modeled explicitly. However, in certain situations, it is useful to only implicitly define the execution order, as it may significantly improve the comprehensibility of a story diagram. Thus, Meyers and Van Gorp [MG08] propose to add a new language construct for the non-deterministic selection of a execution order.

In [Sta08], Stallmann presents an extension of story patterns which is called *enhanced story pattern*. They extend story patterns by so-called *insets*. Insets carry a qualifier which applies to all object and link variables in the inset. That allows to mark sub-graphs as negative, to specify *and* and *or* conditions on subgraphs and to qualify a subgraph by $\forall$. We will adopt these ideas in future versions of this document.

Becker et al. present means for structuring complex transformations into several independent story diagrams which can be called in a well-defined manner [BvDHR11]. They propose inventing explicit call activities which invoke other story diagrams and also support polymorphic dispatching. Polymorphic dispatching can also be used for the aforementioned case of non-deterministic execution order. Calls are described in Section 3.3.5. We will give details on the polymorphic dispatching mechanism in story diagrams in later versions of this document.

Until 2010, different branches of story diagrams and of FUJABA were developed, leading to severe difficulties when exchanging data due to incompatibilities. In an effort to again unify the different branches, a task force was started in 2010. A first result of this joint effort of the SDM community was a new unified and consolidated meta-model for story diagrams based on EMF [HRvD+11]. This new meta-model is the foundation for future projects; this technical report is also based on this meta-model. One extension is the support for explicitly modeling expressions. However, this is not described in detail here.

## 5.2. Applications and Extensions of Story Diagrams

In the area of reengineering, Niere et al. [NSW+02] propose to specify design patterns with a graphical DSL which has strong relations to story patterns. In order to detect the specified patterns in source code, these DSL patterns are translated into story diagrams which are then executed through code generation. This approach has first been implemented in FUJABA and later in the Reclipse Tool Suite [vDMT10]. In follow-up work by Fockel [Foc10], the generated story diagrams are no longer transformed into executable code but are interpreted to allow for easier debugging of the pattern specifications.

Giese and Klein extend story patterns to so called Story Decision Diagrams (SDDs) that allow to express complex safety properties [GK06]. Basically, they require story patterns of SDDs to be non-modifying (i.e., no «`create`» or «`destroy`» elements) and add features of logics such as quantification, implication, and negation. After the evaluation of such a property, a regular story pattern may be specified which describes a change operation that should be executed.

Giese and Klein also present Timed Story Scenario Diagrams (TSSDs), which are used to specify structural and temporal properties of systems in an integrated way [KG07].

Tichy et al. [THHO08] describe how story diagrams can be used to describe reconfigurations of component-based architectures, as, for instance, in MechatronicUML [BBD+12]. A transformation language called Component Story Diagrams is used to specify reconfiguration steps. Component Story Diagrams use the concrete syntax of components for specifying the reconfiguration operations. This language is transformed to story diagrams that can be executed to perform the actual reconfigurations.

Zündorf [Zün09] proposes a framework for computing the state-space of a specification in terms of story diagrams and an initial instance model. It has been used for model checking the leader election protocol and for a case study presented in [HSJZ10].

Meyer [Mey09] adds a few specialized constructs to story diagrams thereby extending them to transformation diagrams. Some of these extensions, such as multiple out parameters, have been integrated into the story diagrams presented in this report (see Section 3.3.2). Similar to our complete example (Chapter 4), Meyer uses the transformation diagrams to specify refactorings of object-oriented source code. To verify that certain properties of the code (e.g. variable accesses) are preserved by the refactorings, he extends Schilling's approach [Sch06] to proving inductive invariants.

In [HH], Heinzemann and Henkler extend story diagrams to timed story diagrams. Timed story diagrams are based on timed graph transformation systems [EHH+11] that extend graph transformation systems by clocks as known from timed automata [AD94]. They are used to model time-dependent reconfigurations of an instance model. In [EHH+11], they are used as a means to define the semantics of reconfigurations in real-time systems. In [HSE10], a framework for reachability analysis on timed story diagrams has been introduced which explores the state-space defined by the timed story diagrams and an initial instance model. It is based on the framework introduced in [Zün09].

## 5.3. Work Related to Story Diagrams

Model transformation has become an important research topic during the last years. Several concepts and tools with different scopes and applications have been proposed.

Several model transformation approaches exist which are similar to story diagrams.

Here, we focus on those solutions that have a reasonable documentation available. For a more comprehensive overview of transformation approaches see, for example, [CH06]. Current transformation tools can, for instance, be found in [MRvG10].

### 5.3.1. Endogenous, In-Place Model Transformations

*Henshin* [BESW10] is a model transformation language for in-place transformations of EMF-based models. It uses pattern-based rewrite rules (called "transformation rules") and control-flow-based operational semantics (called "transformation units") on top of it. Transformation units can also be called by other transformation units, also including parameters.

*MOLA* [KBC04] is an in-place model transformation language with a graphical syntax similar to story diagrams. Transformation rules may consist of multiple matching and modification patterns and the control flow inside a transformation rule can be specified with a focus on the loop construct. Furthermore, it also allows calling other transformations rules.

*Groove* [Ren04] is a graph transformation tool with a focus on analyzing graph transformation systems. Its rules consist of single rewrite patterns. For instance, given a rule set and a start graph, Groove can explore the graph state space and use this for model checking. It also features so called "control programs" which allow the user to restrict which rules can be applied and in which order. It provides model checking of LTL properties [Ren08] and CTL properties [KR06].

*VIATRA* [VB07], a textual language, uses abstract state machines to specify the control flow and graph transformation rules for elementary model manipulations. It also addresses modularization by reusable patterns that are called from the graph transformation rules.

### 5.3.2. Exogenous, Inter-Model Transformations

In general, Story Diagrams can also be used to specify inter-model transformations. In this case, a story diagram would contain elements from both the source and the target model. If necessary, a trace model could also be created. In comparison to dedicated inter-model transformation languages, story diagrams may be more tedious to use in this application scenario. However, when a transformation requires extensive pre-computations or complex distinction of cases, story diagrams are a reasonable alternative.

*QVT Operational* [Obj11b] is a operational model transformation language designed for writing unidirectional transformations which is part of the OMG QVT standard. However, QVT-O is a textual transformation language which may not be well-suited in many cases [Moo09].

In declarative inter-model transformation languages like *Triple Graph Grammars* (TGGs) [Sch94], the control flow cannot be defined explicitly. Instead, the order of the rule application is implicitly defined by preconditions of the transformation rules. However, when more than one rule has a fitting precondition, the rule to be applied is selected non-deterministically, dependent on the concrete transformation tool implementation, or by a given rule priority. This can make the comprehension of a TGG rule set difficult.

In *QVT Relations* [Obj11b], which is similar to TGGs, control flow may also be explicitly specified by using where clauses.

The *Atlas Transformation Language* (ATL) [JABK08] is a hybrid inter-model transformation language, integrating declarative and operational aspects. It is similar to QVT, but only has a textual representation of the transformation rules.

# Chapter 6.

# Conclusions and Future Work

In this technical report, we presented a consolidated version of the endogenous, graphical in-place model transformation language story diagrams. Story diagrams combine imperative modeling of control flow using UML Activity Diagrams which a declarative graph rewriting language called story pattern. Story patterns are based on typed attributed graph transformations and formally define the behavior of the activity nodes of story diagrams. We briefly covered the foundations and explained the most important language concepts and the concrete syntax of, both, story patterns and story diagrams. That includes the idea of story patterns and their usage in story diagrams as well as a concepts for invoking story diagrams from which other story diagrams. We illustrated these concepts with a comprehensive example that showed the application of several story patterns for the purpose of removing an interface violation in a program.

In the past, story diagrams have proven to be useful, both, as a model transformation language and as a language for specifying behavior of object-oriented programs. In the appendix, we provide the description of a reference implementation of an interpreter for story diagrams. It also contains the technical documentation of the current abstract syntax of story diagrams.

Work on and with story diagrams will continue in the future. The new metamodel for story diagrams which is described in detail in Appendix B has been proposed quite recently [HRvD$^{+}$11]. It will definitely be extended and refined. This ensures that story diagrams will proceed to form the basis of scientific approaches in such diverse fields as reverse engineering [vDMT10] or verification of embedded systems [HSE10].

Several advanced concepts of story patterns and story diagrams are currently not presented in this report. We refer to the related publications presented in Sections 5.1 until these concepts have been included in this report. In future versions will elaborate on advanced concepts in story diagrams, like the use of sub patterns in story patterns, complex expressions, and the application of templates. It will also contain concise portrayals of the different approaches that build on story patterns.

## Acknowledgments

This report would not have been possible without the support of many people. The authors would like thank Steffen Becker and Christopher Brink for their proof-reading and helpful comments. Thanks go also to Ingo Budde and Aljoscha Hark for their technical support and their help with the implementation of story diagrams.

The continuous development of story diagrams could only be achieved with the help of all the people that (besides the authors of this report) worked on story diagrams and their various applications over the years.

These are (in alphabetical order): Steffen Becker, Manuel Bork, Thomas Buchmann, Ira Diethelm, Alexander Dotor, Thorsten Fischer, Markus Fockel, Leif Geiger, Holger Giese, Stefan Henkler, Jörg Holtmann, Ruben Jubeh, Felix Klar, Thomas Klein, Hans Köhler, Alexander Königs, Ingo Kreuz, Marius Lauder, Elodie Legros, Matthias Meyer, Bart Meyers, Ulrich Nickel, Jörg Niere, Ulrich Norbisrath, Simon Oberthür, Carsten Reckord, Wilhelm Schäfer, Daniela Schilling, Christian Schneider, Andy Schürr, Florian Stallmann, Mirko Stölzel, Ingo Stürmer, Matthias Tichy, Lars Torunski, Oleg Travkin, Pieter van Gorp, Jörg Wadsack, Robert Wagner, Jens Weber, Ingo Weisemöller, Lothar Wendehals, Jim Welsh, Andreas Winter, Steffen Zschaler, and – especially – Albert Zündorf.

# Bibliography

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 40

[BBD+12] Steffen Becker, Christian Brenner, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Julian Suck, Oliver Sudmann, and Matthias Tichy. The MechatronicUML Method – Process, Syntax, and Semantics. Technical Report tr-ri-12-318, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, February 2012. 40

[BESW10] Enrico Biermann, Claudia Ermel, Johann Schmidt, and Angeline Warning. Visual Modeling of Controlled EMF Model Transformation using Henshin. In *Proceedings of the 4th International Workshop on Graph-Based Tools*, 2010. 41

[BvDHR11] Steffen Becker, Markus von Detten, Christian Heinzemann, and Jan Rieke. Structuring Complex Story Diagrams by Polymorphic Calls. Technical Report tr-ri-11-323, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, March 2011. 39

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45:621–645, July 2006. 11, 41

[DGZ04] Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic Story Driven Modeling, A Case Study. In *Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, May 2004. 37

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006. 4, 6, 7

[EHH+11] Tobias Eckardt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, and Wilhelm Schäfer. Modeling and Verifying Dynamic Communication Structures based on Graph Transformations. *Computer Science - Research and Development*, pages 1–20, July 2011. 40

[Epp95] David Eppstein. Subgraph Isomorphism in Planar Graphs and Related Problems. In *Proceedings of the 6th annual ACM-SIAM Symposium on Discrete Al-*

*gorithms*, pages 632–640. Society for Industrial and Applied Mathematics, 1995. 4, 13

[FNT98] Thorsten Fischer, Jörg Niere, and Lars Torunski. Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling. Diploma thesis, University of Paderborn, Germany, July 1998. In German. 38

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT '98 Selected papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 2000. 1, 11, 37

[Foc10] Markus Fockel. Interpretation von Graphtransformationsregeln zur statischen Erkennung von Software-Mustern. Master's thesis, University of Paderborn, Germany, October 2010. (In German). 39

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. 32

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 31

[GHS09] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT '09)*, volume 18 of *Electronic Communications of the EASST*, 2009. 38, 51

[GK06] Holger Giese and Florian Klein. Beyond Story Patterns: Story Decision Diagrams. In *Proceedings of the 4th International Fujaba Days 2006*, 2006. 1, 40

[HH] Christian Heinzemann and Stefan Henkler. Timed Story Driven Modeling. Technical Report tr-ri-11-326, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Germany, July. 40

[HRvD$^+$11] Christian Heinzemann, Jan Rieke, Markus von Detten, Dietrich Travkin, and Marius Lauder. A new Meta-Model for Story Diagrams. In *Proceedings of the 8th International Fujaba Days*, May 2011. 39, 43

[HSE10] Christian Heinzemann, Julian Suck, and Tobias Eckardt. Reachability Analysis on Timed Graph Transformation Systems. In *Proceedings of the 4th International Workshop on Graph-Based Tools*, 2010. 40, 43

[HSJZ10] Christian Heinzemann, Julian Suck, Ruben Jubeh, and Albert Zündorf. Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study. In *Transformation Tool Contest*, 2010. 40

[JABK08] Frederic Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. 42

[JZ97] Jens H. Jahnke and Albert Zündorf. Rewriting poor Design Patterns by good Design Patterns. In *Proceedings of the ESEC / FSE '97 Workshop on Object-Oriented Reengineering*. Technical University of Vienna, Austria, 1997. 38

[JZ98] Jens H. Jahnke and Albert Zündorf. Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modeling. In *Proceedings of 9th International Workshop on Software Specification and Design*, 1998. 37

[KBC04] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. In *Proceedings of Model-Driven Architecture: Foundations and Applications*, 2004. 41

[KG07] Florian Klein and Holger Giese. Joint Structural and Temporal Property Specification using Timed Story Scenario Diagrams. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, 2007. 1, 40

[Kle99] Thomas Klein. Rekonstruktion von UML-Aktivitäts- und Kollaborationsdiagrammen aus Java-Quelltexten. Diploma thesis, University of Paderborn, October 1999. 25, 37

[KR06] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006. 41

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *International Journal on Software and Systems Modeling*, 5(4):369 – 385, December 2006. 7

[Mey09] Matthias Meyer. *Musterbasiertes Re-Engineering von Softwaresystemen*. PhD thesis, University of Paderborn, 2009. 12, 25, 26, 40

[MG08] Bart Meyers and Pieter Van Gorp. Towards a Hybrid Transformation Language: Implicit and Explicit Rule Scheduling in Story Diagrams. In *Proceedings of the 6th International Fujaba Days*, 2008. 39

[Moo09] Daniel L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756 – 779, 2009. 41

[MRvG10] Steffen Mazanek, Arend Rensink, and Pieter van Gorp, editors. *Transformation Tool Contest 2010*, Malaga, Spain, 2010. 41

[NNZ00] Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. Tool demonstration: The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 742 – 745. ACM Press, 2000. 38

[NSW$^+$02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards Pattern-Based Design Recovery. In *Proceedings of the 24th International Conference on Software Engineering*, pages 338–348. ACM Press, May 2002. 39

