

Hybrid Rule Scheduling in Story Driven Modeling – a tool-independent approach

Bart Meyers

Eindwerk ingediend met het oog op het behalen
van de graad van Master in de Wetenschappen.

Promotor: Prof. Dr. Dirk Janssens
Co-Promotor: Dr. Pieter Van Gorp

UNIVERSITEIT OF ANTWERPEN
FACULTEIT WETENSCHAPPEN
DEPARTEMENT WISKUNDE EN INFORMATICA
ACADEMIEJAAR 2008-2009

Contents

List of Figures	vi
List of Tables	vii
Dankwoord	1
Samenvatting	2
Abstract	4
1 Introduction	5
2 Transformation languages and rule scheduling	7
2.1 Model-driven engineering	7
2.2 Transformation languages in MDE	8
2.3 Graph Grammars	9
2.4 Rule scheduling	10
2.5 Story-driven modeling (SDM)	11
2.6 Explicit rule scheduling in MoTMoT	13
2.7 Implicit rule scheduling in AToM ³	14
2.8 The schism between implicit and explicit rule scheduling	15
2.9 Conclusion of this chapter	17
3 ND-SDM - Hybrid rule scheduling in SDM	18
3.1 Implicit rule scheduling	19
3.2 Layers	35

3.3	Priorities	35
3.4	Caveat	37
3.5	Conclusion of this chapter	39
4	Implementation of ND-SDM	40
4.1	Requirements	40
4.2	Higher order transformations	41
4.3	Prototypes	42
4.4	A higher order transformation and prototypes as implementation of ND-SDM	46
4.5	Conclusion of this chapter	69
5	Case studies in the usage of ND-SDM	71
5.1	The UML to RDB transformation	71
5.2	The Petri Net simulation transformation	78
5.3	The poker game simulation transformation	91
6	Negative application conditions in SDM	102
6.1	Negative application conditions	102
6.2	The negative application conditions implementation	111
7	Conclusion	116
7.1	Related work	116
7.2	Future work	117
7.3	Conclusion	121
A	Auxiliary diagrams	122
A.1	UML metamodel	122
A.2	Story Patterns of the nondeterministic state HOT	125
	Bibliography	144

List of Figures

2.1	The transformation architecture	8
2.2	Transformation language features for rule scheduling (Van Gorp08a)	10
3.1	Three rules scheduled consecutively in MoTMoT	20
3.2	Three rules in another order	20
3.3	Three rules scheduled implicitly in ND-SDM	21
3.4	Three rules scheduled implicitly in AToM ³	21
3.5	The three rules scheduled implicitly in AGG	22
3.6	A household modeled using hybrid (top) and explicit (bottom) rule scheduling	24
3.7	A model of the composition of a plate in a restaurant using hybrid (top) and explicit (bottom) rule scheduling	27
3.8	A series of implicitly scheduled rules with three layers	35
3.9	A series of implicitly scheduled rules with priorities and its meaning	38
3.10	A series of implicitly scheduled rules with priorities in AToM ³	38
4.1	A transformation rule that creates a parent and a child and their association .	44
4.2	A prototype of the transformation rule	45
4.3	The integration of the prototype into the transformation rule	45
4.4	The architecture of the ND-SDM implementation	47
4.5	The application condition of the variant of the household problem	48
4.6	The actual prototype of the variant of the household problem	49
4.7	The application condition of the variant of the restaurant problem	50
4.8	The actual prototype of the variant of the restaurant problem	51
4.9	A household modeled using hybrid rule scheduling	55

4.10	A household after execution of the HOT	56
4.11	An overview of the ND-SDM implementation HOT	57
4.12	Loading the application conditions of the prototypes	59
4.13	Transforming the <i>«nextPriority»</i> transitions	60
4.14	The Story Diagram that checks for incoming priority transitions	60
4.15	The Story Diagram that transforms the next priority	61
4.16	Choosing the right variant according to the application conditions	64
4.17	Reading and integrating the chosen variant	65
4.18	Adding a method and a call to this method	67
4.19	The architecture of the ND-SDM implementation using generic containers	70
5.1	A storehouse inventory system in Class Diagrams	72
5.2	A storehouse inventory system in a UML Relational Data Model	72
5.3	A transformation from Class Diagrams to a Relational Database Scheme	74
5.4	The UML to RDB transformation in ND-SDM	76
5.5	The flattened UML to RDB transformation	77
5.6	A Petri Net in AToM ³ that models access control	79
5.7	A metamodel of Petri Nets in AToM ³	80
5.8	A Petri Net in the UML that models access control	85
5.9	The Petri Net simulation transformation in ND-SDM	87
5.10	A possible marking after the execution of the transformation	88
5.11	The flattened Petri Net simulation transformation	89
5.12	The input model of the poker simulation transformation	93
5.13	The players are dealt two cards each	94
5.14	Frank is winning after the flop	95
5.15	Mary is winning after the turn	96
5.16	Patrick has won after the river	97
5.17	The Story Diagram of the poker simulation transformation	98
5.18	The pattern for matching a full house	100
5.19	The Story Diagram after the execution of the HOT	101
6.1	The UML to RDB transformation in ND-SDM	106
6.2	This pattern matches when no RDB constructs can be found	107
6.3	This pattern transforms classes to tables	108
6.4	The <i>match</i> view of the pattern that transforms datafields	109
6.5	The <i>create</i> view of the pattern that transforms datafields	110
6.6	The class transformation pattern of the MoTMoT equivalent	112
6.7	The Story Diagram of the flattened UML to RDB transformation	112

6.8	The Story Diagram of the NAC	112
6.9	The Story Pattern of the NAC	113
6.10	The Story Diagram of the NAC HOT	115
A.1	UML core backbone (OMG01)	122
A.2	UML core relationships (OMG01)	123
A.3	UML core extension mechanisms (OMG01)	123
A.4	UML data types (OMG01)	123
A.5	UML model management (OMG01)	124
A.6	UML Activity Diagrams (OMG01)	124
A.7	Look up SDM profile package, create prototype application condition package	125
A.8	Match SDM profile metadata	126
A.9	Match SDM profile metadata (cont.)	127
A.10	Match Java profile metadata	128
A.11	List possible prototypes	128
A.12	Increase counter	129
A.13	Add container	129
A.14	Read application condition	130
A.15	Get application condition state	131
A.16	Get application condition stereotypes	131
A.17	Get application condition tag definitions	132
A.18	Match a <i>«nextPriority»</i> transition and create a <i>«success»</i> loop	132
A.19	Check whether <i>source</i> has an incoming <i>«nextPriority»</i> transition	133
A.20	Transform a <i>«nextPriority»</i> transition	133
A.21	Find the next <i>«nextPriority»</i> transition	133
A.22	Set a <i>«failure»</i> stereotype on the outgoing transition	134
A.23	Initialize ID	134
A.24	Find the next Activity Diagram	134
A.25	Check whether the graph is called	135
A.26	Find a nondeterministic state	135
A.27	Increment ID for each nondeterministic state	135
A.28	Initialize a variable denoting if the prototype is found	136
A.29	Find a previously loaded application condition	136
A.30	Initialize a variable denoting if the current prototype is invalid	136
A.31	Iterate over each stereotype of the application condition	136
A.32	Match the stereotype of the application condition	137
A.33	The current prototype is flagged invalid	137
A.34	Re-initialize a variable denoting if the current prototype is invalid	137

A.35 Iterate over each tag definition of the application condition	137
A.36 Match the tag definition of the application condition	137
A.37 The current prototype is found	137
A.38 Set the number of patterns	138
A.39 Initialize <code>it</code> as a counter that will give a unique ID to each package refer- enced by the nondeterministic state	138
A.40 Look up a package referenced by the nondeterministic state	138
A.41 Increase <code>it</code> in order to have a different number for each referenced package	138
A.42 Create a state and its corresponding transitions	139
A.43 Check whether the currently bound package contains an Activity Diagram . .	140
A.44 Create a <i>motmot.transprimitive</i> tag on the state	140
A.45 Create a method and a call	141
A.46 Add a parameter to the method and its call	142
A.47 Close the method call parameter list	143
A.48 Remove a nondeterministic state	143
A.49 Remove the package containing the prototype application conditions	143

List of Tables

2.1	Language constructs in Story Diagrams	12
2.2	Language constructs in Story Patterns	13
2.3	Language constructs in implicit and explicit rule scheduling	17
3.1	A possible execution order of the household tasks	26
3.2	The properties of the household problem and restaurant problem	29
3.3	Possible variants of implicit rule scheduling	32
3.4	Possible variants of implicit rule scheduling (continued)	33
3.5	Possible variants of implicit rule scheduling (continued)	34
5.1	Implementation of the Petri Net simulation transformation in AToM ³	81
5.2	Implementation of the Petri Net simulation transformation in AToM ³ (cont.)	82
5.3	Implementation of the Petri Net simulation transformation in AToM ³ (cont.)	83
5.4	The poker ranking with examples, in descending order (Wikipedia09)	91
6.1	An example of a NAC association	104
6.2	An example of a NAC subgraph	105
7.1	Possible language extensions for SDM	120

Dankwoord

Ik dank professor Dirk Janssens voor het steunen van mijn eigen aanbreng in deze masterproef, voor het begeleiden en evalueren, en voor me de kans te geven om aan de workshop in Dresden deel te nemen. Ik dank professor Albert Zündorf voor de discussies over Story Driven Modeling. Ik dank professor Andy Schürr voor zijn toelichtingen bij Graph Grammars en nondeterminisme. Ik dank professor Hans Vangheluwe voor zijn aanstekelijke gedrevenheid voor dit onderzoeksdomein, wat me hielp tot het kiezen van mijn onderwerp voor deze masterproef. Ik dank Pieter Van Gorp voor zijn uitstekende begeleiding en stimulerend enthousiasme. Ik hoop dat we in de toekomst nog veel kunnen samenwerken. Ik dank Olaf Muliawan voor zijn technische hulp bij MoTMoT. Ik dank mijn familie en vrienden voor hun morele steun, en mijn moeder in het bijzonder voor de taalcorrectie,

Bart

Samenvatting

In *model driven engineering* (MDE) staan modellen centraal. Deze modellen beschrijven een softwaresysteem en zijn in alle ontwikkelingsfases prominent aanwezig. Verschillende soorten modellen worden gebruikt om verschillende aspecten van het systeem te modelleren. Omdat deze modellen allemaal hetzelfde systeem beschrijven moeten ze consistent gehouden worden. Dit gebeurt door modeltransformaties, die een model automatisch omzetten naar een ander model.

In een MDE framework moet het mogelijk zijn om nieuwe modelleertalen en modeltransformaties toe te voegen. Modeltransformaties worden in een transformatietaal gedefinieerd. Een belangrijk deel van deze transformatietalen zijn gebaseerd op *Graph Grammars*. Een *Graph Grammar* bestaat uit een verzameling transformatieregels. Het uitvoeren van een dergelijke regel transformeert een deel van het model. Deze regels worden op een bepaalde volgorde uitgevoerd, en deze volgorde kan gemanipuleerd worden aan de hand van *rule scheduling*-voorzieningen van de transformatietaal.

Transformatietalen of *tools* kunnen *rule scheduling* toelaten op verschillende manieren. Sommige talen voorzien expliciete *rule scheduling* in de vorm van bijvoorbeeld *conditionals* en *loops*. Een voorbeeld hiervan is de familie van *tools* die *Story Diagrams*, gebaseerd op *Graph Grammars*, ondersteunen. Andere talen ondersteunen impliciete *rule scheduling*, waarbij de volgorde van regels declaratief, dus ongedefinieerd is. De originele specificatie van *Graph Grammars* is hiervan een voorbeeld.

De meeste *tools* ondersteunen echter slechts één van beide paradigma's. Hierdoor worden in bepaalde situaties gekunstelde constructies aangewend, terwijl het probleem elegant zou uitgedrukt kunnen worden in het paradigma dat niet ondersteund wordt. Op deze manier zijn transformatiemodellen veel minder leesbaar en duidelijk dan ze zouden kunnen zijn. Daarbovenop moet men in het begin van de software ontwikkelingscyclus een *tool* kiezen, wat betekent dat men in deze fase al voor een designparadigma moet kiezen, terwijl dit in een veel latere fase zou moeten gebeuren.

Deze thesis biedt een oplossing voor dit probleem door de *rule scheduling*-voorzieningen *Story Diagrams* uit te breiden met impliciete *rule scheduling*. Concreet houdt dit in dat taalconstructies als nondeterminisme, prioriteiten en laagstructuren worden geïmplementeerd

in *Story Diagrams*. Zo ontstaat er een transformatietaal, genaamd ND-SDM, die hybride is ten opzichte van *rule scheduling*.

Als men denkt aan nondeterminisme in *rule scheduling*, dan blijkt men niet één eenduidige betekenis te vinden. Wordt er slechts één van de regels geëvalueerd, of allemaal één keer? Verschillende varianten kunnen in een bepaalde context nuttig zijn. Het is echter onmogelijk om alle varianten te implementeren in *Story Diagrams*.

De implementatie in *Story Diagrams* voorziet daarom een *plug-in*-systeem voor prototypes, waarbij nieuwe varianten makkelijk als prototype kunnen toegevoegd of verwijderd worden. Daarnaast is de implementatie *tool*-onafhankelijk. Dit betekent dat de implementatie geschikt is voor gelijk welke *tool* die *Story Diagrams* ondersteunt. Deze *Tool*-onafhankelijkheid wordt bereikt door het gebruik van hogere orde transformaties, of HOTS, die transformatiemodellen zelf transformeren. De HOT neemt ND-SDM modellen als input en transformeert ze naar *Story Diagrams*, waarbij het gedrag wordt behouden. De HOT kan door de *tool* zelf uitgevoerd worden.

Het gebruik van ND-SDM wordt geïllustreerd aan de hand van drie transformaties. De eerste gevalstudie gaat over een transformatie van klassediagrammen naar relationele databasemodellen. De tweede gevalstudie is een transformatie die de werking van een Petri Net simuleert. De laatste gevalstudie simuleert een pokerspel.

Ook andere uitbreidingen op transformatietalen kunnen geïmplementeerd worden met hogere orde transformaties. Om dit aan te tonen worden negatieve applicatiecondities (NACs) geïmplementeerd als HOTS. NACs worden door veel *tools* ondersteund, maar werd nog niet op *tool*-onafhankelijke manier geïmplementeerd.

Abstract

Transformation rules can be controlled explicitly using language constructs such as a loop or a conditional. This approach is realized in Fujaba, VMTS, MOLA and Progres. Alternatively, transformation rules can be controlled implicitly using a fixed strategy. This approach is realized in AGG and AToM³. When modeling transformation systems using one approach exclusively, particular aspects could have been expressed more intuitively using the other approach. Unfortunately, most transformation languages do not enable one to model the control of some rules explicitly while leaving the control of other rules unspecified. Therefore, this thesis proposes a novel integration of implicit scheduling constructs in Story Diagrams, a language based on explicit scheduling.

It has recently been shown how UML profiles and higher order transformations enable one to extend transformation languages in a tool independent way. This technique will be used to produce a tool independent implementation of implicit scheduling in Story Diagrams. Moreover, this thesis validates the generality of this approach by means of another case study. This second case study relates to negative application conditions, a language construct that was already supported by several transformation environments but no transformation tool realized the construct in a tool independent way yet.

CHAPTER 1

Introduction

MoTMoT is a tool that uses the UML (OMG01) as language, which makes platform independence and collaboration with other tools possible (Schippers04). Therefore, when extending the language of MoTMoT, it is very desirable that such an extension is only dependent of the standard language itself, both in syntax and implementation. This means that a model in the new language extension can always be rewritten in the original language. Of course, these extensions would not add to the expressiveness of the language. But this is not worrying if the language is already Turing complete. Although not making the language more expressive, these extensions can greatly enhance the ease of use of the modeler, as well as the readability for fellow developers and third-party non-experts. This thesis validates a technique that allows numerous extensions for such a standardized language.

Transformation languages are the main scope of this thesis. In this thesis, the definition and implementation of two language extensions for SDM Story Diagrams, the transformation language of MoTMoT, are studied. Thereby it will be important that the interchangeability of MoTMoT is preserved. The largest part of this thesis will be the implementation of a language extension to allow hybrid rule scheduling. The general idea of this implementation will be applicable for numerous other language extensions. To back this up, an implementation of another language extension (negative application conditions) will be presented.

In short, the extension of transformation languages and the implementation of hybrid rule scheduling in MoTMoT will be the main contribution of this thesis. To situate hybrid rule scheduling in transformation languages, it has to be considered that certain choices are made when defining transformation languages. The language can be imperative (i.e., operational) or declarative. Explicit rule scheduling mechanisms (e.g., conditionals) tend to be called imperative since they enable one to model the execution of transformation rules in terms of the state of the transformation system. Languages with implicit rule scheduling tend to be called declarative due to the absence of an explicit state concept. There is no better choice in this matter. For certain problems implicit rule scheduling feels more intuitive, whereas in other cases explicit rule scheduling turns out to be convenient. However, tools tend to support only one or the other, and not both implicit and explicit - thus hybrid - rule scheduling.

In Chapter 2 an introduction to the context and the problem of this thesis is thoroughly explained. A new language construct that allows hybrid rule scheduling is introduced in Chapter 3. Chapter 4 explains higher order transformations (HOTs) and the implementation of the new language construct. In Chapter 5 three exemplary transformations are given as cases studies that show the usefulness of the extended transformation language. In Chapter 6, another example of a new language construct that can be implemented using a HOT is introduced, in the form of negative application conditions (NACs). Related work, future work and a final conclusion are presented in Chapter 7.

Transformation languages and rule scheduling

This chapter presents the context of this thesis, as well as the problem statement where this thesis originated from.

2.1 Model-driven engineering

Model-driven engineering (MDE) is a young and upcoming paradigm for software engineering. Some of the main characteristics of MDE are (Kleppe03; Bézivin04):

- the definition and extensive usage of different formal models;
- abstraction through models ("everything is a model");
- the usage of computer aided software engineering (CASE);
- the attempt to overcome the common problems of traditional software development, such as low productivity, portability, interoperability, maintenance and documentation;
- the definition and usage of model transformations.

In particular, OMG's Model Driven Architecture (OMG00) is a well-known framework for model-driven engineering.

Model transformations are of particular importance in MDE (Sendall03). They form the glue that binds the different models together. In this thesis the main context are transformation languages, that is, the languages that can be used to define model transformations.

2.2 Transformation languages in MDE

Transformations are critical in model-driven development. Since in MDE, homemade modeling languages may be defined and integrated at any time in the development process of a software system, transformations have to be tailored accordingly to integrate these new languages before they can actually be used. Therefore, a highly expressive transformation language is very useful, because it facilitates defining the needed transformations. In general, transformation languages should be formal, which means that they have precise syntax and semantics.

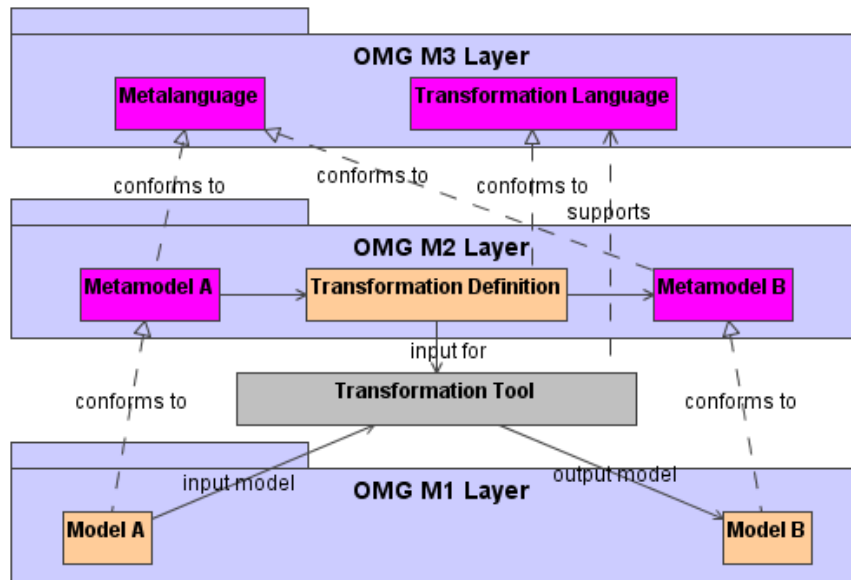


Figure 2.1: The transformation architecture

Figure 2.1 is a draft of how a transformation is built and executed. Purple squares represent languages, and orange squares are models. All languages are also models. The architecture of Figure 2.1 follows the different modeling layers in OMG's MDA (OMG00). Each higher level is able to define the concepts of its lower layer, and each lower layer is an instance of its higher layer. There are no instances of models in the M0 layer. Concepts

in the M3 level can be described by a model in the same layer. This way, there is no need for an infinite number of layers (Kleppe03). The layers are:

- layer M0: the concrete instances, like run-time information of a running application such as data, logs, stackframes;
- layer M1: the models (each model is specified in a modeling language);
- layer M2: the models of the modeling languages, or the metamodels (specified in a metamodeling language);
- layer M3: the models of the metamodeling languages, or the metametamodels.

In Figure 2.1, `Model A` and `Model B` conform to their Metamodels `Metamodel A` and `Metamodel B`. A transformation definition is modeled to transform models that conform to `Metamodel A` to models that conform to `Metamodel B` by a transformation tool. This can be done formally because the metamodels are written in a metalanguage. It is very common that these metamodels are written in the same metalanguage, because they are usually written using the same tool. It is also common that this tool is the transformation tool as well. The transformation definition itself is written in a transformation language, which can be understood by the transformation tool. Although Figure 2.1 follows MDA's layers to clarify the differences in the used models, the idea of its transformation architecture can be applied to MDE in general.