[Obj03] Object Management Group. *Unified Modeling Language (UML) 1.5 Specification*, 2003. Document formal/03-03-01. 11

[Obj10a] Object Management Group. *Object Constraint Language (OCL), Version 2.2*, 2010. Document formal/2010-02-01. 21

[Obj10b] Object Management Group. *Unified Modeling Language (UML) 2.3 Superstructure Specification*, May 2010. Document formal/2010-05-05. 6

[Obj11a] Object Management Group. *Meta Object Facility (MOF) 2.4.1 Core Specification*, 2011. Document formal/2011-08-07. 6

[Obj11b] Object Management Group. *Query/View/Transformation (QVT), Version 1.1*, 2011. Document formal/2011-01-01. 41, 42

[QBe06] QBench project, 2006. http://www.fzi.de/index.php/de/forschung/forschungsbereiche/se/projekte/abgeschlossene-projekte/135-projekt-qbench. 8

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *Applications of Graph Transformation with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer Berlin / Heidelberg, 2004. 41

[Ren08] Arend Rensink. Explicit State Model Checking for Graph Grammars. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132. Springer Berlin / Heidelberg, 2008. 41

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. I. Foundations*. World Scientific Publishing Co., Inc., 1997. 3, 4, 6, 12

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition, 2008. 6

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163. Springer, 1994. 41

[Sch06] Daniela Schilling. *Kompositionale Softwareverifikation mechatronischer Systeme*. PhD thesis, University of Paderborn, 2006. 12, 40

[Sta08] Florian Stallmann. *A Model-Driven Approach to Multi-Agent System Design*. PhD thesis, University of Paderborn, April 2008. 39

[SWZ95] Andy Schürr, Andreas Winter, and Albert Zündorf. Graph Grammar Engineering with PROGRES. In *Proceedings of the 5th European Software Engineering Conference*, pages 219–234. Springer, 1995. 37

[SZG07] Mirko Stölzel, Steffen Zschaler, and Leif Geiger. Integrating OCL and Model Transformations in Fujaba. *Electronic Communications of the EASST*, 5, 2007. 38

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Component Story Diagrams: A Transformation Language for Component Structures in Mechatronic Systems. In *Postproceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems*, pages 27–39, 2008. 40

[TMG06] Matthias Tichy, Matthias Meyer, and Holger Giese. On Semantic Issues in Story Diagrams. In *Proceedings of the 4th International Fujaba Days*, 2006. 2, 38

[Tra11] Oleg Travkin. Kombination von Clustering- und musterbasierten Reverse-Engineering-Verfahren. Master's thesis, University of Paderborn, June 2011. In German. 9

[VB07] Daniel Varró and Andras Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214 – 234, 2007. Special Issue on Model Transformation. 41

[vDMT10] Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 299–300. ACM, May 2010. 39, 43

[WTF11] Claas Wilke, Michael Thiele, and Björn Freitag. Dresden OCL – Manual for Installation Use and Development. Technical report, TU Dresden, Germany, 2011. 38

[ZSW99] Albert Zündorf, Andy Schürr, and Andreas Winter. Story Driven Modeling. Technical Report tr-ri-99-211, Software Engineering Group, Universitiy of Paderborn, Germany, 1999. 1, 38

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. Habilitation, University of Paderborn, 2001. 1, 11, 37, 38

[Zün09] Albert Zündorf. Model Checking the Leader Election Protocol with Fujaba. In *5th International Workshop on Graph Based Tools*, pages 1–11, 2009. 25, 40

# Appendix A. Execution of Story Diagrams

In general, there are two possibilities to execute models: Executing them directly using an interpreter [GHS09] or generating GPL code, which is either compiled or interpreted.

FUJABA can generate Java or C code from story diagrams and their accompanying class diagrams. Here, a story diagram describes the behavior of a single method. Therefore, this method and its containing class must be defined first.

Interpreting a story diagram does not impose this restriction. In the following sections, we describe the structure and operation principles of an interpreter for story diagrams.

## A.1. Interpreting Story Diagrams

### A.1.1. Interpreter Architecture

Figure A.1 shows the package structure of the story diagram interpreter. Currently, there are multiple story diagram metamodels in use that must all be supported by the interpreter. Therefore, the interpreter is divided into a metamodel-independent core (*de.mdelab.sdm.interpreter.core*) and multiple metamodel-dependent extensions (*org.storydriven.modeling.interpreter* and *de.mdelab.sdm.interpreter.sde*). This separation of metamodel-dependent and independent parts allows for easier maintenance of the interpreter. The classes of the core package define a quite extensive list of generic type parameters (not shown in the subsequent class diagrams), e.g., for activity nodes, classifiers, or features. Subclasses in the metamodel-dependent packages replace these generic types with the concrete types defined in the respective metamodel.

Furthermore, those parts of the interpreter that depend on Eclipse are also separated (**.eclipse* packages). This allows to use the interpreter in stand-alone applications without Eclipse. In addition, the interpreters for expression languages like OCL are also separated (*de.mdelab.sdm.interpreter.ocl*). The story diagram interpreter provides an extension mechanism to add interpreters for other expression languages. The Eclipse-specific plug-ins provide the additional functionality that expression languages and interpreters for them are registered automatically using an extension point. Otherwise, the registration of expression languages must be performed explicitly by the interpreter user. This is the only difference between the

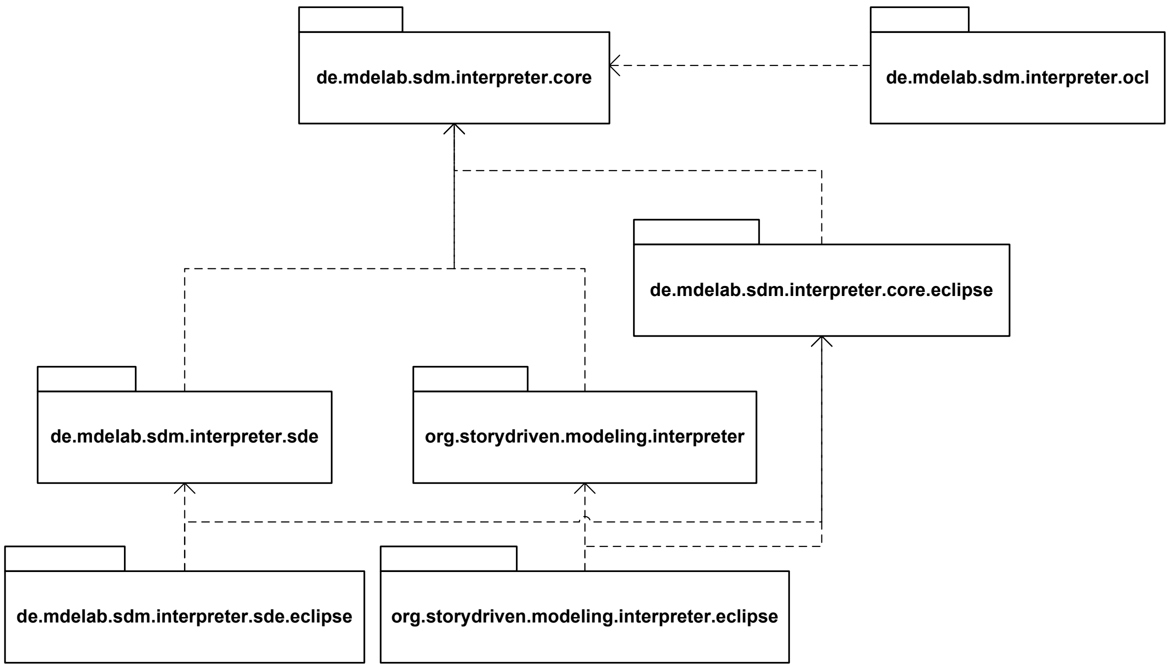

de.mdelab.sdm.interpreter.core
de.mdelab.sdm.interpreter.ocl
de.mdelab.sdm.interpreter.core.eclipse
de.mdelab.sdm.interpreter.sde
org.storydriven.modeling.interpreter
de.mdelab.sdm.interpreter.sde.eclipse
org.storydriven.modeling.interpreter.eclipse

Figure A.1.: Overview of the Packages of the Interpreter

core and the Eclipse-based plug-ins. Therefore, the **Eclipse* classes will not be explained in detail in the following sections.

#### A.1.1.1. Story Diagram Interpreter

Figure A.2 shows the main classes of the interpreter core. *SDMInterpreter* is the abstract superclass of all story diagram interpreters. It is responsible for the execution of a whole story diagram. *StoryDrivenInterpreter* and *StoryDrivenEclipseInterpreter* inherit from it to narrow the generic type parameters of *SDMInterpreter* to the specific types of the particular metamodel. The *SDMInterpreter* provides the *executeActivity()* method to execute a story diagram.

A *VariableScope* is a collection of *Variable*s that are valid in a specific scope. A *Variable* is a triple of the name, the classifier, and the value of the variable. The *SDMInterpreter* maintains multiple *VariableScope*s, one for each activity node.

The *ExpressionInterpreterManager* is responsible for managing the interpreters for expression languages and delegating the evaluation of an expression to the appropriate *Expression-Interpreter*. The *evaluateExpression()* method is provided for that purpose. Subclasses of *ExpressionInterpreter*s have to be registered at the *ExpressionInterpreterManager* via the *registerExpressionInterpreter()* method before expressions of that language can be handled, e.g., the *OCLExpressionInterpreter* has to be registerd for OCL expressions before OCL expressions in a story diagram can be evaluated. Similarly, the *CallsInterpreter* is registered for

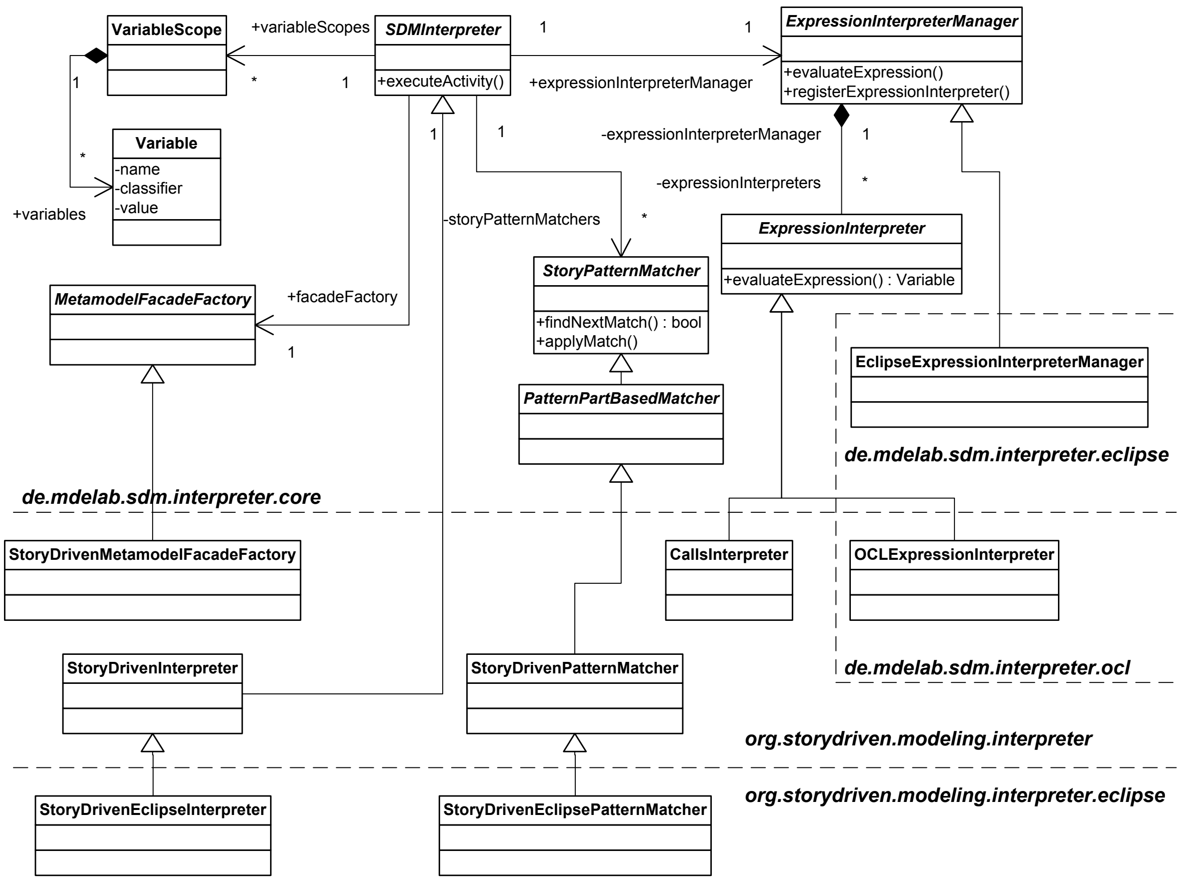

VariableScope
+variableScopes
SDMInterpreter
+executeActivity()
1
*
1
Variable
-name
-classifier
-value
+variables
ExpressionInterpreterManager
+evaluateExpression()
+registerExpressionInterpreter()
+expressionInterpreterManager
-expressionInterpreterManager
-expressionInterpreters
ExpressionInterpreter
+evaluateExpression() : Variable
-storyPatternMatchers
StoryPatternMatcher
+findNextMatch() : bool
+applyMatch()
PatternPartBasedMatcher
MetamodelFacadeFactory
+facadeFactory
EclipseExpressionInterpreterManager
de.mdelab.sdm.interpreter.eclipse
de.mdelab.sdm.interpreter.core
StoryDrivenMetamodelFacadeFactory
CallsInterpreter
OCLExpressionInterpreter
de.mdelab.sdm.interpreter.ocl
StoryDrivenInterpreter
StoryDrivenPatternMatcher
org.storydriven.modeling.interpreter
org.storydriven.modeling.interpreter.eclipse
StoryDrivenEclipseInterpreter
StoryDrivenEclipsePatternMatcher

Figure A.2.: Main classes of the interpreter core

Calls. The *EclipseExpressionInterpreterManager* performs this registration automatically. The plug-in *de.mdelab.sdm.interpreter.eclipse* defines an extension point for *ExpressionInterpreter*s. All interpreters extending this extension point are registered automatically by the *EclipseExpressionInterpreterManager*. If the interpreter is not used within Eclipse, *ExpressionInterpreter*s have to be registered explicitly before executing a story diagram.

The abstract class *ExpressionInterpreter* only defines the *evaluateExpression()* method that must be implemented by subclasses such as the *OCLExpressionInterpreter* and the *CallsInterpreter*. The method performs the execution of the expression, which may have side effects, and has to return a *Variable* with the return type and return value of the expression. In this method, the current *VariableScope* can also be accessed and modified so that variables of the story diagram can be used in expressions.