2.3 Graph Grammars

A popular technique for transforming models are Graph Grammars (also known as Graph Rewriting). A Graph Grammar is defined by a set of rules that describe changes to a graph (Rozenberg97; Van Gorp08a). Since models can be interpreted as graphs, many model transformation languages are based on Graph Grammars. With their set of transformation rules (also known as rewrite rules), graphs are transformed to new graphs. Basically, a rule consists of two graphs that share a common subgraph (sometimes called the invariant graph). An application of a rule goes as follows. If the first graph (also called the left graph or left-hand side, because historically visualized on the left-hand side of a rule) of the rule matches a subgraph of the host graph (the graph representing the model), this subgraph is structurally transformed according to the second graph (also called the right graph or right-hand side).

The left-hand side (LHS) is also called the application condition. The act of trying to match a subgraph is called pattern matching or evaluating a rule. When a subgraph is found, it is said that the rule matches, if not, it is said that the rule mismatches, or simply does not match. Finding a matching subgraph causes the graph to be transformed according to the right-hand side (RHS). This is called the application of a rule, the execution of the side-effects, or simply the execution of a rule. The evaluation was then successful.

A note has to be made on the terminology used throughout this thesis. To be entirely correct, it is said that a subgraph of the hostgraph matches the pattern (i.e. the left graph). So one has to say "the pattern is matched". Sometimes however, it is simply said that "the pattern matches", as this reads more smoothly. Saying it this way is not ambiguous, as the pattern and the subgraph actually match one another.

2.4 Rule scheduling

As explained in the previous chapter, rules can be scheduled in many different ways in transformation languages, as shown in Figure 2.2 (Van Gorp08a). Some transformation languages select rules explicitly, which allows controlled scheduling of rules with features like loops and conditionals. Other languages select rules implicitly or nondeterministically. In this case, no explicit scheduling order is put on the set of available rules. The use of features like priorities and layers can however constrain which rules are scheduled at a certain point in the transformation execution. The matching of rules can be subject to negative application conditions (NACs) or regular expressions. The application of a rule can be done for each matching subgraph or just one. In the transformation languages used in this thesis, the latter applies. Alternatively, a matching subgraph can be manually chosen by interaction with the modeler.

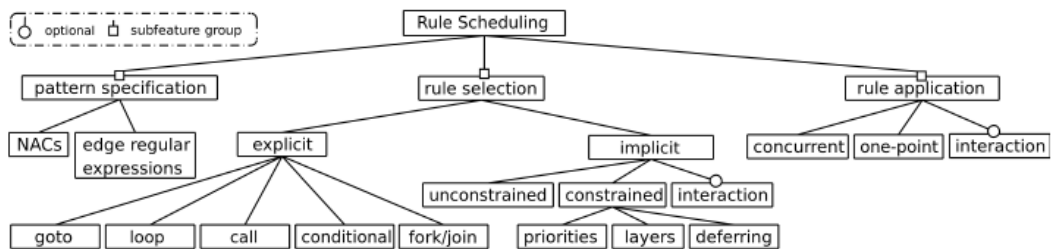


Figure 2.2: Transformation language features for rule scheduling (Van Gorp08a)

In transformation tools or languages that are based on Graph Grammars, it is generally considered out of the scope of Graph Grammars to define how multiple rules are scheduled in a model transformation.¹ This means that tools or languages based on Graph Grammars can support any of the features described in Figure 2.2.

2.5 Story-driven modeling (SDM)

The context of this thesis is Story Driven Modeling (Zündorf02). SDM is designed by Zündorf as a software engineering methodology highly based on the UML. It aims to bridge the gap between analysis techniques and the formal specifications of a software system. This is done by supporting analysis, design and implementation with graph-like structures. These structures form the formal model of the system. Also, the input and output data of the system are treated as graphs.

2.5.1 Story Diagrams

The core of a system specification is the main class or transformation class, with one or more transformation methods. The behavior of a system is defined by so-called Story Diagrams. For every Story Diagram, there is a method in the main class. Story Diagrams are activated by a call to the method it is defined for. Therefore, they can be considered method bodies.

Since the input to the system can be viewed as a graph, in Story Diagrams, these graphs are transformed using a variant of Graph Grammars. Story Diagrams are thus a set of scheduled rules, called Story Patterns. Roughly, Story Diagrams are represented by UML Activity Diagrams, which define the control flow of the Story Patterns. Given the nature of Activity Diagrams, SDM thus employs explicit rule scheduling.

More in detail, each state in a Story Diagram contains a Story Pattern. The transitions between states define in which order the patterns are evaluated. Apart from the Activity Diagram syntax, some elements of Story Diagrams are provided by introducing some additional language constructs. Most of them (i.e. the ones that will be used throughout this thesis) are given in Table 2.1.

¹However, in the original definition of Graph Grammars, rules are all scheduled nondeterministically, and a so-called start graph is given, which is the first graph that is evaluated. Rules are thus scheduled implicitly (Rozenberg97; Van Gorp08a).

Concept	Annotated element	Description
<i>success</i>	Transition	is followed after successful evaluation of a rule
<i>failure</i>	Transition	is followed after unsuccessful evaluation of a rule
<i>loop</i>	ActionState	iteratively loops over a rule
<i>each time</i>	Transition	followed each iteration
<i>link</i> (String)	ActionState	call to transformation method
<i>constraint</i> (String)	ActionState	application condition

Table 2.1: Language constructs in Story Diagrams

When the rule of a state is evaluated, an outgoing transition annotated with *success* is followed if the rule matched. If the rule mismatched, an outgoing *failure* transition is followed. If there is no annotated transition, the outgoing blank transition is followed. A rule is executed repeatedly if it is annotated with the *loop* construct. More specifically, it is executed for each subgraph the rule matches. Each time, an outgoing *each time* transition is followed, allowing the iterative execution of a nested flow. Another Story Diagram can be called with a method call in a *link* state. A boolean expression can be used as *constraint* on a state as an additional application condition. If the Story Pattern of the state matches but the constraint does not, the rule does not match.

2.5.2 Story Patterns

Story Patterns define the rewriting rules scheduled by the Story Diagrams. They are thus closely related to Graph Grammars. But, instead of having to define two graphs for each rule which is the case in Graph Grammars, only one graph is needed. This graph consists of nodes and links, with some of them annotated with *create* or *destroy*. They will be created and destroyed respectively when the side-effects are executed. So in terms of Graph Grammars, the left diagram would be the subgraph of all elements that are not annotated at all or annotated with *destroy*, and the right diagram would be the subgraph of all elements that are not annotated at all or annotated with *create*.

Since SDM is based on the UML, Story Patterns are represented by Class Diagrams. They are visualized inside their corresponding states in the Story Diagram. Classes are nodes and associations are links. Labels (i.e. names and multiplicities) are added to the links, adding to the expressiveness of the pattern definition. Like Story Diagrams, there are some additional language constructs, presented in Table 2.2.

Create and *destroy* have already been explained as part of the side-effect of the rule. A *bound* node is a node that was part of a previously matched rule, so it can be reused as a

Concept	Annotated elements	Description
<i>create</i>	UML <i>Class</i> , UML <i>Association</i>	creates node as side-effect
<i>destroy</i>	UML <i>Class</i> , UML <i>Association</i>	destroys node as side-effect
<i>bound</i>	UML <i>Class</i>	node matched by previous rule
<i>closure</i>	UML <i>Association</i>	transitive closure for link label
<i>constraint</i> (String)	UML <i>Class</i>	application condition

Table 2.2: Language constructs in Story Patterns

fixed node. A transitive *closure* can be put on an association, meaning that this pattern can match for nodes with one or more similar (meaning that the labels are equal to the label on the given association) links between them. A *constraint* is a boolean expression that can be put on a node as an additional application condition.

2.6 Explicit rule scheduling in MoTMoT

The tool that represents the school of explicit rule scheduling will be MoTMoT in this thesis. MoTMoT (Van Gorp07) is based on Story Driven Modeling. In contrast to SDM in Fujaba (Fujaba07), MoTMoT uses the standard UML 1.4 profile (OMG01). As a consequence, the syntax of the models is UML compliant. Instead of employing a new UML-like syntax for Story Diagrams, MoTMoT strictly uses a combination of UML Activity Diagrams and UML Class Diagrams as an equivalent of Story Diagrams (Van Gorp08a). The ubiquitous usage of wide-spread UML is thus maintained. The main difference is that MoTMoT takes this UML basis a step further. In MoTMoT, the Activity Diagram part of Story Diagrams is fully UML compliant, and therefore the Story Patterns can not be visualized inside the states. A state may contain a tagged value named *motmot.transprimitive* with type UML *Package*, that refers to the respective Story Pattern. A Story Pattern is implemented as a UML Class Diagram residing in this package. MoTMoT uses explicit rule scheduling by using Activity Diagrams. The language constructs of SDM, given in Tables 2.1 and 2.2, are implemented as UML *Stereotypes* and *TagDefinitions*.

The usage of a UML profile is an advantage of MoTMoT (Schippers04). A basic profile for transformation modeling was designed in (Van Gorp08b) as a means to allow portability and reuse between tools². On the other hand, a tool specialized in SDM, such as Fujaba, makes modeling and reading better as it visualizes Story Diagrams by putting the Story

²The language constructs in Tables 2.1 and 2.2 are actually part of this profile.

Patterns inside the states (Zündorf02).

MoTMoT is only a transformation tool, not a modeling tool. MoTMoT can read models and transformations, which are also models, and they are stored as XML Metadata Interchange (XMI) (OMG05). Therefore, in the context of MoTMoT, the definition of a transformation can be called a transformation model. Throughout this thesis, the award-winning MagicDraw tool (No Magic09) is used as modeling tool.

Other tools supporting explicit rule scheduling are the previously mentioned SDM tool Fujaba (Fujaba07), VMTS (Levendovszky07), MOLA (Kalnins08), MOFLON (Amelunxen09), and Progres (Schürr09a).

2.7 Implicit rule scheduling in AToM³

In this thesis, AToM³ (De Lara00) represents the tools that use implicit rule scheduling. In AToM³, rules are also heavily based on Graph Grammars. Additionally to the left-hand side (LHS), a Condition can be added as some Python code. Additional side-effects can also be added as Python code in the Action.

All defined rules are scheduled declaratively using priorities. Only if none of the rules of a higher priority match, the rule of a certain priority can be evaluated. Note that the implicit scheduling paradigm of AToM³ refers to the order of evaluation between rules. It does not refer to the order in which subgraphs are matched, if more than one subgraph of the model matches the LHS graph of the evaluating rule. This is random (and thus declarative) in AToM³. When running a transformation, the application of a rule is performed one at a time (as is the case in all transformation languages used throughout this thesis), but it is of course possible in AToM³ (and very common) that the same rule is applied consecutively. AToM³ also offers a feature where a matching graph can be chosen by user interaction.

Another tool that supports implicit rule scheduling is AGG (Tántzer97). As said previously, in principle, Graph Grammars also use implicit rule scheduling, as there is no order of rule application at all. Sometimes, like in AGG, the first graph that must be applied (the start graph) is given.

2.8 The schism between implicit and explicit rule scheduling

There is a schism between implicit and explicit rule scheduling. It turns out that transformation tools only tend to support one of the two given paradigms of rule scheduling. Therefore, the choice between implicit and explicit rule scheduling has to be made at the moment when a tool is chosen, that is, before the design and implementation phase of the development of the transformation. This is in contrast to usual tool (and language) choices in other software projects, where this choice depends on the project's architecture rather than on its design or implementation. For example, when developing a web application, the developer might want to use Ruby on Rails or the Spring framework. When a developer wants to program artificial intelligence, he might choose Lisp, Prolog or Python. These choices can be easily made because certain languages, frameworks and tools have proven to offer the right concepts for solving these problems.

This is not the case in the choice of a transformation tool. The choice of a tool, and thus of a rule scheduling paradigm, depends on the developer's implementation of the transformation rather than the architecture or nature of the project. For example, when a developer feels he will need many conditionals, and will want to heavily define the control flow of his transformation, he will want to use a tool that supports explicit rule scheduling. On the other hand, when a developer wants to keep the order of execution of rules as abstract as possible in order to avoid over-specification of his transformation, he will want to use a tool that supports implicit rule scheduling. This design choice is part of the implementation phase of the transformation, and should not be made in this early development phase. Unfortunately, when choosing between the present transformation tools, a developer is forced to make such a choice at the very beginning of the development cycle.

This problem becomes even more apparent when thinking of a framework for model driven development. This framework would contain a transformation tool and numerous useful transformations between different modeling languages. On top of that, the framework must allow developers to add new modeling languages and transformations for these new languages. When choosing tool support for such a framework, there are three options.

2.8.1 Solution 1 - Always using the same transformation tool

In the first solution, the framework forces the user to model all transformations using either implicit or explicit rule scheduling, possibly making some transformations a lot less readable, too verbose and full of code smells. This is the current situation. In fact this is not a solution to the problem, it is evading it and dealing with the consequences.

2.8.2 Solution 2 - A bridge between data representations for different tools

In the second solution, the framework includes more than one transformation tool, allowing both implicit and explicit rule scheduling. This would allow developers to choose their paradigm for each different transformation model, but it does not solve the previously stated problem where the developer has to choose a paradigm too early in the development cycle. A developer should be able to use both paradigms in the same transformation definition.

On top of that, the models resulting from a transformation execution performed by these tools must be somehow interchangeable between the different transformation tools, to allow the models to be kept synchronized (Kleppe03). This would be possible if all the tools in the framework would use the same language for data representation for its models and metamodels, such as XMI (OMG05) or GXL (Holt00). Unfortunately, the lack of a standard hinders this option. However, for example GXL is supported by a number of tools, so exchanging models between GXL tools is of course no problem. Using tools that support a different data representation language can become very cumbersome however, but transformations between these languages exist. For example, MOF based XMI to Ecore/EMF based XMI has been described as an XSLT transformation (Gerber03).

2.8.3 Solution 3 - Using a tool supporting both paradigms

A better solution would be using one tool, or transformation language, that supports both paradigms. That way, there can be no interchangeability problems of the transformed models. This tool or language would be hybrid with respect to rule scheduling. This language can be either completely new or originate from an existing language that already supports one of the paradigms.

Developing a new language from scratch is not necessary when there are already so many qualified languages. Extending a language seems to be a better idea. The extended language must allow new language constructs to model the features of the opposing paradigm. In other words, the extended language should allow all of the features listed in Table 2.3. Applying this to the two given tools, this means that we have a choice between either implementing ordering, conditionals and iterations in AToM³, or implementing nondeterministic scheduling, layers and priorities in MoTMoT.

When extending a language, it is important to take into account that the new language constructs should be easy to use and intuitive. It is hard to think of visually intuitive constructs for ordering, conditionals and iterations to extend a language that uses implicit rule scheduling. In Chapter 3 it turns out that implementing implicit rule scheduling (i.e.

Constructs for implicit rule scheduling	Constructs for explicit rule scheduling
<i>Basic language constructs</i>	
Nondeterministic scheduling	Ordering Conditionals
<i>Optional language constructs</i>	
Layers Priorities	Iterations

Table 2.3: Language constructs in implicit and explicit rule scheduling

nondeterministic scheduling, layers and priorities) in a language that supports explicit rule scheduling can be done by adding readable and intuitive constructs that are easy to use.

2.9 Conclusion of this chapter

In this chapter the context of this thesis was introduced. In model-driven engineering, finding a suitable language for modeling the indispensable model transformations is essential. In this so-called transformation language, there are two separate paradigms with respect to rule scheduling: implicit and explicit rule scheduling. AToM³ is an example of a tool that uses implicit rule scheduling, and MoTMoT is an example of a tool that uses explicit rule scheduling. As tools only support the one or the other, this reduces the ability of the language as a flexible, easy-to-use, easy-to-read transformation language, while this is a very important property for models (if not everything) in a volatile MDE environment and transformation languages in particular. That is why a transformation language with both implicit and explicit (i.e. hybrid) rule scheduling would be very useful. It turns out that extending a language that supports explicit rule scheduling would be the best choice.

ND-SDM - Hybrid rule scheduling in SDM

This chapter introduces some language constructs for the integration of implicit rule scheduling in an imperative language in order to allow both implicit and explicit rule scheduling. The language constructs will be implemented in MoTMoT, but can be implemented in any tool that supports Story Driven Modeling (SDM) as transformation language. The extended language that arises from these new constructs will be called ND-SDM (nondeterminism in SDM) throughout this thesis.

First, a new construct for nondeterminism is introduced that allows modeling the different rules without having to impose an order on them. Moreover, the syntax allows both paradigms to be used simultaneously in the same transformation model. When thinking of the exact meaning of this construct, it turns that multiple possibilities arise. Therefore, all possibilities that seem relevant are analyzed and classified.

Next, some more language extensions are presented that implement popular scheduling concepts, namely layers (as in AGG) and priorities (as in AToM³), that are available in tools that support implicit rule scheduling.

The resulting ND-SDM language is hybrid (imperative as well as declarative) with regards to rule scheduling. ND-SDM is not more *expressive* than the SDM language (in fact, they are both Turing complete). This means that a model written in ND-SDM can be rewritten in Story Diagrams. This property will be used in the implementation of ND-SDM in Chapter 4. The usefulness of this language is shown in Chapter 5, where a few case studies are given.

To stay in accordance with the UML profile for SDM like MoTMoT envisions, this implementation of ND-SDM must also be UML compliant. Similar to the syntax of the SDM constructs in MoTMoT, UML *Stereotypes* and *TaggedValues* (OMG01) can be used as syntax for the new constructs. When implementing ND-SDM in other SDM tools, alternative visualizations for these constructs can be easily thought of.

3.1 Implicit rule scheduling

As stated previously, there are several constructs in SDM Story Diagrams to allow explicit rule scheduling. This section presents syntax and semantics for a new language construct that allows nondeterministic scheduling. Hence, this new construct is part of the new ND-SDM language.

3.1.1 Syntax

As previously said, the new language construct should be UML compliant. This limits the options for visualization of the new construct. By contrast, in Fujaba (Fujaba07) for example, it would be possible to visually show that rules are scheduled nondeterministically by dividing up the state in question into a number of sections, according to the number of states that are scheduled nondeterministically. In each section, the pattern of each rule can then be displayed. Unfortunately, this is not possible for ND-SDM if UML compliance has to be maintained. However, a very similar notation to *motmot.transprimitive* (see also Section 2.6) can be used.

Figure 3.1 shows in a generic example how rules can be scheduled in MoTMoT using UML compliant Activity Diagrams (see also Section 2.6). A diagram written in SDM is given, and as a consequence it employs explicit rule scheduling. Inside each state, on the first line, there is a rule name, like `Rule A`. The tagged value *motmot.transprimitive* is visualized between braces on the second line. Its value contains the reference to a UML *Package*, in this case named `SP A`, containing the story pattern modeled as a UML Class Diagram (Van Gorp08a). In effect, the value of this tagged value states which pattern has to be evaluated when entering the state. This is different from original Story Diagrams, where rules are visually embedded in the Activity Diagram. The transitions show the direction of scheduling.

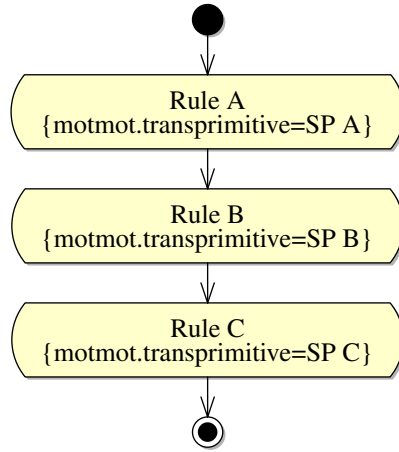


Figure 3.1: Three rules scheduled consecutively in MoTMoT

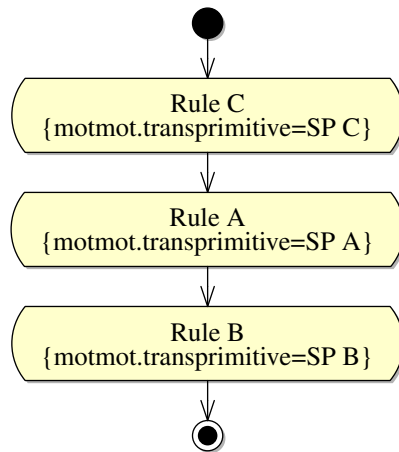


Figure 3.2: Three rules in another order

When one wants to schedule Rules A, B and C implicitly, MoTMoT fails to offer a solution. In MoTMoT, only explicit rule scheduling is possible, so an order has to be imposed when scheduling the rules. In this case, every Story Diagram where the rules are scheduled in any order, as in Figures 3.1 and 3.2, suffers from overspecification.

This chapter proposes a solution for this problem with a new language construct in the new language ND-SDM. An example of what this construct would look like is shown in Figure 3.3. The three rules are scheduled inside one state, disposing of the notion of order in the evaluation of the rules. In this new language construct, the three rules are summed up as a *motmot.transprimitiveND* tagged value. Such a state is called a nondeterministic state throughout the rest of this thesis. More general, a nondeterministic state can reference more than one UML Package and chooses nondeterministically in which order the packages are evaluated, hence "ND" in the name. When control arrives in this state, the rules are scheduled implicitly. Once these rules have been evaluated, control switches back to explicit rule scheduling and follows the outgoing transition of the state.