The interpreter often needs to access specific properties of story diagram elements, e.g., the name of elements or incoming and outgoing edges of activity nodes. While the interpreter core is metamodel-independent, it cannot access these properties directly but needs a facade for that purpose. The *MetamodelFacadeFactory* provides access to these facades. There are several interfaces for common kinds of story diagram elements (e.g., story nodes, junction nodes, object variables or link variables), which have to be implemented for specific story diagram metamodels. Subclasses of *MetamodelFacadeFactory* create the facades for the specific story diagram metamodel.

A *StoryPatternMatcher* is responsible for the execution of a single story pattern. This abstract superclass defines the methods *findNextMatch()* to search for the next match of a story pattern and *applyMatch()* to execute the graph transformation rule's side effects on the last match. The class *StoryPatternMatcher* does not implement a particular matching strategy, i.e., a particular algorithm how to search for matches of the pattern. This is done by *PatternPartBasedPatternMatcher*. This pattern matching strategy is explained in more detail in Section A.1.1.2.

#### A.1.1.2. Story Pattern Matcher

Figure A.3 shows the classes of the story pattern matcher. Currently, only one pattern matching strategy is implemented. The *PatternPartBasedPatternMatcher* splits the story pattern into multiple *PatternPart*s. What exactly constitutes a *PatternPart* is not specified in the interpreter core. This has to be implemented in the metamodel specific subclasses. Currently, the *StoryDrivenPatternMatcher* enforces the following semantics: A pattern part consists either of a single variable that has no incoming or outgoing links (*VariableOnlyPatternPart*), or of a single link and its adjacent object variables (*StoryDrivenLinkVariablePatternPart*, *StoryDrivenContainmentRelationPatternPart*, and *StoryDrivenPathPatternPart* depending on the kind of link). This implies that a variable can be contained in more than one pattern parts. This semantics can also be modified to support, e.g., complex application conditions or subpatterns, which form a distinct subunit of the pattern. But this remains transparent to the basic *PatternPartBasedPatternMatcher*. *MatchState*s are used by *PatternPart*s to temporarily store information about the current matching state, e.g., the iterator of a link to improve performance.

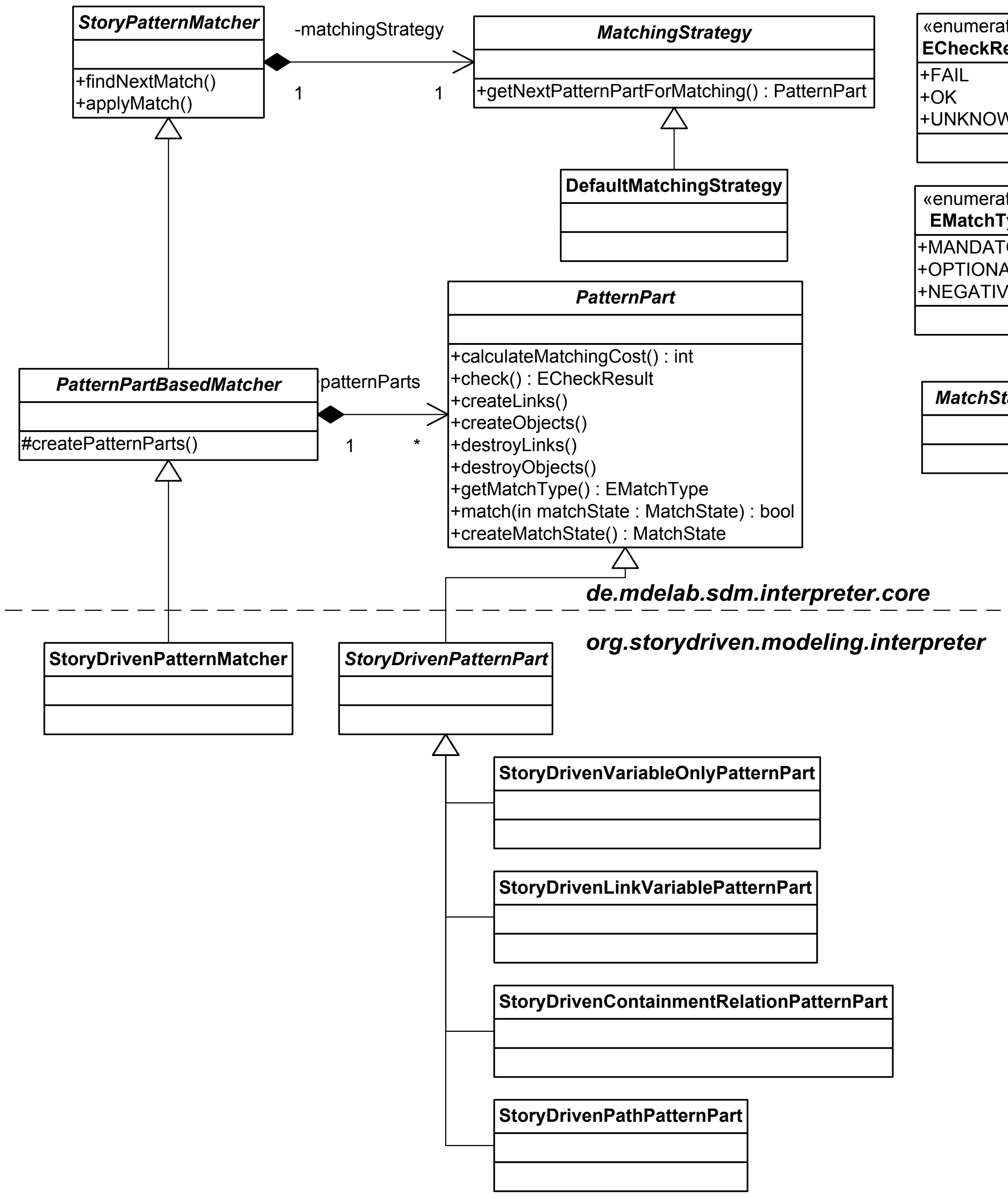

StoryPatternMatcher
+findNextMatch()
+applyMatch()
-matchingStrategy
1
1
MatchingStrategy
+getNextPatternPartForMatching() : PatternPart
DefaultMatchingStrategy
«enumeration»
ECheckResult
+FAIL
+OK
+UNKNOWN
«enumeration»
EMatchType
+MANDATORY
+OPTIONAL
+NEGATIVE
PatternPart
+calculateMatchingCost() : int
+check() : ECheckResult
+createLinks()
+createObjects()
+destroyLinks()
+destroyObjects()
+getMatchType() : EMatchType
+match(in matchState : MatchState) : bool
+createMatchState() : MatchState
PatternPartBasedMatcher
#createPatternParts()
-patternParts
1
*
MatchState
de.mdelab.sdm.interpreter.core
org.storydriven.modeling.interpreter
StoryDrivenPatternMatcher
StoryDrivenPatternPart
StoryDrivenVariableOnlyPatternPart
StoryDrivenLinkVariablePatternPart
StoryDrivenContainmentRelationPatternPart
StoryDrivenPathPatternPart

Figure A.3.: Main classes of the story pattern matcher

Subclasses are specific for pattern parts and have to define appropriate attributes, e.g., fields for iterator objects. More information can be found in Sec. A.1.3.2.

The *MatchingStrategy* determines the order in which pattern parts are matched. The *DefaultMatchingStrategy* matches pattern parts in the order of their matching cost estimates, i.e., *getNextPatternPartForMatching()* returns that pattern part with the lowest cost estimate.

There are also two additional pattern matching strategies: *DefaultMatchingStrategyWithLog* and *LogReproducingMatchingStrategy*. These are required for *for-each* story nodes. For more information, see Sec. A.1.2.

*PatternPart* defines several abstract methods:

1. *getMatchType()* returns whether matching the pattern parts is mandatory or optional, or whether the pattern part is a negative application condition (cf. Section 3.2.4.2).

2. *check()* checks whether the link exists in the instance graph, which requires that all variables of the pattern part are already bound to an instance object. If this is not the case, *check()* returns *UNKNOWN*.

3. *calculateMatchingCost()* provides an estimate of the cost to match a variable using the link of the pattern part. This estimate can be based, e.g., on the number of elements contained in the link. If it is currently not possible to match this pattern part (e.g., because all variables of the pattern part are still unbound), *-1* is returned. This operation is called by the *MatchingStrategy* to select the *PatternPart* that the pattern matcher should use to match the next variable.

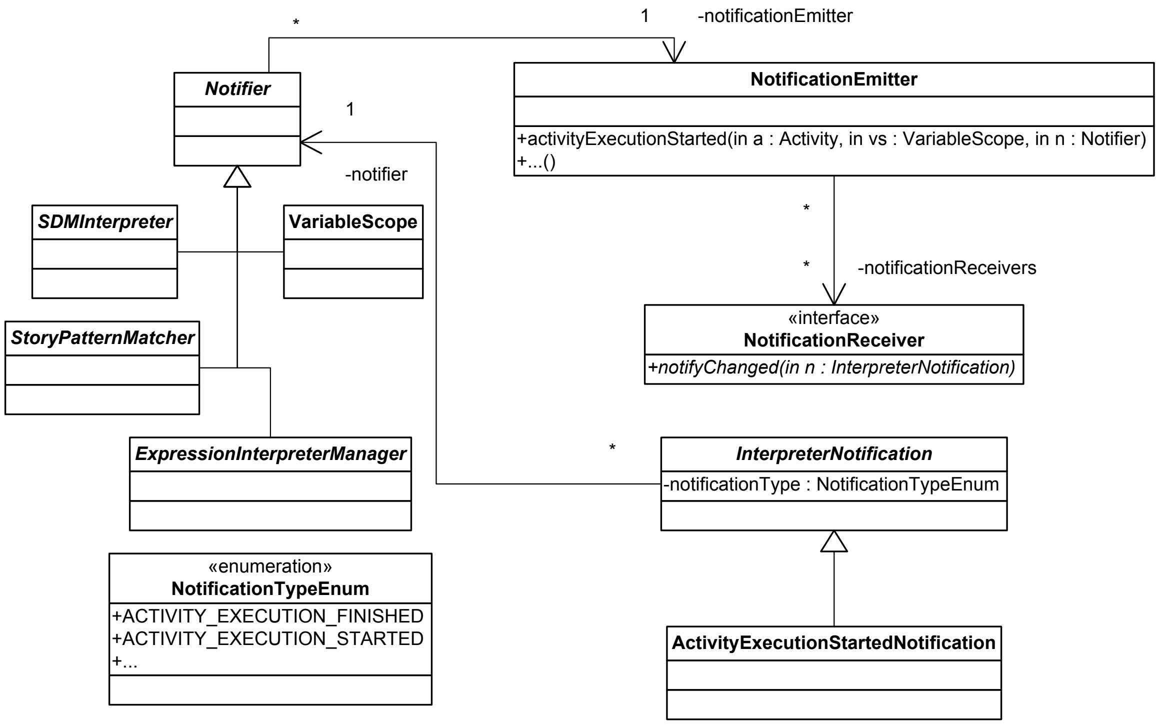
4. *createMatchState()* creates a *MatchState* object that is by the *match()* operation to store information about the matching process.

5. *match(MatchState matchState)* implements the pattern matching for this kind of pattern part. It is called after *calculateMatchingCost()*. To find a match, at least one variable of the pattern part has to be bound and at least one has to be unbound. Then, *match()* tries to find matches for all unbound variables. This part of the pattern matching algorithm is highly implementation specific. It is not only different for different metamodels, it also has to be implemented differently for different kinds of link variables. For example, matching an object via an ordinary *LinkVariable* has to be done differently than matching via a *Path* or a *ContainmentRelation*. However, this also allows to exploit certain features of the metamodel to improve execution performance. For example, *StoryDrivenContainmentRelationPatternPart* uses EMF's *eContainer()* method to navigate containment links in the opposite direction. The *matchState* parameter can be used to store information about the matching process, e.g., the iterator object of the link.

6. *createLinks()* and *createObjects()* create those elements of the pattern part, that are marked with «`create`».

7. *destroyLinks()* and *destroyObjects()* destroy links and objects. In contrast to the creation of elements, these steps are separated to ensure an orderly deletion of story pattern variables in the *VariableScope*. [1]

### A.1.1.3. Notification Mechanism



*
1
-notificationEmitter
Notifier
NotificationEmitter
1
+activityExecutionStarted(in a : Activity, in vs : VariableScope, in n : Notifier)
+...()
-notifier
SDMInterpreter
VariableScope
*
*
-notificationReceivers
«interface»
NotificationReceiver
+notifyChanged(in n : InterpreterNotification)
StoryPatternMatcher
ExpressionInterpreterManager
*
InterpreterNotification
-notificationType : NotificationTypeEnum
«enumeration»
NotificationTypeEnum
+ACTIVITY_EXECUTION_FINISHED
+ACTIVITY_EXECUTION_STARTED
+...
ActivityExecutionStartedNotification

Figure A.4.: Relevant classes of the interpreter's notification mechanism

The interpreter and its subcomponents provide a notification mechanism to inform clients of all important steps during the execution of a story diagram. This is an implementation of the observer design pattern.

*SDMInterpreter*, *StoryPatternMatcher*, *VariableScope*, and *ExpressionInterpreterManager* extend the *Notifier* superclass, see Fig. A.4. *Notifier* defines a reference to a *NotificationEmit-*

[1]Background: There are two ways to execute story patterns with deleted elements: Destroy all links first and then all objects, or the other way round. EMF also supports unidirectional references. Therefore, deleting objects as implemented in *EcoreUtil.delete()* is done by going from the deleted object to the root of the containment hierarchy (usually the *Resource* or *ResourceSet*) and searching for cross-references to the deleted object. If the links are deleted first when the story pattern is executed, the destroyed object may be removed from its containment hierarchy (if a destroyed link represents this containment). After that, existing cross-references to the destroyed object that are not represented by a link in the story pattern (remember that story patterns have SPO semantics) cannot be found and deleted. For this reason, the interpreter first deletes all objects and then all links.

*ter*. This class provides an operation for each type of notification defined in *NotificationType-Enum* that creates an *InterpreterNotification* and forwards it to all registered *NotificationReceivers* by calling their *notifyChanged()* operations. Clients can add their own *NotificationReceiver*s to the *NotificationEmitter*'s list of receivers.

By default, each *Notifier* uses the default implementation in *NotificationEmitter*. However, it is also possible to create *Notifiers* with custom implementations of *NotificationEmitter* to directly process notifications there or process notifications asynchronously, for example.