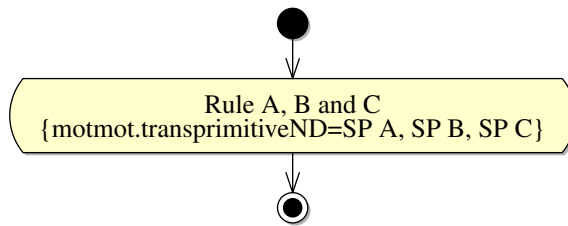


Figure 3.3: Three rules scheduled implicitly in ND-SDM

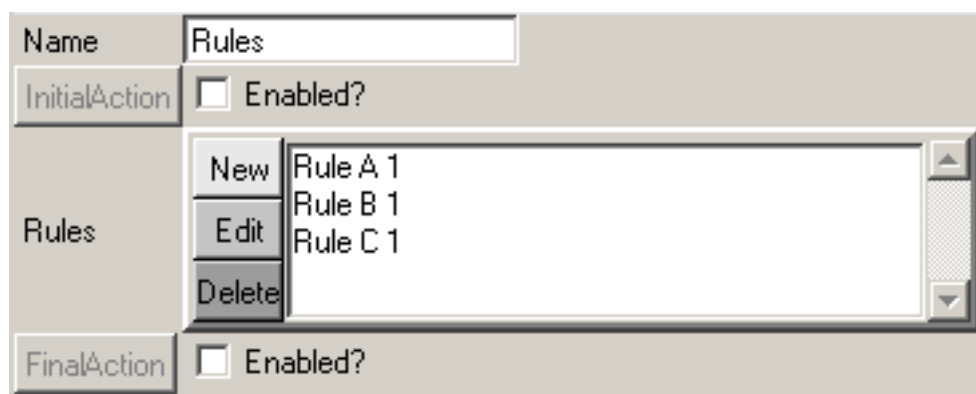


Figure 3.4: Three rules scheduled implicitly in ATOM³

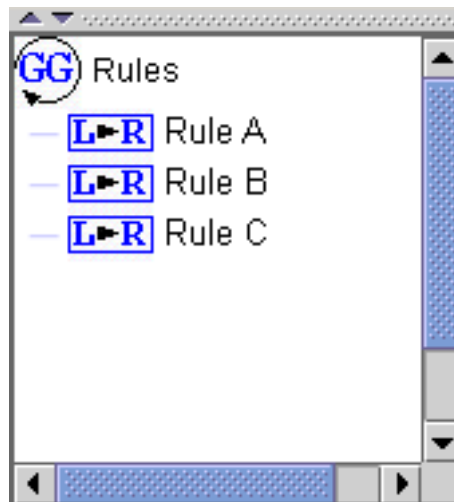


Figure 3.5: The three rules scheduled implicitly in AGG

One can argue that the *motmot.transprimitiveND* tag still imposes an order to the rules, as the rules are visually summed up in a particular order. However, in Story Diagrams the semantics of UML Activity Diagrams are followed. First, by definition only transitions impose an order on states. Second, when tags reference multiple values, there is by definition no ordering between these values (OMG01). Moreover, in tools that support implicit rule scheduling, the "listed" rules are also subject to a visual order. For example, an implementation in ATOM³ of the three rules is shown in Figure 3.4. The rules are still visually ordered, but they all have the same priority (i.e. 1) and can be evaluated in any order. AGG offers a similar list-like visualization, as in Figure 3.5.

A nondeterministic state does not only allow some rules to be scheduled implicitly. Besides story patterns, it can also reference whole Activity Diagrams. This allows the modeler not only to use implicit rule scheduling inside a diagram based on explicit rule scheduling, but also supports explicit rule scheduling in a diagram based on implicit rule scheduling. In this way, the language construct truly allows hybrid scheduling.

However, a small problem may occur when putting many states into one nondeterministic state. Constraints in *motmot.constraint* tags can not be used anymore on nondeterministic states, as it is impossible to derive to which of the rules these constraints apply. This is not a big problem however, as these constraints can be put on any node of the pattern it relates to, while maintaining the same behavior. Alternatively, instead of a Story Pattern, a new Story Diagram can be referenced by the nondeterministic state, and this Story Diagram can obviously include states with *motmot.constraint* tags.

3.1.2 Semantics

An obvious question arises. What is the exact meaning of this state? Nondeterministic states can be implemented according to many different criteria. Can the same rule be evaluated many times? Will the algorithm stop after a rule failed to match? Many different interpretations can be considered, and each of them can turn out to be useful in a certain context.

Two useful variants

In particular, two different interpretations will turn out to be most useful in the case studies in Chapter 5. As an example for the first variant, consider a housewife managing her household. There are three tasks she must attend to:

- mop up a dirty room;
- help a child with his or her homework;
- clean a dirty window.

There are several rooms, children and windows in her house, and these tasks have to be executed for each room, child or window. It doesn't matter in which order all these tasks are performed, but the job has to be done by the end of the day. Because the scheduling order is not important, the tasks are best modeled using implicit rule scheduling.

The top side of Figure 3.6 models the household using the new language construct of ND-SDM presented in Section 3.1.1. The housewife has to get up before performing any household tasks, so this is scheduled explicitly. After the tasks are done, her spouse takes her to dinner, because he is delighted with the finished work.

The states `Get up` and `Have dinner` are MoTMoT's usual *motmot.transprimitive* states (see also Section 2.6). The state `Manage household` is a nondeterministic state containing a *motmot.transprimitiveND* tag with references to the rules representing mopping, helping and cleaning, similar to the new ND-SDM construct introduced in Section 3.1.1. However, there is a subtle difference with the nondeterministic state of Figure 3.3: the nondeterministic state contains a *«loop»* stereotype. This stereotype denotes that the three different tasks can be executed more than once. In general, stereotypes on the nondeterministic state can be used to indicate which variant is applied.

As said previously, one of the referenced packages in `Manage household` could contain another Activity Diagram with several scheduled rules instead of just a rule. In this

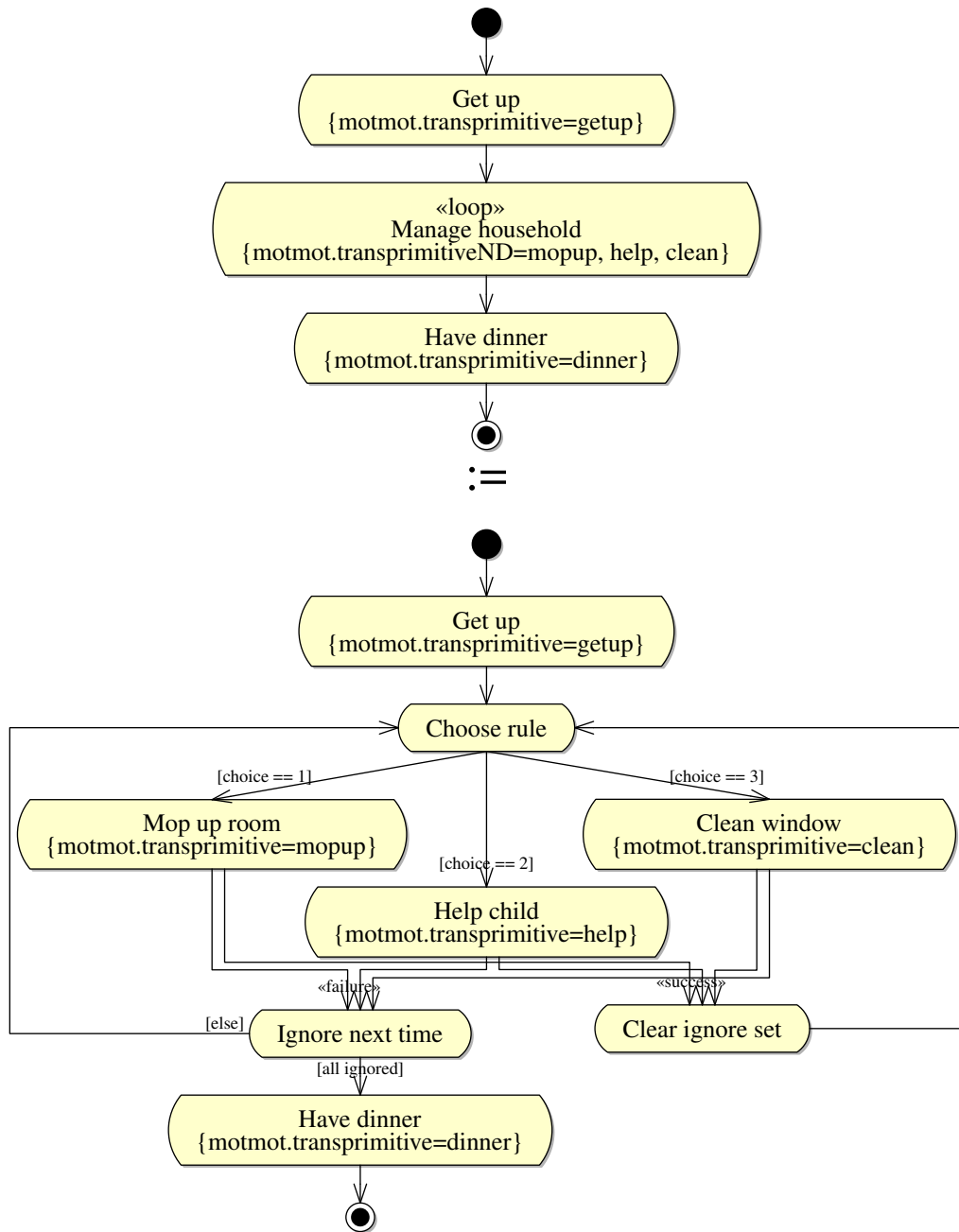


Figure 3.6: A household modeled using hybrid (top) and explicit (bottom) rule scheduling

way it could be modeled more in detail how, for example, the housewife cleans a window. The Activity Diagram could start with the initial state, with a transition to a state `Fetch tools`, from which a transition goes to a nondeterministic state `clean inner and outer window-glass`, from which a transition goes to a nondeterministic state `dry inner and outer window-glass`, from which a transition goes to a state `Put away tools`, from which a transition goes to a final state.

In the bottom diagram of Figure 3.6 it is modeled what the top diagram exactly means. Both are equivalent, but the bottom diagram uses plain old explicit rule scheduling. The nondeterministic state is replaced with several states that are modeled explicitly, thus according to the existing language featured in MoTMoT. After getting up, a rule is chosen¹, meaning that a household task is chosen. According to this choice, one of the task states is entered, for example the `Mop up room` state. Because all rooms have yet to be mopped, a certain room is chosen and eventually mopped. The `<<success>>` transition is followed to the `Clear ignore set` state, which is explained below. From the `Clear ignore set` state the transition is followed to the `Choose rule` state. In other words, a task is finished (that is, one room is mopped, but there might be several other dirty rooms), and the housewife is ready to choose the next task.

Having completed a number of tasks, it might occur that for example the `Mop up room` task is chosen, but there are no rooms left to be mopped (i.e. the rule doesn't match). In this case, the `<<failure>>` transition is followed to the `Ignore next time` state. There, the rule, in this case `Mop up room`, is added to a set called the `ignore set`. This means that the housewife has concluded that all rooms are cleaned, so there is no point in trying to find dirty rooms at some later point that day, and she will ignore the `Mop up room` task. Then, the transition is followed to the `Choose rule` state. Now, a new rule is chosen with the exception of `Mop up room`, as this rule would not match anyway. In a following iteration, a window is cleaned for example. The transition to `Clear ignore set` is followed. In this state, the `ignore set` is cleared (i.e. no tasks will be ignored when choosing a new task), because it might occur that due to the completion of the previous task, an ignored task might match again. In this case, the soapy water used when cleaning on the window might spread on the floor, so the floor has to be mopped again. So next iteration, the `ignore set` is empty again. If all tasks are done, all three would automatically end up in the `ignore set`. Then, from the `Ignore next time` state, the transition is followed to the `Have dinner` state. An example execution is given in Table 3.1 for a household with three rooms (say `r1`, `r2`, `r3`), two children (say `c1`, `c2`) and four windows (say `w1`, `w2`, `w3`, `w4`). Note that in

¹In the current implementation of ND-SDM, this choice is made by a random number generator. It can also be done by a fixed choice, or by letting the user choose at run-time. However, in this context, it is not that important how the choice is really made. See also Chapter 4.

iteration 9, by cleaning window 3, room 3 gets dirty again. Every other time the housewife cleans a window, she manages to keep the room clean. After iteration 14, the `Have dinner` state is entered.

#	Rule chosen	Subgraph	Tasks to be done	Ignore set
1	Mop up room	room 2	r1,r3,c1,c2,w1,w2,w3,w4	{}
2	Clean window	window 4	r1,r3,c1,c2,w1,w2,w3	{}
3	Clean window	window 1	r1,r3,c1,c2,w2,w3	{}
4	Mop up room	room 1	r3,c1,c2,w2,w3	{}
5	Help child	child 1	r3,c2,w2,w3	{}
6	Mop up room	room 3	c2,w2,w3	{}
7	Clean window	window 2	c2,w3	{}
8	Mop up room	-	c2,w3	{mopup}
9	Clean window	window 3	r3,c2	{}
10	Help child	child 2	r3	{}
11	Mop up room	room 3	-	{}
12	Help child	-	-	{help}
13	Mop up room	-	-	{mopup, help}
14	Clean window	-	-	{mopup, help, clean}

Table 3.1: A possible execution order of the household tasks

It turns out that the behavior of this variant can be described briefly as *keep matching rules until all failed to match*. Because each task can be done more than once, a `<<loop>>` stereotype² is used on the nondeterministic state at the left hand side.

An example of the second variant is given in Figure 3.7. It models the way a sous chef composes a plate. The sous chef needs to apply asparagus, beef and mashed potatoes. Note that this nondeterministic state does not have a `<<loop>>` stereotype, because the same rule must not be applied twice. Again, the order in which these ingredients are applied to the plate is not important. However, in a fancy restaurant, the presentation of the plate is essential. If the sous chef fails applying any of these ingredients properly, the plate goes wrong, and the sous chef has to start over by fetching a new plate.

According to the top diagram of Figure 3.7, the sous chef first has to fetch a plate and can then apply the ingredients. When failing, the sous chef has to start over, when succeeding,

²Originally in MoTMoT, this stereotype enumerates all possible subgraphs one by one (and only once) as explained in Section A.6.2 of (Zündorf02). This is not the case in this usage of the stereotype. The same subgraph can be evaluated several times. The stereotype is used to denote the iterating nature of the variant.

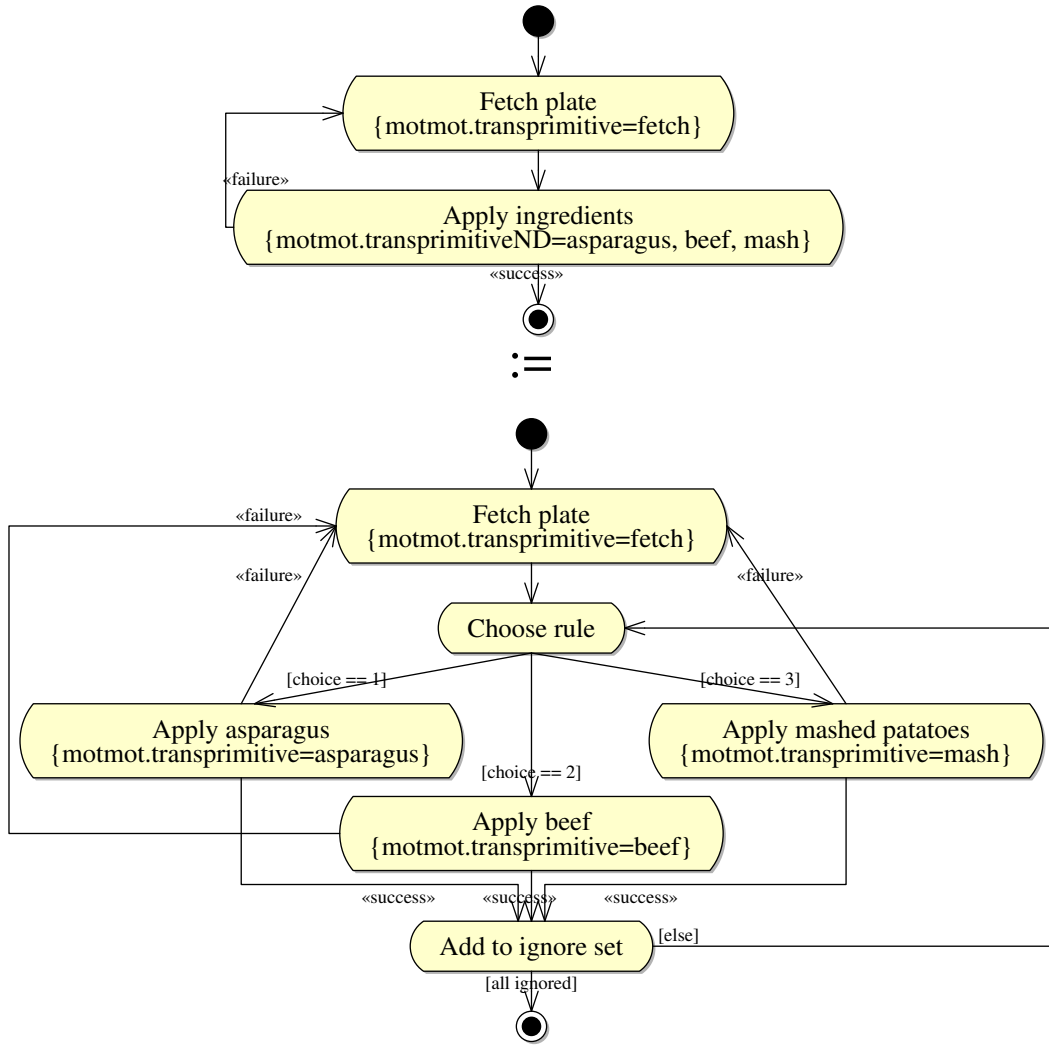


Figure 3.7: A model of the composition of a plate in a restaurant using hybrid (top) and explicit (bottom) rule scheduling

the sous chef's job is done. In the bottom diagram, the meaning of the top diagram with the nondeterministic state in particular is modeled in detail. After fetching a plate, the sous chef can apply one of the three ingredients. When successful, the sous chef applies another ingredient (not the same because the performed rule is added to an ignore set), and finally the sous chef applies the final ingredient. However, if he fails, he starts over. In this example, the variant means *match each rule once until one fails to match*. Each rule can be executed or evaluated at most once, which is illustrated by the absence of the `<<loop>>` stereotype.

As a conclusion, it seems appropriate to define the ND-SDM language to behave like the first variant in the case of a `<<loop>>` stereotype, and like the second variant when there is no stereotype. This means that a modeler is able to use two variants of nondeterministic scheduling. However, there may be more useful variants besides these two.

All possible relevant variants

The previous section presented two variants of a nondeterministic state that turn out to be most useful (see also Chapter 5 for case studies). However, there are many other variants, which may also turn out to be useful in a certain context. Therefore, it is useful to analyze all possible variants.

Let us analyze the two previous examples, with nondeterministic states interpreted as *keep matching rules until all failed to match* and *match each rule once until one fails to match*. They resemble each other in that there is a nondeterministic choice between some available rules. This choice is made iteratively. The variants differ in how these iterations are terminated. In the first variant, termination only occurs when all rules failed to match. In the second variant, termination can occur when all rules matched at some point, or when a rule fails. Also, in the second variant, any rule can only be evaluated once. In other words, when a rule is evaluated, it must be ignored in the next iteration.

In conclusion, different interpretations can be made according to whether a rule, or all rules, matched after evaluation (succeeded), or failed to match (failed). There are numerous variants that can be useful in a certain context, but it turns out that four dimensions cover the relevant variants:

- terminate/continue after one success or terminate after all succeeded at some point;
- terminate/continue after one failure or terminate after all failed at some point;
- remove/keep a rule after success;
- remove/keep a rule after failure.

It turns out that the first two dimensions address the termination criteria of the nondeterministic state. These two dimensions are used in disjunction (OR). They can also be used in conjunction (AND), but those variants (e.g. *terminate after all succeeded and one failed*) turn out to be somewhat far-fetched. The last two dimensions address the per-rule repetition criteria and they must be used in conjunction. So taking these possibilities into account, it means that we will see 36 ($3 \cdot 3 \cdot 2 \cdot 2$) combinations.

All variants ignore failed rules (in the aforementioned `ignore` set) until any rule succeeds in order to boost performance and allow pseudo-nondeterministic implementations (which would impose an order on the rules, rather than choosing a rule at run-time). It also means that when all rules fail consequently to match, the nondeterministic state is exited. In this way, endless loops that are caused by rules that do not match, are avoided.

Taking these criteria into account, the variant of the household problem (*keep matching rules until all failed to match*) and the restaurant problem (*match each rule once until one fails to match*) can be described as in Table 3.2. Note that in the restaurant problem, it makes no difference whether the variant is defined to keep or remove a rule after failure, because the iterations are ended after the first failure. This suggests that some combinations of the criteria might result in the same variant. All the possibilities are presented in

The household problem	The restaurant problem
continue after one success	terminate after all succeeded at some point
continue after one failure	terminate after one failure
keep a rule after success	remove a rule after success
keep a rule after failure	keep a rule after failure

Table 3.2: The properties of the household problem and restaurant problem

Tables 3.3, 3.4 and 3.5 below, where the first two dimensions are combined in rows and the last two are combined in columns. As said, many combinations result in the same variants. In fact, there are 14 different variants, and each has its own color.

The tables can be read as follows: the upper left combination of the diagram, part of variant `try_one`, follows criteria $S^1 \vee F^1 / S^- \wedge F^-$ (see legend). According to termination criteria $S^1 \vee F^1$, the iterations are terminated after a successful rule or a failed rule (i.e. after the evaluation and possible execution of the first rule). According to repetition criteria $S^- \wedge F^-$, a rule is ignored after it failed and after it succeeded (i.e. the same rule can never be evaluated again). In this case, the repetition criteria do not really matter, since the variant

terminates iterations after the first rule. Therefore, all possibilities with termination criteria $S^1 \vee F^1$ (i.e. the first row) are the same variant: `try one`.

The meaning of each possibility in Tables 3.3, 3.4 and 3.5 is illustrated with a simple, concise diagram. Since all variants in the tables below share similarities, their diagrams all use a combination of the following states:

- `? (choose)`: this state chooses and evaluates an available rule. An available rule is a rule that is not in `ignore set` (as explained in the household example) or `ignore2 set`. When the chosen rule matches, its side-effects are executed and the `«success»` transition, here denoted by `s`, is followed. If the rule doesn't match, the failure transition, here denoted by `f`, is followed;
- `i` and `i2 (ignore and ignore2)`: these states put the evaluated rule in an ignore set, respectively `ignore set` and `ignore2 set`. When an outgoing transition has a guard `[i full]` or `[i2 full]`, it means that every available rule is in respectively `ignore set` or `ignore2 set`. `ignore set` and `ignore2 set` are technically the same, but in some variants two different ignore sets are needed;
- `_i (clear ignore)`: this states empties the ignore set `ignore set`. This means that the ignored rules can be chosen again in the choose state;
- `r` and `r2 (remember and remember2)`: these states put the evaluated rule in a set that remembers it. The rules that are elements of these remember sets are not ignored by the `choose` state. These sets are used to track which rules have been already evaluated and/or executed.