### A.1.2. Interpreting Story Diagrams

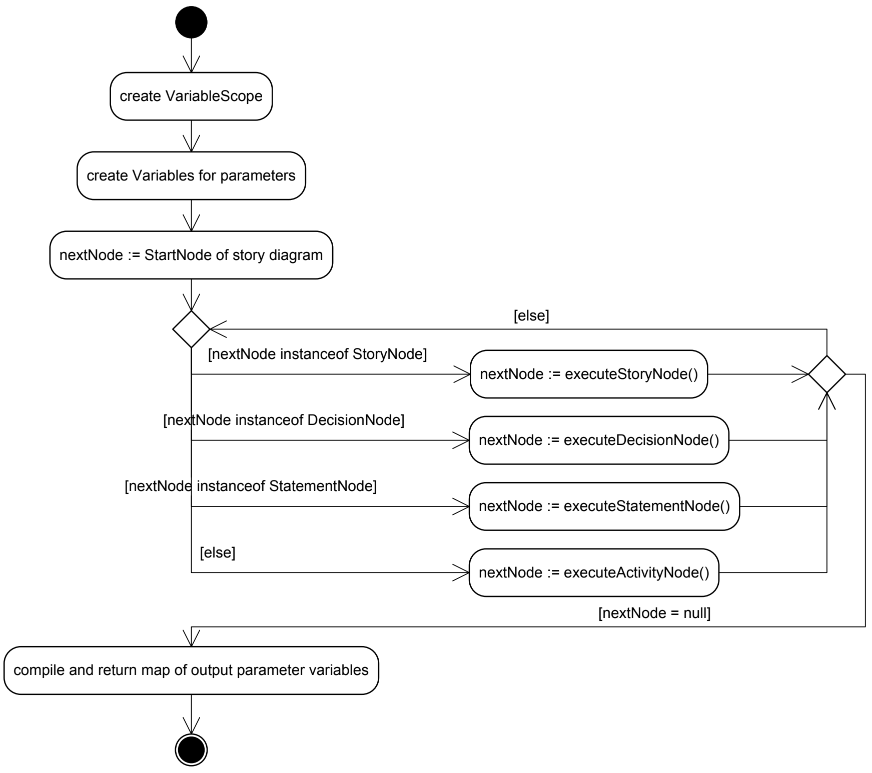

create VariableScope
create Variables for parameters
nextNode := StartNode of story diagram
[else]
[nextNode instanceof StoryNode]
nextNode := executeStoryNode()
[nextNode instanceof DecisionNode]
nextNode := executeDecisionNode()
[nextNode instanceof StatementNode]
nextNode := executeStatementNode()
[else]
nextNode := executeActivityNode()
[nextNode = null]
compile and return map of output parameter variables

Figure A.5.: Execution Scheme of the *SDMInterpreter*

The overall interpretation of a story diagram is a simple graph traversal algorithm. The interpretation starts at the story diagram's *StartNode* and traverses the story diagram until it reaches a *StopNode*. All activity nodes are executed by specialized methods, which return the next activity node to execute afterwards. The activity diagram in Figure A.5 shows the overall execution scheme of the interpreter.

The interpreter is started with the *executeActivity()* method. This method creates the root *VariableScope* and a *Variable* for each parameter of the story diagram. Then, the *StartNode* of the story diagram is obtained and executed.

In general, the execution of activity nodes works as follows: First, the kind of the activity node is determined. Then, it is executed by the appropriate execution method. *StoryNode*s, *JunctionNode*s, *StopNode*s, and *StatementNode*s require special handling by distinct execution methods. All other kinds of activity nodes are skipped. After execution of a node, the next node to execute is returned by the execution methods. This process is repeated until a *StopNode* is reached, which has no subsequent nodes. Then, the loop terminates. The return value expressions of all outgoing parameters are evaluated and put into a map, which is returned by *executeActivity()*. This map maps the parameter names to their values.[2]

A non-for-each *StoryNode* is executed using the *StoryPatternMatcher* (cf. Section A.1.1.2) with the *DefaultMatchingStrategy*. It searches for a match of the story pattern and applies the graph transformation rule if a match was found. For for-each nodes, the process is more complex. If the story pattern is executed for the first time, a new *StoryPatternMatcher* is created and stored in a local map for this *StoryNode*. The pattern matcher is executed with the *DefaultMatchingStrategyWithLog*, which keeps a log of the order in which elements were matched. If a match was found, the next activity node of the loop body is returned, i.e., that activity node that is connected to the for-each node via a for-each edge. The interpreter executes that node and eventually the control flow returns to the for-each node. Now, the existing *StoryPatternMatcher* is reused so that it continues pattern matching where it left off. This time, however, the pattern matcher uses the *LogReproducingMatchingStrategy*. This ensures, that all elements are matched in the same order as the first time. For-each nodes are executed with the *fresh match* semantics. After a match was found, the story pattern's side effects are executed immediately. Then, the next match is sought. Side-effects may influence subsequent matches, they may create new matches or eliminate existing ones. They may also influence they matching order if they change the number of elements in references, which changes the cost estimates (for details, see Section A.1.3). If the *DefaultMatchingStrategy* would be used in each iteration of the for-each node, it may choose a different matching order in subsequent iterations. Due to the way how previous matches are managed, this may cause the pattern matcher to return a match multiple times or skip valid matches. Therefore, the *DefaultMatchingStrategyWithLog* is used in the first iteration of a for-each story node to log the matching order, and the *LogReproducingMatchingStrategy* is used in all subsequent iterations, which matches elements in exactly the same order.

The stored mapping between the *StoryNode* and the pattern matcher is discarded after the last loop iteration. If the story diagram's control flow returns to the for-each node again, the pattern matching process starts anew.

In addition to the *fresh match* semantics, it is also possible to add other execution semantics for for-each nodes, e.g., *pre-select*, which searches for all matches before executing side-effects.

[2]For backward compatibility, all variables of the story diagram are currently returned, not only parameters.

## A.1.3. Interpreting Story Patterns

The execution of a single story pattern comprises three steps: Initialization of the pattern matcher and analysis of the story pattern (Section A.1.3.1), pattern matching (Section A.1.3.2), and pattern application (Section A.1.3.3). These steps are executed in the constructor of the pattern matcher, the *findNextMatch()*, and the *applyMatch()* operations respectively. *findNextMatch()* can be called successively to return all matches for a story pattern one-by-one. These operations are separated, to allow for additional operations between these phases by the user of the pattern matcher. The pattern matcher can also be used without the story diagram interpreter to execute a single story pattern.

### A.1.3.1. Initialization and Pattern Analysis

A *StoryPatternMatcher* is instantiated for a specific story pattern. Therefore, the *StoryPatternMatcher*'s constructor already requires the story pattern as a parameter. In the constructor, the matcher's internal data structures are set up. These comprise lists of the bound and unbound pattern variables[3], checked and unchecked *PatternParts*, bound instance objects, the matching history, and the stack of match transactions. The matching history is a mapping between pattern variables and lists of instance objects, that were previously bound to that pattern variable. The match transaction stack is a stack that contains stack elements for each relevant action of the matcher. Each time, a pattern part is matched or checked, or a pattern variable is bound to an object, a match transaction is executed and pushed onto the stack. A transaction usually involves a manipulation of the internal data structures of the matcher, e.g., moving an element from the list of unbound to that of bound pattern variables. When the matcher has to step back, these match transactions are popped from the stack and rolled back. After initializing these data structures, the story pattern is divided into pattern parts as described in Sec. A.1.1.2.

### A.1.3.2. Pattern Matching

*findNextMatch()* is responsible for searching for the next valid match of the story pattern in the instance model. The operation returns a boolean value indicating whether a match was found or not. If a match was found, the pattern matcher's *VariableScope* is manipulated accordingly. If no match was found, the *VariableScope* is left untouched. Fig. A.6 shows the overall scheme of this method.

First, a new variable scope is created, which is a child of the current variable scope. During pattern matching, this child variable scope is modified but its parent is left untouched. Next, all pattern variables that are marked as bound are bound to the appropriate variables in the *VariableScope*. Pattern variables with binding expressions are also handled here. After that, all unchecked pattern parts are checked. After binding all pattern variables, some pattern parts may contain only bound objects. These pattern parts can be checked already at this point. If

[3]Subsequently, we use the term *pattern variable* to refer to object variables or primitive variables in a story pattern in contrast to *Variables* to refer to *Variable* objects used internally by the pattern matcher.

```
create child variable scope of current variable scope;

bind bound objects;

check unchecked pattern parts;

boolean match = true;

do {
  while (nextPatternPart =
      matchingStrategy.getNextPatternPart() != null) {

    commit matchPatternPart transaction;
    match = nextPatternPart.match();

    if (not match) {
      roll back last two matchPatternPart transactions;

      if (matchingStack is empty)
        break;
    }
  }

  if (match) {
    match = checkStoryPatternConstraints();
    if (not match)
      roll back last matchPatternPart transaction;
  }
} while (matchingStack is not empty and not match)

if (match) {
   merge child variable scope into its parent scope;
}
```

Figure A.6.: Overall pattern matching algorithm

a check fails, there can be no match for the story pattern and the pattern matcher terminates. Otherwise, the actual pattern matching algorithm starts.

In two nested loops, the matching strategy returns the next pattern part to use for matching. The matching strategy may use arbitrary heuristics to choose a pattern part from the list of unchecked pattern parts. The default strategy returns that pattern part with the lowest cost estimate. This pattern part must contain at least one bound and one unbound object and it must be possible to navigate from the bound objects to the unbound objects. The *calculateMatchingCost()* operation used in both metamodel specific implementations checks this. A transaction for matching the current pattern part is pushed on the stack and the pattern part's *match()* operation is called. This operation is specific to the actual type of the link of the pattern. In case of ordinary *LinkVariables*, *match()* simply follows the link from the bound instance object and tries to bind the unbound pattern variable to an instance object of the object graph.

If matching the pattern part was successful, i.e. *match()* returned true, the loop continues with the next pattern part. When all pattern parts have been matched, the matching strategy returns null instead of a pattern part. Now, all constraints are checked that are defined for the whole story pattern. If these conditions are also satisfied, a valid match has been found. Now, the child variable scope is merged into its parent scope to persist the match, i.e. the *Variable*s created for the matched objects are merged into the parent scope so that the caller of the pattern matcher can access the matched objects.

In case the *match()* operation did not find a match, the last transactions including the one committed in line 13 of Fig. A.6 for matching pattern parts have to be rolled back. Of course, this also rolls back all bindings of pattern variables that were performed in the meantime. Here, also the second last transaction has to be rolled back because the pattern variable matched in that transaction has now shown to be an invalid match and a new match has to be found for it. A roll back is also performed if the constraints defined on the whole story pattern are not satisfied. If a roll back leads to an empty matching stack, the pattern matching process is terminated because no valid match exists.

#### A.1.3.3. Pattern Application

After finding a match for a story pattern, the story pattern can be applied, i.e. its side-effects can be executed. This is done in *applyMatch()*, which must be called be the caller of the pattern matcher explicitly. First, all attribute assignments are evaluated and their results are stored in a list. After that, all objects marked as «`destroy`» are deleted and then all links. Finally, all objects and links marked for creation are created and all attributes are assigned their new values.

It is important that all objects are deleted before deleting any links. Object deletion is performed via the *EcoreUtil.delete()* operation, which also deletes all cross-references to the deleted object. To do so, this operation walks the containment hierarchy upwards and searches for cross-links in the whole model tree. If the pattern matcher would delete links first, it might not be possible to walk the containment hierarchy upwards if the deleted link pointed to the container of a deleted object. Then, it would be impossible to delete all cross-references.

Attribute assignment expressions are evaluated at the beginning to support expressions that refer to deleted elements. If attribute assignments would be evaluated at the end, it would not be possible to evaluate expressions with references to deleted elements because these do not exist anymore in the *VariableScope*.

# Appendix B. Technical Reference

## B.1. Package `modeling`

### B.1.1. Package Overview

The modeling package is the root package for the story diagram meta-model. It defines several abstract super classes which implement an extension mechanism as well as recurring structural features like, e.g., names of elements. The classes in this package are intended to be subclassed by any meta-model element.

### B.1.2. Detailed Contents Documentation

#### B.1.2.1. Class `CommentableElement`

**Overview** Abstract super class for all meta-model elements that may carry a comment in form of a string.

**Class Properties** Class `CommentableElement` has the following properties:

**comment : EString [0..1]**

The comment string that can be used to attach arbitrary information to CommentableElements.

**Parent Classes**

- ExtendableElement see Section B.1.2.2 on Page 65

#### B.1.2.2. Class `ExtendableElement`

**Overview** Abstract base class for the whole story diagram model. The ExtendableElement specifies the extension mechanism that can be used to extend an object by an Extension containing additional attributes and references.

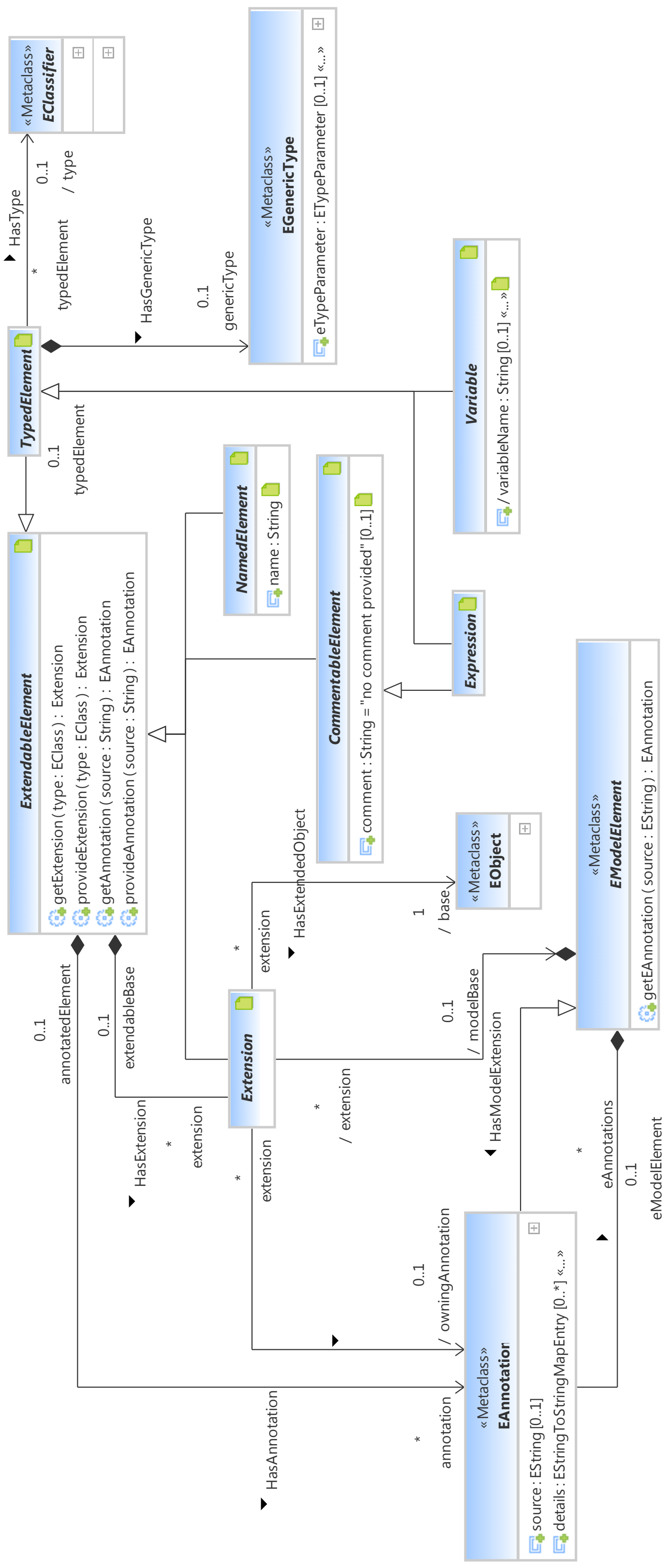


Figure B.1.: Metamodel of the modeling Package

#### Parent Classes

- EObject

### B.1.2.3. Class `Extension`

**Overview** Abstract super class for an Extension that can be defined for an object.

#### Parent Classes

- ExtendableElement see Section B.1.2.2 on Page 65

### B.1.2.4. Class `NamedElement`

**Overview** Abstract super class for all meta-model elements that carry a name.

**Class Properties** Class `NamedElement` has the following properties:

**name : EString**

The name attribute of a meta-model element.