These variants do not take the return value of the nondeterministic state into account. In MoTMoT, a `motmot.transprimitive` state returns `true` when its rule matched or `false` when it didn't, which means that respectively a `«success»` link or a `«failure»` transition will be followed. Hence, variants of nondeterministic states should also take return values into account.

A natural policy for the return value would be that if the state is exited according to a termination criterion it implemented, it returns `true`. If the state was exited abnormally, `false` is returned. For example, consider the variant where termination occurs after all rules ever succeeded (class `S*` in Table 3.4). When all rules fail to match consequently before all rules ever matched, the nondeterministic state would be exited abnormally, returning `false`. Note that, when this return value policy would be applied to the variants, there would be more than only 14 different variants. For example, variant A following criteria $S^* \vee F^* / S^- \wedge F^-$

would be different from variant B following criteria $S^*/S \wedge F^-$. A would always return *true* if all rules matched or all failed to match, but it would return *false* if some rules matched and others didn't. On the other hand, B would return *false* when one of the rules failed to match.

However, this policy might also result in unnatural behavior in some cases. The variant of the household problem (the bottom right variant *match until all fail*, ϕ/ϕ) could never return *true*, as it can only terminate "abnormally" by failing to match all rules consecutively. However, it might be more desirable to some modelers that *true* is returned if one rule matched at some point. Other modelers want the variant to return *true* if all rules matched at some point. This discussion can occur for several other variants too.

The purpose of this section is to show that these differences must be considered when implementing a variant for the implementation of a nondeterministic state. 14 variants are presented in Tables 3.3, 3.4 and 3.5, but infinitely more are possible (e.g. taking the return value into account, looping exactly five times, etc.). In other words, all possible variants can not be implemented when realizing ND-SDM. Moreover, doing so would make it harder to come up with suitable syntactical differences (in the form of stereotypes and tags on the nondeterministic state) between all variants. The section also shows that the exact meaning of variants can differ slightly from modeler to modeler. Therefore, when discussing the implementation in Chapter 4, it is taken into account that it should be easy for a modeler to add his own variants in the implementation of his own transformation.

Legend

S^1 : terminate after one success S^* : terminate after all ever succeeded

F^1 : terminate after one failure F^* : terminate after all ever failed

S^- : remove a rule after success

F^- : remove a rule after failure

?: chooses a rule

i: add rule to ignore set

i2: add rule to another ignore set

_i: ignore set is cleared

r: rule is remembered

r2: rule is remembered in another set

+: rule matched

-: rule didn't match

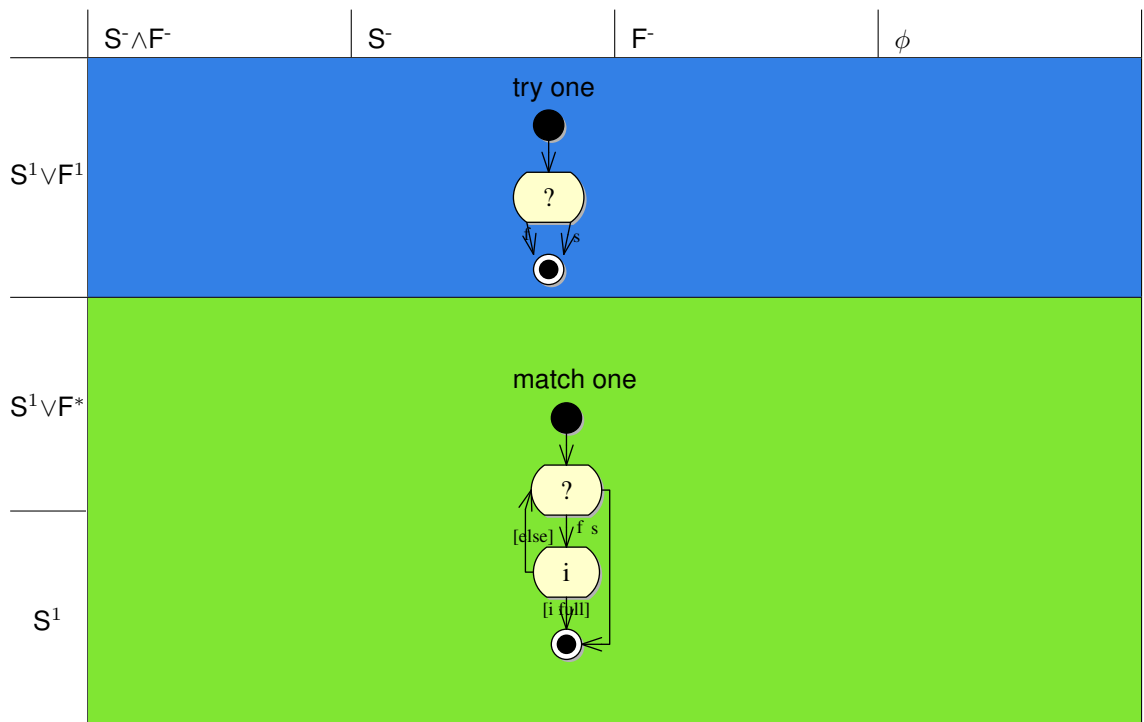


Table 3.3: Possible variants of implicit rule scheduling

	$S^- \wedge F^-$	S^-	F^-	ϕ
$S^* \vee F^1$	<p>match all once until one fails</p>			
			<p>match until all matched once or one fails</p>	
$S^* \vee F^*$	<p>try all once</p>	<p>match all once until all failed once</p>	<p>match succeeding until all matched once</p>	<p>match until all matched once or all failed once</p>
S^*		<p>match all once</p>	<p>match succeeding until all failed once</p>	<p>match until all matched once</p>

Table 3.4: Possible variants of implicit rule scheduling (continued)

	$S^- \wedge F^-$	S^-	F^-	ϕ
F^1	<p>match all once until one fails</p>		<p>match until one fails</p>	
F^*	<p>try all once</p>		<p>match succeeding until all failed once</p>	<p>match all until all failed once</p>
		<p>match all once</p>		<p>match until all fail</p>
ϕ				

Table 3.5: Possible variants of implicit rule scheduling (continued)

3.2 Layers

Literature proposes helpful language constructs for implicit rule scheduling. Because the goal of designing ND-SDM is to facilitate the modeling of model transformations, all remaining constructs of Table 2.3 (i.e. layers and priorities) should be available in ND-SDM. As it turns out, they can be easily implemented.

In 1997, Rekers and Schürr introduced layers as a means to order implicitly scheduled rules (Rekers97). The need to order rules can be interpreted as an early need for a hybrid transformation language with respect to rule scheduling. Layers are for instance implemented in AGG. Layers allow rules or groups of rules to be evaluated in a certain order. This behavior can be easily obtained in ND-SDM by simply using transitions between different (nondeterministic) states as shown in an example in Figure 3.8.

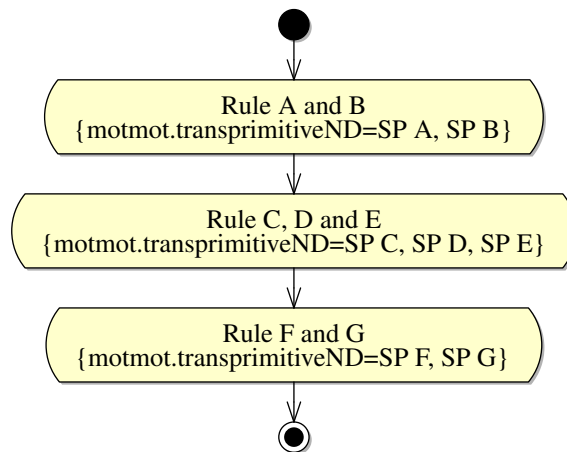


Figure 3.8: A series of implicitly scheduled rules with three layers

3.3 Priorities

Another scheduling mechanism for implicit rule scheduling is the use of priorities. The role of priorities in rule scheduling was first described by Kaplan and Goering in 1989 (Kaplan89). It is implemented in AToM³ (see also Section 2.7). Priorities are similar to layers. In case of priorities however, rules with a higher priority are always evaluated first. Only when rules with a higher priority fail to match, rules with a lower priority can be evaluated. Priorities are realized in AToM³.

The first question that arises when thinking of realizing priorities in ND-SDM is how priorities will be visualized. One could argue that rules ordered implicitly with priorities must be modeled into one state, because they are in the first place a set of rules scheduled implicitly. It would be very hard to achieve this syntactically while maintaining UML compliance. For instance, it is not possible to have a set of ordered pairs (`UmlPackage,priority`) as tagged value. So a notation in the form of $\{motmot.transprimitiveND=(SP\ A, 1), (SP\ B, 2), (SP\ C, 2)\}$ is impossible.

It would however be possible to omit the priority numbers and make the order in which the packages are summed up decisive. In that case, a new tag must be defined, because if *motmot.transprimitiveND* would be used, suddenly an order (in the sense of priorities) would be imposed on all rules, which is exactly what we tried to avoid. So suppose that *motmot.transprimitiveNDprior=SP A, SP B, SP C* would mean that rule A has priority over rule B, which has priority over rule C. But then two problems arise. First, every single rule has a different priority. Again, imposing an order on rules is exactly what we wanted to avoid in the first place, so rules with the same priority should be scheduled like nondeterministic states described earlier in this chapter. Second, as previously said, tagged values are by definition unordered (OMG01). It would be very confusing to consider the value of *motmot.transprimitiveND* as a set and the value of *motmot.transprimitiveNDprior* as a list.

On the other hand, one could argue that the availability of transitions can offer an elegant way of describing priorities. Indeed, using different states with "special" transitions between them are visually clarifying for the modeler. The transitions that describe the priorities can be distinguished from regular transitions (which describe explicit scheduling) by adding a stereotype. The top diagram of Figure 3.9 shows the syntax of eight rules scheduled using priorities. Two or more states are scheduled with priorities if there are *«nextPriority»* transitions between them. If there is an outgoing *«nextPriority»* transition from a state, another outgoing transition from that state is never followed, except if it is an *«each time»* transition from a *«loop»* state. Rule A and B both have top priority, followed by C, D and E, and F and G have lowest priority. While the presence of some transitions would suggest otherwise, this is an example of rules scheduled completely implicitly in ND-SDM. This means that this example can be copied straight into AToM³, as shown in Figure 3.10. The priorities are visualized by integers next to the rule name. The lowest number has highest priority. Note that in the ND-SDM solution, the separation between different priorities is syntactically more distinct.

The bottom diagram explains the meaning of priorities in ND-SDM as an equivalent in which the *«nextPriority»* transitions are converted to regular transitions known by SDM.