#### Parent Classes

- ExtendableElement see Section B.1.2.2 on Page 65

### B.1.2.5. Class `TypedElement`

**Overview** Abstract super class for all meta-model elements that are typed by means of an EClassifier or an EGenericType.

#### Parent Classes

- ExtendableElement see Section B.1.2.2 on Page 65

### B.1.2.6. Class `Variable`

**Overview** Represents a variable which can be, for example, an object variable, an attribute, or any other kind of variable.

**Class Properties** Class `Variable` has the following properties:

**/variableName : EString [0..1]**

A variable is identified by its variable name.

#### Parent Classes

- TypedElement see Section B.1.2.5 on Page 67

## B.2. Package `modeling::activities`

### B.2.1. Package Overview

This package contains everything to model activities: the different kinds of activity nodes and edges as well as guards.

### B.2.2. Detailed Contents Documentation

#### B.2.2.1. Class `Activity`

**Overview** The diagram that describes the control flow of an operation. It is used to structure a number story patterns into a stroy diagram. Story patterns are contained in activity nodes which are connected by activity edges. In addition, there are special nodes like start, stop, and juction nodes.

**Parent Classes**

- Callable see Section B.4.2.1 on Page 78,
- NamedElement see Section B.1.2.4 on Page 67

#### B.2.2.2. Class `ActivityCallNode`

**Overview** The ActivityCallNode is a special ActivityNode which represents the calling of another story diagram within an activity. To support polymorphic dispatching, multiple activities can be assigned to it (all of which must have the same call signature, i.e. matching in and out parameters). All assigned activities are then called in the given order and the first one whose precondition is fulfilled is executed (Chain of Responsibilty).

**Parent Classes**

- ActivityNode see Section B.2.2.4 on Page 71,
- Invocation see Section B.4.2.2 on Page 78

#### B.2.2.3. Class `ActivityEdge`

**Overview** The ActivityEdge represents the control flow in an activity. It is a dericted connection from one activity to another one. There exist different kinds of activity edges which are differentiated by the guard attribute.

**Class Properties** Class `ActivityEdge` has the following properties:

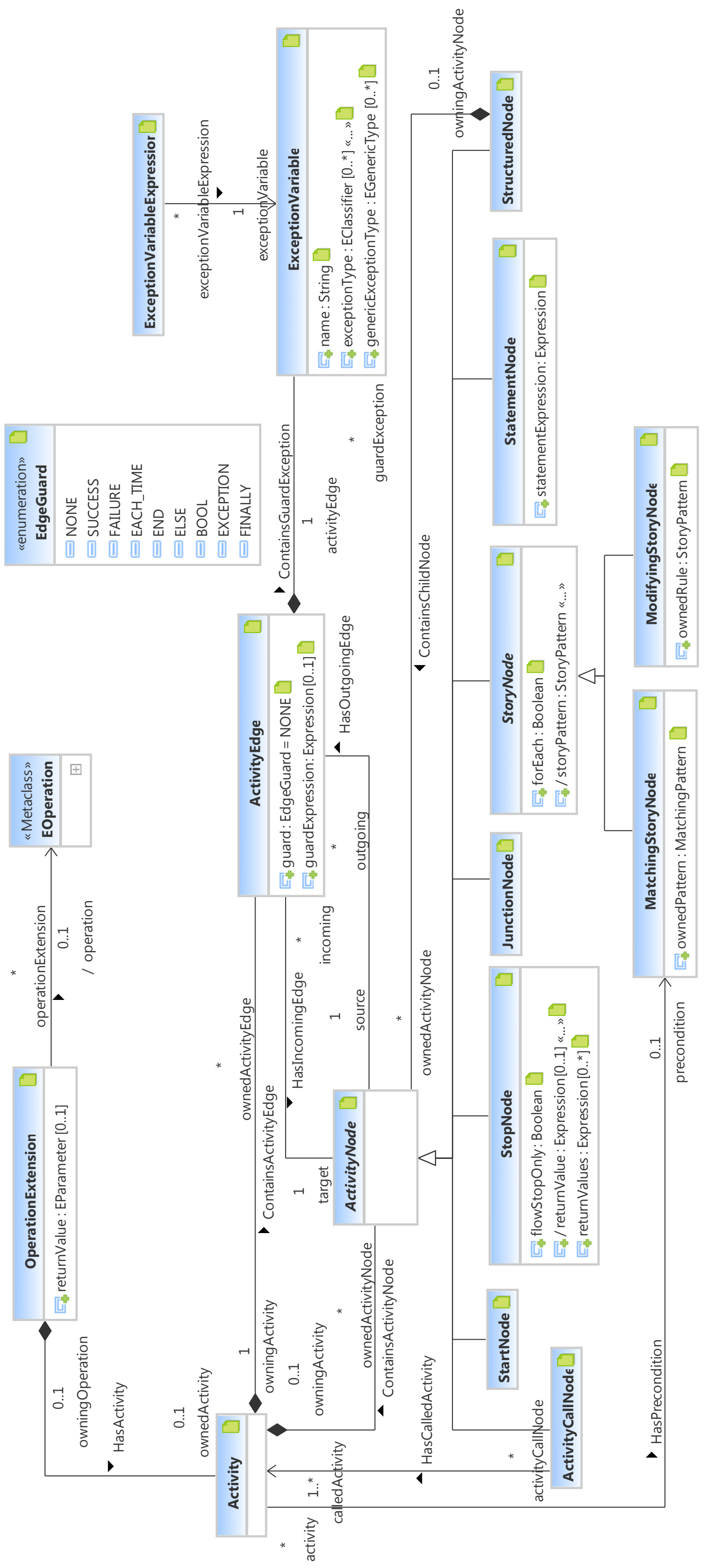


Figure B.2.: Metamodel of the activities Package

**guard : EdgeGuard** see Section B.2.2.5 on Page 71

The guard defines the kind of the activity edge. The possible kinds of guards are specified by the EdgeGuard enum.

**Class References** Class `ActivityEdge` has the following references:

**guardException : ExceptionVariable [0..*]** see Section B.2.2.6 on Page 73

Declares variables representing the Exceptions that lead to firing this transition.

**guardExpression : Expression [0..1]** see Section B.6.2.7 on Page 85

Points to an expression in case the transition guard is BOOL. The expression has to evaulate to a boolean value.

**owningActivity : Activity** see Section B.2.2.1 on Page 69

Points to the activity this ActivityEdge is contained in.

**source : ActivityNode** see Section B.2.2.4 on Page 71

The source node of this ActivityEdge.

**target : ActivityNode** see Section B.2.2.4 on Page 71

The target node of this ActivityEdge.

#### Parent Classes

- ExtendableElement see Section B.1.2.2 on Page 65

### B.2.2.4. Class `ActivityNode`

**Overview** Abstract super class for all kinds of nodes that may be added to an activity. This class provides the basic functionality of connecting the activity nodes in the activity by ActivityEdges.

#### Parent Classes

- NamedElement see Section B.1.2.4 on Page 67,
- CommentableElement see Section B.1.2.1 on Page 65

### B.2.2.5. Enumeration `EdgeGuard`

**Overview** This enum is used to model different kinds of activity edges.

**Enum Properties** Enumeration `EdgeGuard` has the following literals:

**NONE = 0**

No guard, only one outgoing activity edge of this kind is supported per activity node. If an edge with EdgeGuard NONE is used, it must be the only edge leaving a state.

**SUCCESS = 1**

Edge will be taken if execution of the souce activity node was successful, e.g., a story pattern was matched successfully. There must be another edge leaving the same node which is of kind FAILURE.

**FAILURE = 2**

Edge will be taken if execution of the source activity node was not successful, e.g., a story pattern could not be matched. There must be another edge leaving the same node which is of kind SUCCESS

**EACH_TIME = 3**

Edge may only leave a StoryNode whose forEach attribute is true. It will be taken for each match that can be identified for the story pattern in the foreach StoryNode. There must be another edge leaving the same node which is of kind END

**END = 4**

Edge may only leave a StoryNode whose forEach attribute is true. It will be taken if no more fresh matches for the story pattern in the foreach node can be found.

**ELSE = 5**

Complement to the BOOL guard, ELSE may only be used if at least one BOOL activity edge leaves the same state. The edge will be taken if none of the BOOL guards can be evaluated to true

**BOOL = 6**

An activity edge specifying a boolean guard using variables that have been previously used in the activity. Edge will be taken if the guardExpression of the activity edge evaluates to true. More than one BOOL edge is allowed to leave an activity node.

**EXCEPTION = 7**

An EXCEPTION edge will be taken if an exception of the type defined by the ExceptionVariable connected to the activity edge occured while executing the source activity node of the edge. More than one edge of kind EXCEPTION is allowed to leave a node.

**FINALLY = 8**

An activity edge of kind FINALLY may only leave an activity node that has at least one other outgoing edge of kind EXCEPTION. The finally edge will be taken after the source node has been executed and after, possibly, the EXCEPTION edge has been taken.

#### B.2.2.6. Class `ExceptionVariable`

**Overview** Declares a variable representing an Exception that leads to firing a transition (ActivityEdge). Can only be applied to ActivityEdge whose guard is set to EXCEPTION.

**Class Properties** Class `ExceptionVariable` has the following properties:

**name : EString**

Specifies the name of the declared exception variable.

**Class References** Class `ExceptionVariable` has the following references:

**activityEdge : ActivityEdge** see Section B.2.2.3 on Page 69

Specifies the transition (activity edge) where the exception variable is declared.

**exceptionType : EClassifier [0..$*$]**

Specifies the type of the declared exception variable.

**genericExceptionType : EGenericType [0..$*$]**

Allows the use of generics for the declaration of exception types.

**Parent Classes**

- Variable see Section B.1.2.6 on Page 67

#### B.2.2.7. Class `JunctionNode`

**Overview** A JunctionNode represents a pseudo-activity which is used for branching and merging the control flow in an activity. It is visualized by a diamond shaped figure.

**Parent Classes**

- ActivityNode see Section B.2.2.4 on Page 71

#### B.2.2.8. Class `MatchingStoryNode`

**Overview** A MatchingStoryNode may only contain a MatchingPattern which does not change the graph. I.e., no element contained in this activity carries a create or destroy annotation. Thus, after executing a MatchingStoryNode, the underlying graph is guaranteed to be unchanged.

#### Parent Classes

- StoryNode see Section B.2.2.14 on Page 75

### B.2.2.9. Class `ModifyingStoryNode`

**Overview** A ModifyingStoryNode contains a story pattern which may change the underlying graph upon execution.

#### Parent Classes

- StoryNode see Section B.2.2.14 on Page 75

### B.2.2.10. Class `OperationExtension`

**Overview** An OperationExtension is a stand-in for an EOperation in our model. It is necessary because we cannot change the type EOperation. Thus, OperationExtension points to an EOperation but adds the reference to an Activity that describes the operations behavior.

#### Parent Classes

- Extension see Section B.1.2.3 on Page 67,
- Callable see Section B.4.2.1 on Page 78

### B.2.2.11. Class `StartNode`

**Overview** The start node of an activity defines the starting point for the execution of the activity.

#### Parent Classes

- ActivityNode see Section B.2.2.4 on Page 71

### B.2.2.12. Class `StatementNode`

**Overview** A statement node is a node that just contains an expression defining its behavior. In combination with a textual expression, arbitrary souce code might be added by using StatementNodes.

#### Parent Classes

- ActivityNode see Section B.2.2.4 on Page 71

### B.2.2.13. Class `StopNode`

**Overview** At a StopNode, the execution of an activity terminates. If the activity specifies any out-parameters, they have to be bound to a return expression.

**Class Properties** Class `StopNode` has the following properties:

**flowStopOnly : EBoolean**

true if subactivity is stopped, but not the whole control flow

**Class References** Class `StopNode` has the following references:

**/returnValue : Expression [0..1]** see Section B.6.2.7 on Page 85

Convenience method when dealing with activities that implement an EOperation. In this case, only one out parameter is supported. This attributes then returns the first out parameter.

**returnValues : Expression [0..∗]** see Section B.6.2.7 on Page 85

Defines the return values of the activity. These return values will be assigned to the out-parameters.

#### Parent Classes

- ActivityNode see Section B.2.2.4 on Page 71

### B.2.2.14. Class `StoryNode`

**Overview** An activity node containing a story pattern.

**Class Properties** Class `StoryNode` has the following properties:

**forEach : EBoolean**

Specifies whether just one match should be found for the contained pattern (forEach = false) or whether all matches should be found (forEach = true).

**Class References** Class `StoryNode` has the following references:

**/storyPattern : StoryPattern** see Section B.7.2.18 on Page 97

#### Parent Classes

- ActivityNode see Section B.2.2.4 on Page 71

### B.2.2.15. Class `StructuredNode`

**Overview** A structured node is a node that contains several other activities.

### Parent Classes

- ActivityNode see Section B.2.2.4 on Page 71

## B.3. Package `modeling::activities::expressions`

### B.3.1. Package Overview

This package offers expressions that can be used in activities, e.g., exceptions triggered by activity edges.

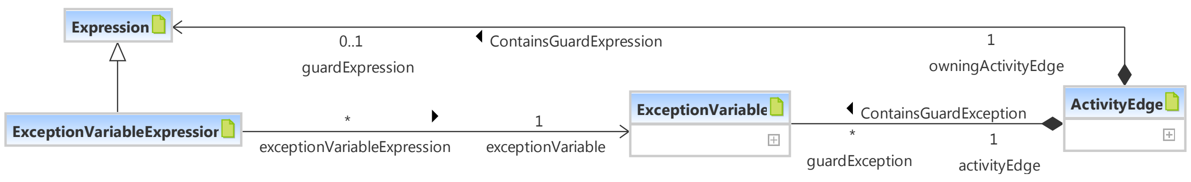


Figure B.3.: Metamodel of the activities::expressions Package

### B.3.2. Detailed Contents Documentation

#### B.3.2.1. Class `ExceptionVariableExpression`

**Overview** Represents the value of an exception variable declared as a transition guard (the guard of an activity edge).

**Parent Classes**

- Expression see Section B.6.2.7 on Page 85

## B.4. Package `modeling::calls`

### B.4.1. Package Overview

This package contains all classes for modeling calls to activities and EOperations from within an activity.

### B.4.2. Detailed Contents Documentation

#### B.4.2.1. Class `Callable`

**Overview** An entity which can be called by an Invocation. A Callable can have a number of (ordered) parameters which are either in or out parameters. In the case of activities, the number of in and out parameters is unbounded, whereas OperationExtensions and OpaqueCallables can only have one out parameter (This is enforced by an OCL constraint).

**Parent Classes**

- CommentableElement see Section B.1.2.1 on Page 65

#### B.4.2.2. Class `Invocation`

**Overview** Superclass for invocations of behavior which is specified elsewhere, e.g. in methods (MethodCallExpression) or activities (ActivityCallNode). An invocation has one parameter binding for each parameter (in or out) of the called method/activity. For Callables which are contained in the model (i.e. Activities and OperationExtensions) the Invocation directly points to the callee. OpaqueCallables are directly referenced by (and contained in) the MethodCallExpressions.

**Parent Classes**

- CommentableElement see Section B.1.2.1 on Page 65

#### B.4.2.3. Class `OpaqueCallable`

**Overview** An OpaqueCallable represents an external method which is not explicitly modeled (e.g. a method in an external library). Because it is not contained anywhere in the model it is directly referenced by and contained in the MethodCallExpression.

**Class Properties** Class `OpaqueCallable` has the following properties:

**name : EString**

The name of the Callable.

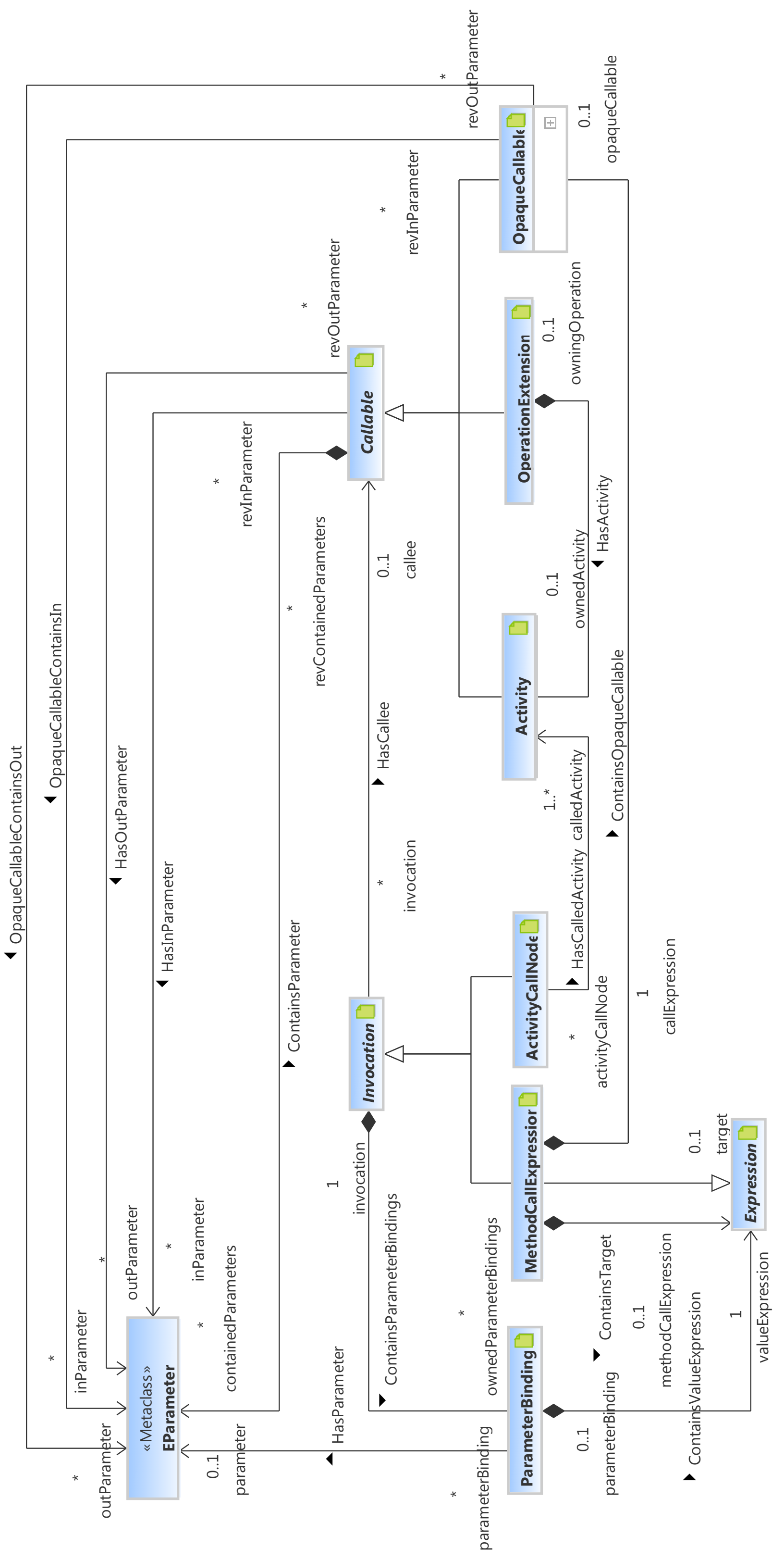


Figure B.4.: Metamodel of the calls Package

**Class References** Class `OpaqueCallable` has the following references:

**callExpression : MethodCallExpression** see Section B.5.2.1 on Page 81

An expression that specifies which method should be called by the Opaque-Callable.

#### Parent Classes

- Callable see Section B.4.2.1 on Page 78

#### B.4.2.4. Class `ParameterBinding`

**Overview** Binds a parameter to a certain value for a given invocation. The value of the parameter is represented by an expression.

#### Parent Classes

- CommentableElement see Section B.1.2.1 on Page 65

#### B.4.2.5. Class `ParameterExtension`

**Overview** Represents an EParameter and adds functionality to it, especially beiing subtype of Variable.

#### Parent Classes

- Variable see Section B.1.2.6 on Page 67,
- Extension see Section B.1.2.3 on Page 67

## B.5. Package `modeling::calls::expressions`

### B.5.1. Package Overview

This package offers expressions to describe, e.g., method calls and the corresponding arguments.

### B.5.2. Detailed Contents Documentation

#### B.5.2.1. Class `MethodCallExpression`

**Overview** A MethodCallEpression represents the direct invocation of a method. This can either be a method which is explicitly modeled as an EOperation in a class diagram (referenced by the OperationExtension) or an unmodeled method in an external library (referenced by an OpaqueCallable). Therefore, a MethodCallExpression references either an OperationExtension (indirectly via the callee role between Invocation and Callable) or an OpaqueCallable.

**Parent Classes**

- Expression see Section B.6.2.7 on Page 85,
- Invocation see Section B.4.2.2 on Page 78

#### B.5.2.2. Class `ParameterExpression`

**Overview** An Expressions that represents a parameter value, e.g. the value of an Activity's parameter.

**Parent Classes**

- Expression see Section B.6.2.7 on Page 85

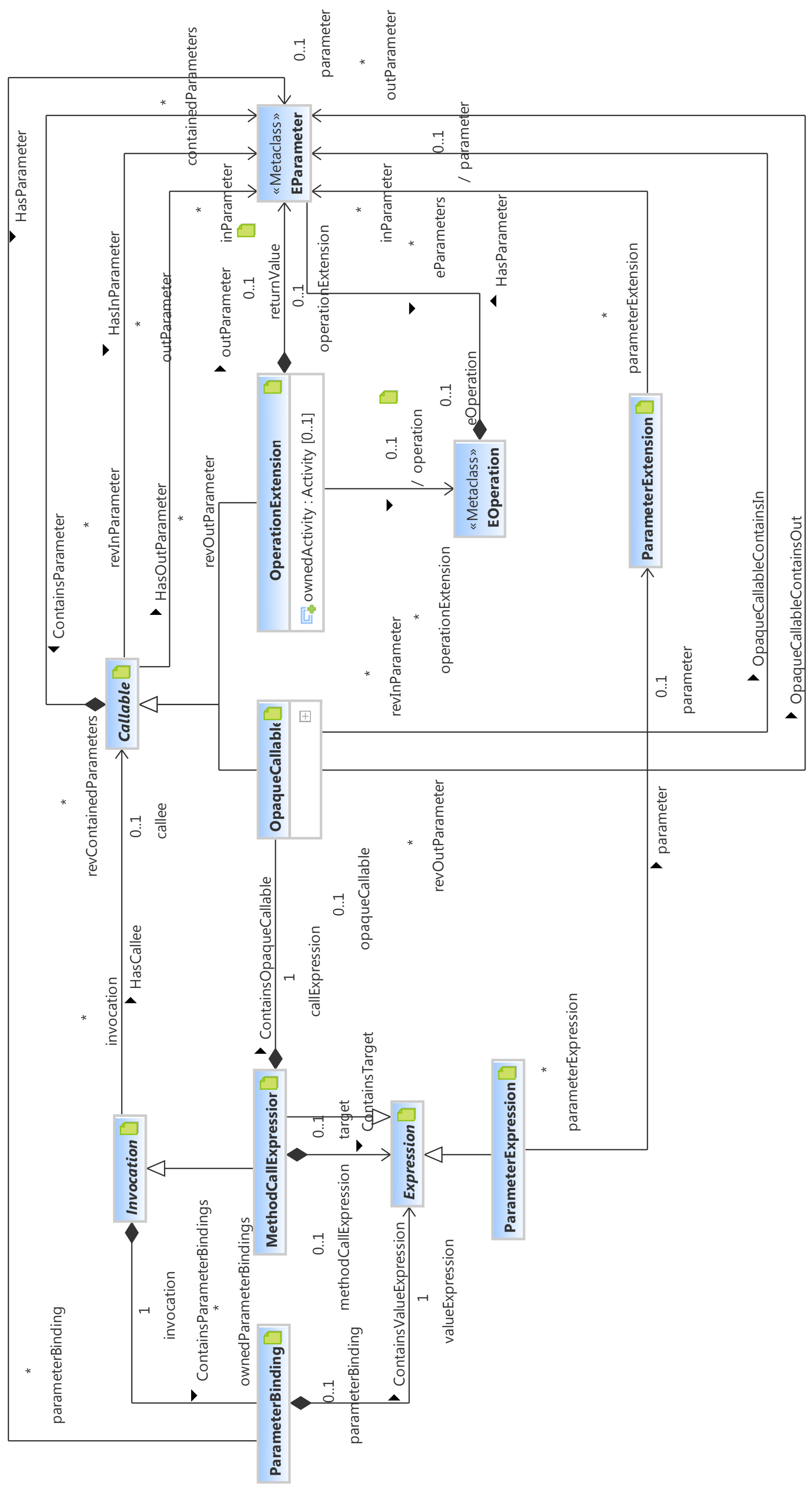


Figure B.5.: Metamodel of the calls::expressions Package

## B.6. Package `modeling::expressions`

### B.6.1. Package Overview

The base package for all expressions which can be used for modeling activities and patterns.

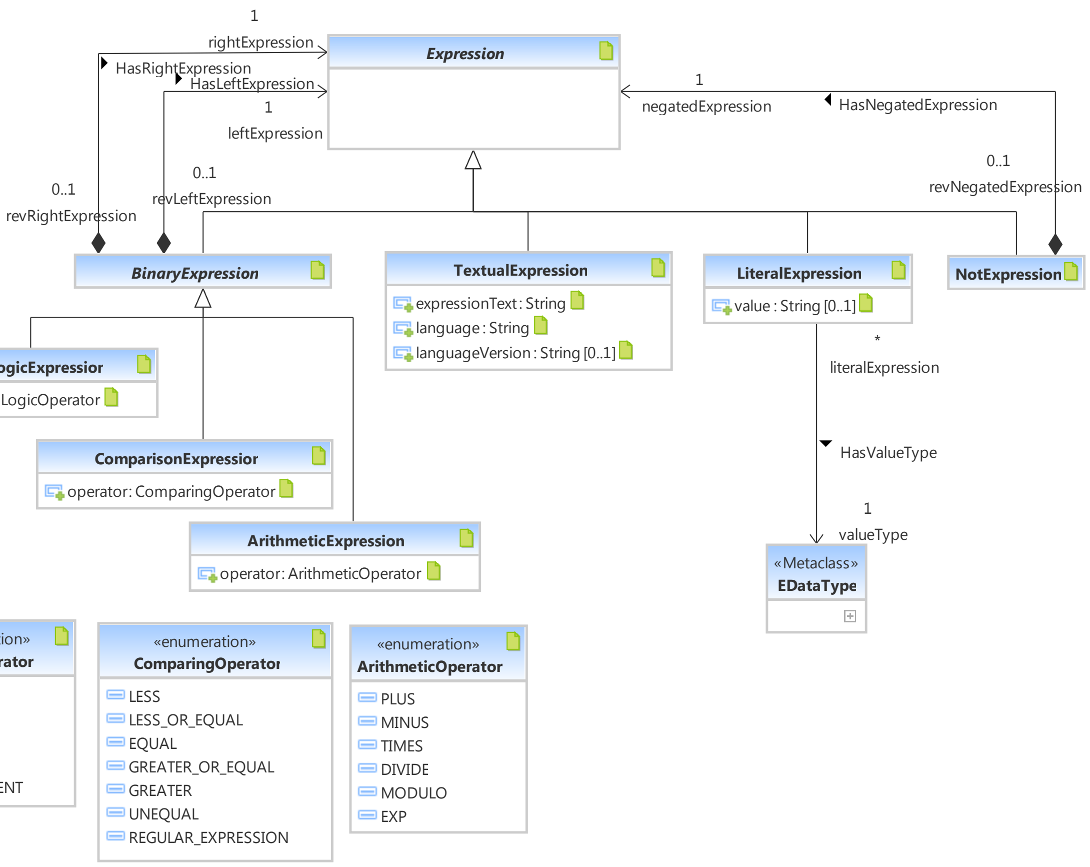


Figure B.6.: Metamodel of the expressions Package

### B.6.2. Detailed Contents Documentation

#### B.6.2.1. Class `ArithmeticExpression`

**Overview** Represents arithmetic expressions like a + 5 or a * 7.

**Class Properties** Class `ArithmeticExpression` has the following properties:

**operator : ArithmeticOperator** see Section B.6.2.2 on Page 84

Specifies the expression’s arithmetic operator, e.g. +, -, *, /, or MODULO.

#### Parent Classes

- BinaryExpression see Section B.6.2.3 on Page 84

### B.6.2.2. Enumeration `ArithmeticOperator`

**Overview** Defines the operators for arithmetic expressions.

**Enum Properties** Enumeration `ArithmeticOperator` has the following literals:

**PLUS = 0**

**MINUS = 1**

**TIMES = 2**

**DIVIDE = 3**

**MODULO = 4**

**EXP = 5**

For formulas like aˆb.

### B.6.2.3. Class `BinaryExpression`

**Overview** Represents any binary expression like v < 5 or x + 7.

#### Parent Classes

- Expression see Section B.6.2.7 on Page 85

### B.6.2.4. Class `BinaryLogicExpression`

**Overview** Represents binary, logic expressions like a AND b and a OR b.

**Class Properties** Class `BinaryLogicExpression` has the following properties:

**operator : LogicOperator** see Section B.6.2.9 on Page 86

Specifies the expression's logic operator, e.g. AND, OR, or XOR.

#### Parent Classes

- BinaryExpression see Section B.6.2.3 on Page 84

### B.6.2.5. Enumeration `ComparingOperator`

**Overview** Defines the operators for comparing expressions.

**Enum Properties** Enumeration `ComparingOperator` has the following literals:

**LESS = 0**

**LESS_OR_EQUAL = 1**

**EQUAL = 2**

**GREATER_OR_EQUAL = 3**

**GREATER = 4**

**UNEQUAL = 5**

**REGULAR_EXPRESSION = 6**

For comparison of a String with a regular expression.

### B.6.2.6. Class `ComparisonExpression`

**Overview** Represents comparing expressions like a < 5 or a >= 7.

**Class Properties** Class `ComparisonExpression` has the following properties:

**operator : ComparingOperator** see Section B.6.2.5 on Page 85

Specifies the expression's comparing operator, e.g. <, >=, !=.

**Parent Classes**

- BinaryExpression see Section B.6.2.3 on Page 84

### B.6.2.7. Class `Expression`

**Overview** Represents any expression in an embedded textual language, e.g. OCL or Java. An expression's type is dynamically derived by an external mechanism (see TypedElement).

**Parent Classes**

- TypedElement see Section B.1.2.5 on Page 67,
- CommentableElement see Section B.1.2.1 on Page 65

#### B.6.2.8. Class `LiteralExpression`

**Overview** Represents any literal, i.e. a value whose type is an EDataType. Literals are, for example, 5, 3.14, 'c', "text", true.

**Class Properties** Class `LiteralExpression` has the following properties:

**value : EString [0..1]**

String representation of the value, e.g. "5", "3.14", "c", "text", or "true".

**Class References** Class `LiteralExpression` has the following references:

**valueType : EDataType**

The literal's type, e.g. EInt, EString, etc.

**Parent Classes**

- Expression see Section B.6.2.7 on Page 85

#### B.6.2.9. Enumeration `LogicOperator`

**Overview** Defines the operators for binary logic expressions. The unary logic expression representing negated expressions is reflected by the NotExpression.

**Enum Properties** Enumeration `LogicOperator` has the following literals:

**AND = 0**

**OR = 1**

**XOR = 2**

**IMPLY = 3**

**EQUIVALENT = 4**

#### B.6.2.10. Class `NotExpression`

**Overview** Represents a negated expression, e.g. NOT (a < 5).

**Parent Classes**

- Expression see Section B.6.2.7 on Page 85

#### B.6.2.11. Class `TextualExpression`

**Overview** Represents any expression in a textual language embedded into Story Diagrams, e.g. OCL or Java .

**Class Properties** Class `TextualExpression` has the following properties:

**expressionText : EString**

Holds the expression, e.g. in OCL or Java.

**language : EString**

String representation of the used language which has to be unique. Examples are OCL and Java.

**languageVersion : EString [0..1]**

String representation of the used language's version. The format is <Major>.<Minor>[.<Revision>[.<Build>]] Examples: 1.4 or 3.0.1 or 1.0.2.20101208.

**Parent Classes**

- Expression see Section B.6.2.7 on Page 85

## B.7. Package `modeling::patterns`

### B.7.1. Package Overview

This package contains all classes for modeling story patterns that may be embedded into StoryActivityNodes of an Activity.

### B.7.2. Detailed Contents Documentation

#### B.7.2.1. Class `AbstractLinkVariable`

**Overview** Abstract super class for all kinds of link variables that represent links between two objects in a story pattern.

**Class Properties** Class `AbstractLinkVariable` has the following properties:

**bindingOperator : BindingOperator** see Section B.7.2.4 on Page 91

The binding operator defines whether this link will be matched, created or destroyed by the story pattern. The default value ist "check_only", i.e., the link will be matched.

**bindingSemantics : BindingSemantics** see Section B.7.2.5 on Page 91

The binding semantics defines whether the link must be matched for a successful application of the containing story pattern, whether it must not be matched or whether it is optional, i.e., it will be bound if it can be bound but that does not affect the success of matching the story pattern. The default value is "mandatory" (i.e., it must be matched).

**bindingState : BindingState** see Section B.7.2.6 on Page 92

The binding state defines whether the link is already bound or whether a match has to be obtained for it.

**Class References** Class `AbstractLinkVariable` has the following references:

**firstLinkConstraint : LinkConstraint [0..$*$]** see Section B.7.2.10 on Page 93

The constraint that refers to this link as the "first link".

**pattern : StoryPattern** see Section B.7.2.18 on Page 97

The story pattern in which the link variable is contained.

**secondLinkConstraint : LinkConstraint [0..$*$]** see Section B.7.2.10 on Page 93

The constraint that refers to this link as the "second link".

**source : ObjectVariable** see Section B.7.2.15 on Page 95

The source of the link. This always has to be an object variable.

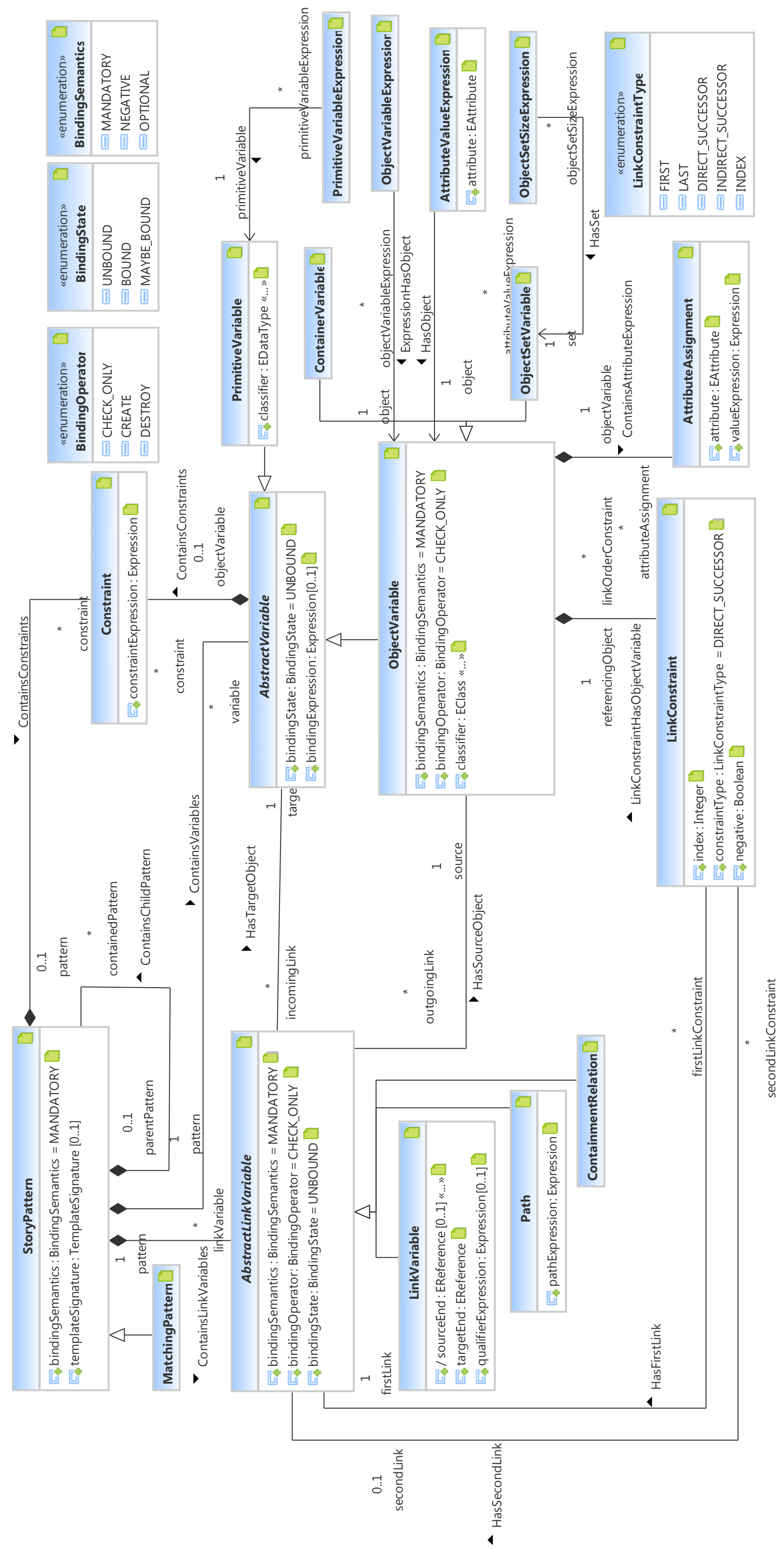


Figure B.7.: Metamodel of the patterns Package

**target : AbstractVariable** see Section B.7.2.2 on Page 90

The target of the link.

**Parent Classes**

- NamedElement see Section B.1.2.4 on Page 67

### B.7.2.2. Class `AbstractVariable`

**Overview** Abstract super class for object and primitive variables.

**Class Properties** Class `AbstractVariable` has the following properties:

**bindingState : BindingState** see Section B.7.2.6 on Page 92

The binding state defines whether the variable is already bound or whether a match has to be obtained for it. The default value is "unbound".

**Class References** Class `AbstractVariable` has the following references:

**bindingExpression : Expression [0..1]** see Section B.6.2.7 on Page 85

A binding expression can be used to bind a variable in a different way than just by pattern matching. This way, for example, the return value of a call can be bound to a variable.

**constraint : Constraint [0..$*$]** see Section B.7.2.7 on Page 92

All constraints which are defined for this variable. For a successful matching, all constraints for this variable must evaluate to true.

**incomingLink : AbstractLinkVariable [0..$*$]** see Section B.7.2.1 on Page 88

Represents the link variables whose target is this variable.

**pattern : StoryPattern** see Section B.7.2.18 on Page 97

Represents the story pattern this variable is contained in.

**Parent Classes**

- Variable see Section B.1.2.6 on Page 67,
- NamedElement see Section B.1.2.4 on Page 67

### B.7.2.3. Class `AttributeAssignment`

**Overview** An AttributeAssignment is used to set the value of a certain attribute of an object. It references the attribute that is to be set and the value. The value can be an expression to allow for calculations or calls that determine the final value. AttributeAssignments are carried out during the final phase of pattern application, i.e. after the matching and destruction are completed.

### B.7.2.4. Enumeration `BindingOperator`

**Overview** The BindingOperator enum defines all possible operations for object and link variables. An object or link variable may be checked for existence be the story pattern (black object/link variable), it may be created (green object/link variable), or it may be destroyed (red object/link variable).

**Enum Properties** Enumeration `BindingOperator` has the following literals:

**CHECK_ONLY = 0**

CHECK_ONLY is the default value of this enum. It requires an object or link variable just to be matched by the story pattern.

**CREATE = 1**

An object or link variable marked as CREATE will be created by the story pattern.

**DESTROY = 2**

An object or link variable marked as DESTROY will be destroyed be the story pattern.

### B.7.2.5. Enumeration `BindingSemantics`

**Overview** The binding semantics defines which kind of match will be obtained for the object or link variable.

**Enum Properties** Enumeration `BindingSemantics` has the following literals:

**MANDATORY = 0**

For a mandatory object or link variable, a match has to be found for a pattern to be successfully applied.

**NEGATIVE = 1**

If an object or link variable is marked as NEGATIVE, no match may be found for that object or link variable. If a match can be found, the execution of the story pattern fails.

**OPTIONAL = 2**

For an OPTIONAL object or link variable, the matching tries to find a match. If no match can be found, this does not affect the success of the pattern application. If a match can be found, the respective object or link is bound to the variable.

#### B.7.2.6. Enumeration `BindingState`

**Overview** The BindingState defines whether an object or link variable is already bound to a concrete value or not.

**Enum Properties** Enumeration `BindingState` has the following literals:

**UNBOUND = 0**

UNBOUND is the default value for this enum. If an object or link variable in a story pattern is unbound, a new match has to be obtained for that variable.

**BOUND = 1**

A bound variable has already been bound to a concrete value. The concrete value has to be passed either as a parameter or it has to be bound in a previous activity. If, during the execution of a story pattern, a bound variable has no value, the execution of the story pattern fails.

**MAYBE_BOUND = 2**

A variable marked with maybe_bound indicates that it is unknown (or unimportant) at design time whether the variable is bound or not. If, during the execution of the pattern, the variable is not bound, an object is matched and bound to the variable. If it is already bound, it is not altered. If the variable is still unbound after this process, the matching fails (except for OPTIONAL variables).

#### B.7.2.7. Class `Constraint`

**Overview** A constraint represents a condition which must be fulfilled for a successful pattern matching. It can either be contained in the story pattern or in a variable. In the former case, the constraint is evaluated after the matching of the object structure is complete. It still has to be true for the pattern application to be sucessful (and therefore for creations and destructions to be carried out). If the constraint is contained in a variable, it constrains the matching of that variable, i.e., it is evaluated during the matching of the containing variable and has to be true for a successful matching. If the variable is an ObjectSetVariable, the constraint has to be true for every object in the set.

### B.7.2.8. Class `ContainerVariable`

**Overview** Represents a single container, e.g. a Set or List. ContainmentRelations can be used to add or remove objects to or from this container. Every Constraint or AttributeAssignment can use the variable as a container (e.g., "set->size() > 5").

#### Parent Classes

- ObjectVariable see Section B.7.2.15 on Page 95

### B.7.2.9. Class `ContainmentRelation`

**Overview** Specifies the containment of an object in a set (represented by a ContainerVariable). Will be displayed by a line having a circle with a plus inside at the end of the container (the source end of the link). A create modifier specifies that the object will be added to the container, delete that it will be removed, and none that it will be checked to be contained.

#### Parent Classes

- AbstractLinkVariable see Section B.7.2.1 on Page 88

### B.7.2.10. Class `LinkConstraint`

**Overview** Link constraints (formerly known as MultiLinks in old meta-model) constrain the ordering of links if the referencingObject is a collection. This way objects can be required to have a certain position in the collection (FIRST, LAST, INDEX) or a certain ordering relative to each other (DIRECT_SUCCESSOR, INDIRECT_SUCCESSOR). While the first kind of LinkConstraint can be imposed upon a single link, the second kind requires two links that are related to each other (e.g., have the same referencingObject).

**Class Properties** Class `LinkConstraint` has the following properties:

**constraintType : LinkConstraintType** see Section B.7.2.11 on Page 94

The constraint type of the LinkConstraint.

**index : EInt**

The index of the linked object in the collection. The semantics of this attribute is only defined if the constraintType of the LinkConstraint is INDEX.

**negative : EBoolean**

If the negative attribute is true, the link constraint may not be fulfilled for the complete pattern application to be successful.

**Class References** Class `LinkConstraint` has the following references:

**firstLink : AbstractLinkVariable** see Section B.7.2.1 on Page 88

The first link that the link constraint refers to. For link constraints that only relate to one link (FIRST, LAST, INDEX) this is the only referenced link.

**referencingObject : ObjectVariable** see Section B.7.2.15 on Page 95

The ObjectVariable to which this LinkContraint is associated. That ObjectVariable has to be a collection.

**secondLink : AbstractLinkVariable [0..1]** see Section B.7.2.1 on Page 88

The second link that the link constraint refers to. Applies only to link constraints that relate two links to each other (DIRECT_SUCCESSOR and INDIRECT_SUCCESSOR). For all other constraints this variable is always null.

#### Parent Classes

- ExtendableElement see Section B.1.2.2 on Page 65

### B.7.2.11. Enumeration `LinkConstraintType`

**Overview** The LinkConstraintType represents the different uses of LinkConstraints. Objects can be required to have a certain position in their containing collection (FIRST, LAST, INDEX) or a certain ordering relative to each other (DIRECT_SUCCESSOR, INDIRECT_SUCCESSOR).

**Enum Properties** Enumeration `LinkConstraintType` has the following literals:

**FIRST = 0**

**LAST = 1**

**DIRECT_SUCCESSOR = 2**

**INDIRECT_SUCCESSOR = 3**

**INDEX = 4**

### B.7.2.12. Class `LinkVariable`

**Overview** A link variable represents one link between two object variables. It is typed over one of the associations between the classes of those objects. Because EMF only directly supports references, the two link ends are typed over these references. In case of a uni-directional association, only the targetEnd is typed. In case of a bi-directional association, the reference that types the source end is automatically determined.

#### Parent Classes

- AbstractLinkVariable see Section B.7.2.1 on Page 88

### B.7.2.13. Class `MatchingPattern`

**Overview** A MatchingPattern is a special kind of story pattern that does not change the underlying graph. Thus, no contained object or link may carry an create or destroy BindingOperator.

#### Parent Classes

- StoryPattern see Section B.7.2.18 on Page 97

### B.7.2.14. Class `ObjectSetVariable`

**Overview** Represents a set of objects of the same type that are represented by a single node. The context for contained Constraints and AttributeAssignments is every single object in the set. E.g., if the constraint is "name = 'abc'", only objects with that name are matched and added to the set. The use of the binding operator "CREATE" is not defined for ObjectSetVariables, i.e., the sets can only be matched and deleted.

#### Parent Classes

- ObjectVariable see Section B.7.2.15 on Page 95

### B.7.2.15. Class `ObjectVariable`

**Overview** An ObjectVariable holds a value of a complex type which is defined by an EClass.

**Class Properties** Class `ObjectVariable` has the following properties:

**bindingOperator : BindingOperator** see Section B.7.2.4 on Page 91

The binding operator defines whether this object will be matched, created or destroyed by the story pattern.

**bindingSemantics : BindingSemantics** see Section B.7.2.5 on Page 91

The binding semantics defines whether the object must be matched for a successful application of the containing story pattern, whether it must not be matched or whether it is optional, i.e., it will be bound if it can be bound but that does not affect the success of matching the story pattern.

**Class References** Class `ObjectVariable` has the following references:

**attributeAssignment : AttributeAssignment [0..∗]** see Section B.7.2.3 on Page 91

The AttributeAssignments that have to be executed for this ObjectVariable.

**classifier : EClass**

The type of this ObjectVariable, given as an EClass.

**linkOrderConstraint : LinkConstraint [0..∗]** see Section B.7.2.10 on Page 93

The LinkConstraints that are imposed on the links of this ObjectVariable. Only makes sense if the ObjectVariable is a collection.

**outgoingLink : AbstractLinkVariable [0..∗]** see Section B.7.2.1 on Page 88

Represents the link variables whose source is this object variable.

#### Parent Classes

- AbstractVariable see Section B.7.2.2 on Page 90

### B.7.2.16. Class `Path`

**Overview** A path is a special link variable that specifies an indirect connection between two objects. That means, the connected objects have other links and objects "between them". Exactly which types of links may be traversed during the matching of a path can be constrained by a path expression.

#### Parent Classes

- AbstractLinkVariable see Section B.7.2.1 on Page 88

### B.7.2.17. Class `PrimitiveVariable`

**Overview** Represents a variable that holds a value of a primitive type, e.g. integer, boolean, String.

#### Parent Classes

- AbstractVariable see Section B.7.2.2 on Page 90

#### B.7.2.18. Class `StoryPattern`

**Overview** A Story Pattern is a graph rewrite rule that may be embedded into a StoryActivityNode of an Activity.

**Class Properties** Class `StoryPattern` has the following properties:

**bindingSemantics : BindingSemantics** see Section B.7.2.5 on Page 91

The binding semantics of a story pattern express if a pattern as a whole should matched normally (MANDATORY), if it should be NEGATIVE or OPTIONAL. This only makes sense if the StoryPattern is a sub pattern f another StoryPattern. Top-level StoryPatterns should always have the binding semantics MANDATORY.

**Class References** Class `StoryPattern` has the following references:

**constraint : Constraint [0..∗]** see Section B.7.2.7 on Page 92

All constraints which are defined for this story pattern. For a successful matching, all constraints for this story pattern must evaluate to true.

**containedPattern : StoryPattern [0..∗]** see Section B.7.2.18 on Page 97

A number of sub patterns that are contained in this StoryPattern. They can be negated or made optional as a whole by setting their binding semantics accordingly.

**linkVariable : AbstractLinkVariable [0..∗]** see Section B.7.2.1 on Page 88

All the LinkVariables that are contained in this StoryPattern.

**parentPattern : StoryPattern [0..1]** see Section B.7.2.18 on Page 97

If the StoryPattern is a sub pattern, this points to the StoryPattern in which the sub pattern is contained.

**templateSignature : TemplateSignature [0..1]** see Section B.9.2.3 on Page 102

**variable : AbstractVariable [0..∗]** see Section B.7.2.2 on Page 90

All the variables that are contained in the StoryPattern.

**Parent Classes**

- CommentableElement see Section B.1.2.1 on Page 65

## B.8. Package `modeling::patterns::expressions`

### B.8.1. Package Overview

This package offers expressions that refer to different kinds of variables like object variables, attributes, etc.

### B.8.2. Detailed Contents Documentation

#### B.8.2.1. Class `AttributeValueExpression`

**Overview** Represents the value of an object's attribute, e.g. obj.attr for an object obj and an attribute attr.

##### Parent Classes

- Expression see Section B.6.2.7 on Page 85

#### B.8.2.2. Class `ObjectSetSizeExpression`

**Overview** Represents the number of elements in the set of objects that is represented by an object set variable. For example, if you have an object set variable mySet, then this expression would represent something like mySet.size(). The expression can be used to constrain the pattern application, e.g., to only a apply the pattern when at least two objects can be matched for the set.

##### Parent Classes

- Expression see Section B.6.2.7 on Page 85

#### B.8.2.3. Class `ObjectVariableExpression`

**Overview** Represents the reference to an object in an expression, i.e. the value of an object variable.

##### Parent Classes

- Expression see Section B.6.2.7 on Page 85

#### B.8.2.4. Class `PrimitiveVariableExpression`

**Overview** Represents the value of a primitive variable, e.g., 5 or "MyName".

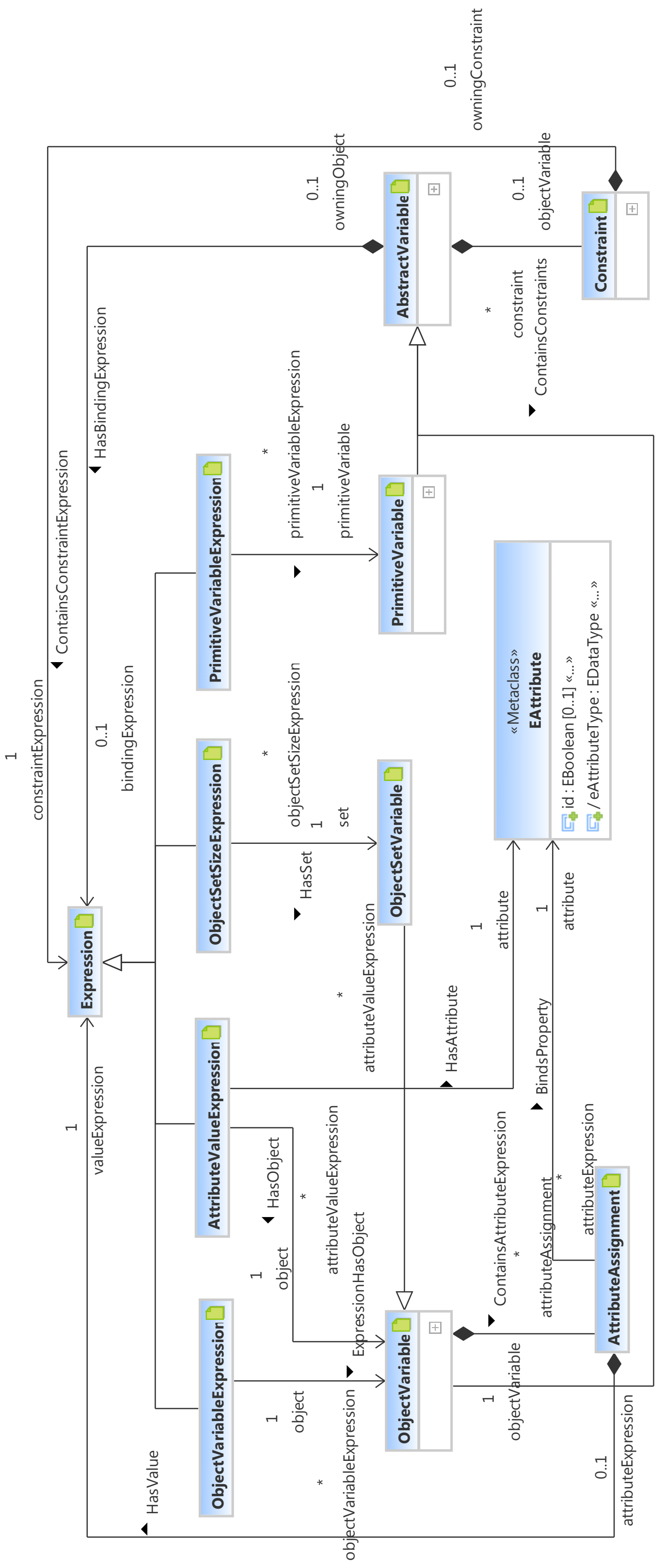


Figure B.8.: Metamodel of the patterns::expressions Package

### Parent Classes

- Expression see Section B.6.2.7 on Page 85

## B.9. Package `modeling::templates`

### B.9.1. Package Overview

This package offers classes that enable to define template for story patterns to re-use existing, structurally similar story patterns in other story patterns. The templates are flexible such that the object variable types can be replaced when a template is used in a certain context.

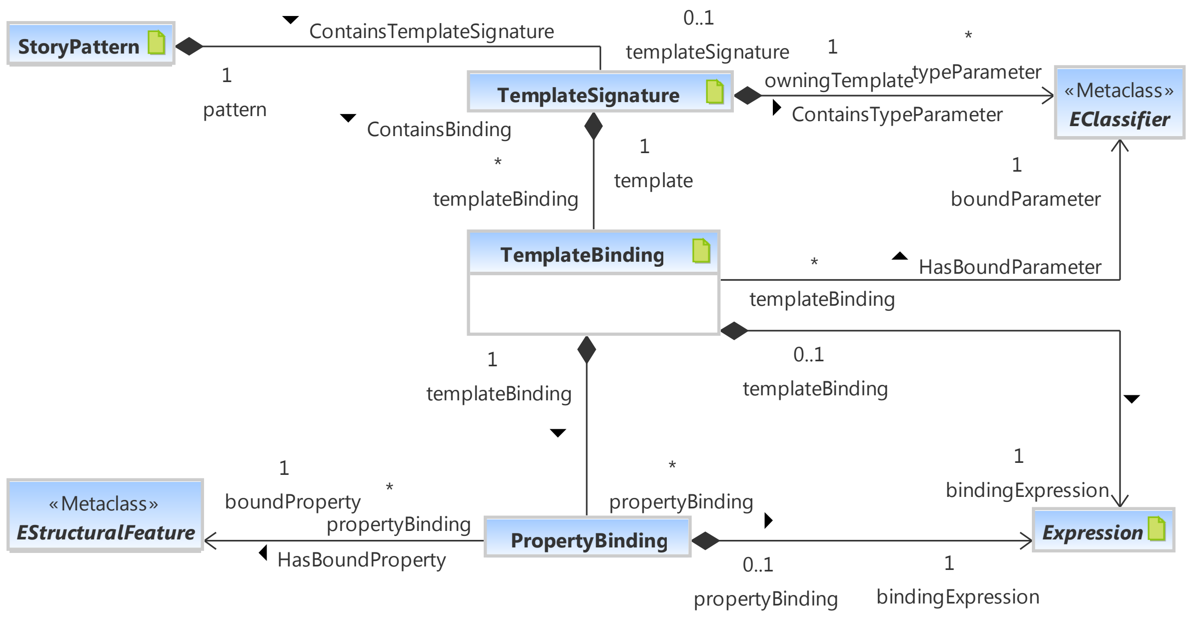


Figure B.9.: Metamodel of the templates Package

### B.9.2. Detailed Contents Documentation

#### B.9.2.1. Class `PropertyBinding`

**Overview**

**Parent Classes**

- ExtendableElement see Section B.1.2.2 on Page 65

#### B.9.2.2. Class `TemplateBinding`

**Overview** This class represents the binding of a story pattern template's type parameter to a concrete type.

#### Parent Classes

- ExtendableElement see Section B.1.2.2 on Page 65

### B.9.2.3. Class `TemplateSignature`

**Overview** This class is used to define type parameters for a template that represents a story pattern to be reused. The type parameter is replaced by a concrete type when the template is applied in a story pattern.