When a state returns *false*, the rules of the next priority can be evaluated. When a state returns *true*, rules of the top priority have to be evaluated again, as they might match now because side-effects of the execution could have changed the model. Note that the return value of the variant of a nondeterministic state, as discussed in section 3.1.2, is very important in this context. Therefore, in order to ensure proper behavior, priorities should be used only with a regular state or with a nondeterministic state of the variant *match one* (see also Table 3.3), with the variant returning *true* if a rule matched and *false* if all rules failed to match. These states will then exhibit an iterative behavior, as the state with highest priority will be visited again after a rule matched.

There is another restriction on the use of priority transitions. When there is an outgoing `<<nextPriority>>` transition, there cannot be an outgoing `<<success>>` or `<<failure>>` transition because this would semantically result in two outgoing `<<success>>` or `<<failure>>` transitions, which is illegal in SDM. A `<<loop>>` stereotype on a regular state is allowed, and this state can have an outgoing `<<each time>>` transition. These restrictions can be added to ND-SDM as OCL constraints (OMG06) to the model of the transformation language itself. However, since OCL constraints are not used yet in the metamodel in MoTMoT, this is beyond the scope of this thesis.

3.4 Caveat

The nondeterminism of implicit rule scheduling can lead to unexpected results: in many cases, one rule for example creates elements that are used by another one and deleted by yet another rule. Since such dependencies can be introduced accidentally, dedicated analysis support is desirable in transformation tools that support languages with implicit rule scheduling. For example, AGG offers a so-called Critical Pair Analysis (CPA) (Lambers08). To be applicable on the proposed hybrid language, CPA algorithms need to take into account nodes that are already bound from previously executed rules in a control flow. In this thesis, this issue is considered to be the responsibility of the programmer because not knowing (and not caring) in which order the rules are executed is part of the meaning of nondeterminism.

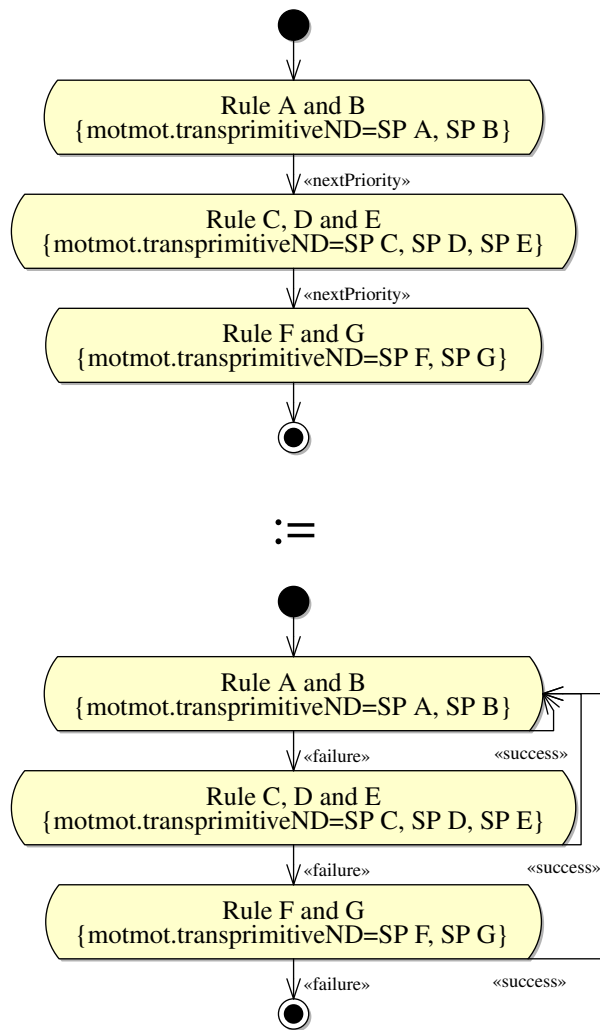


Figure 3.9: A series of implicitly scheduled rules with priorities and its meaning

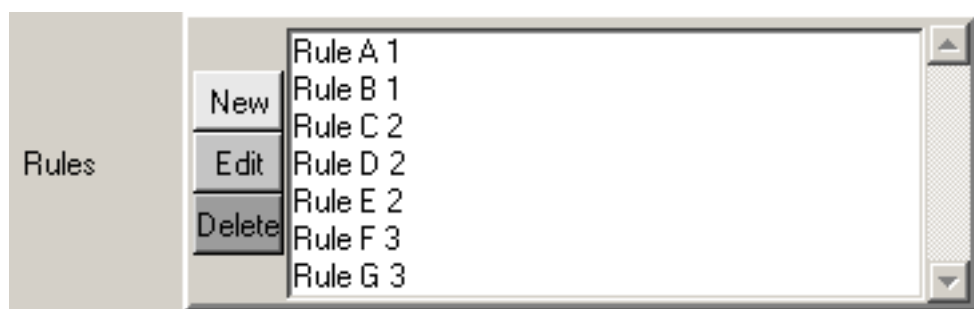


Figure 3.10: A series of implicitly scheduled rules with priorities in AToM³

3.5 Conclusion of this chapter

This chapter introduced ND-SDM as a language extended from SDM Story Diagrams. ND-SDM allows modelers to schedule their rules implicitly and explicitly. It thus allows hybrid rule scheduling. In defining this language, it is important that it remains UML compliant. The core extension is the construct for nondeterministic scheduling. Syntactically, many rules can be summed up in one so-called nondeterministic state (which can be thought of, in terms of explicit rule scheduling, as an atomic scheduling entity) instead of one, as in SDM. An equivalent with the same behavior as this nondeterministic state can be defined using only explicit scheduling. Therefore, it is possible to make the implementation of this construct a *preprocessing step* (a technique used very often in compiler design), which converts instances of the nondeterministic state into regular MoTMoT diagrams (see Chapter 4). When trying to define the semantics of such a nondeterministic state, it turns out that there are many different possible interpretations. Therefore, a number of useful variants are analyzed and classified into four dimensions. Also, the return value of the nondeterministic state itself is an important factor. Some variants appear more useful than others in certain contexts, or for certain modelers. As a consequence, it will be important for the implementation of the nondeterministic state that it is possible for the modeler to easily define his own variant, and plug it into the implementation. To make this convenient for the modeler, new personalized variants should be plugged in into the implementation without having to compile or interpret it again. Also, it can be shown that the implicit scheduling concepts of layers and priorities can be added easily in ND-SDM. Layers do not need an extra language construct, and the construct for priorities can also be implemented as a preprocessing step.

Implementation of ND-SDM

In this chapter we will discuss how implicit rule scheduling is implemented in order to make ND-SDM transformations readable by the Core profile of SDM. The presented implementation is done in MoTMoT. First we will list some not-so-obvious requirements for the implementation. Then we will propose two techniques that will be used, being higher order transformations and prototypes. In the next section we will present the implementation itself, followed by a conclusion of this chapter.

4.1 Requirements

As discussed in the introduction of Chapter 3, the MoTMoT implementation of ND-SDM has to be UML compliant, just like the SDM language for MoTMoT. In that way the implementation is tool-independent, which means that it can be used on any UML compliant SDM tool.

The implementation has to support the following features:

- implicit scheduling of possibly mixed story patterns (rules) and Story Diagrams (a set of scheduled rules);
- the implementation of priorities, by implementing the meaning of the `<<nextPriority>>`

stereotype (see also Section 3.3). Layers are implemented without an additional construct (see also Section 3.2);

- the ability to easily add and remove new variants of nondeterministic scheduling (see also Section 3.1.2).

Besides these requirements, it is desirable that this implementation is platform independent. In other words, the SDM tool may not be changed, and the solution should be applicable for all tools implementing SDM. The Critical Pair Analysis mentioned in Section 3.4 is out of the scope of this thesis and is not implemented.

4.2 Higher order transformations

Higher order transformations (HOTs, or metatransformations) are transformations that operate on other transformations as their input or output (Varró04). In other words, by using a higher order transformation, a transformation definition itself can be transformed to another transformation definition. In MDE this is possible, since transformation models can be represented as models. According to Varró and Pataricza, a framework or tool is suited for higher order transformations when the following prerequisites are met:

- there is a representation of *instance-of* relations in the modeling space;
- transformation rules are represented as models.

The first prerequisite can be met either in an explicit way, by e.g. edges between instance and class, or in an implicit way, by e.g. instance names of the form `o: Class`. A model of the transformation language (i.e. the metamodel of a transformation definition) must be available in order to support higher order transformations. In MoTMoT, instance-of relationships are possible in the implicit way. The second prerequisite is also met by MoTMoT and SDM in general, as transformation models are represented in UML (in this case Class Diagrams and Activity Diagrams), just like first order models.

In my opinion, these prerequisites fit very well in the MDE vision, where models conform to metamodels (first prerequisite) and everything is a model (second prerequisite). This is especially applicable to the domain of tools with visual transformation languages (MoTMoT (Van Gorp07), AToM³ (De Lara02), AGG (Tántzer97), Progres (Schürr09a), Fujaba (Fujaba07)), which is the domain of languages in this thesis. However, not many tools support higher order transformations (yet), because generally the metamodel of the transformation language is not available. AToM³ for example does not feature a metamodel for

its Graph Grammar transformations. Also, some (older) tools with textual transformation languages sometimes tend to fail the second prerequisite.

Generally a tool has only one transformation language, so one might think that higher order transformations tend to be endogenous transformations (i.e. transformations with input and output models that conform to the same metamodel), as the metamodel of the input and output transformation model can only be that one transformation language. In this thesis, higher order transformations are used as a means to transform transformation models written in an extended transformation language to a well known transformation language. Both are the UML, so technically this is an endogenous transformation. The resulting transformation model can then be executed by any tool supporting the original language. This idea is presented by Muliawan (Muliawan08) and is used by Van Gorp et al. (Van Gorp08b) as a means to overcome the problem of portability and to allow reusing existing language constructs among transformation tools. A preface to this thesis also presented this idea at the Sixth International Fujaba Days (Meyers08).

4.3 Prototypes

In the context of this thesis, prototypes do not refer to software prototyping in software development, a practice in which basic versions of a software project are being developed. In this thesis, prototypes are generic models containing a partial implementation.

When used in a transformation, prototypes typically are a pre-generated, fixed part of the output model. In this case, the metamodel of prototypes is the same as the metamodel of the output model. When the transformation is executed, the prototype is copied into a designated place in the output model.

Prototypes can also be used for pattern matching¹, but then their metamodel is the same as the input model and the prototype is deleted at the end of the transformation execution. (In fact, they can even mix the metamodels of the input and output models, or have their own metamodel. In these cases, the transformation will either have to remove the copied prototype, or alter the copied prototype in a later stage of the execution, in such a way that it conforms to the metamodel of the output model at the end.) In any case, although the prototype is considered a part of the transformation, it is not written in the transformation

¹This idea is very similar to transformation patterns in AToM³, which actually use subgraphs that conform to the metamodel of the input/output model for its LHS/RHS.

language, but in the language of the input or output model. In terms of OMG layers, a prototype resides in the same layer as the input and output models, one layer lower than where the transformation model resides.

Using prototypes has the following advantages:

- because prototypes reside in a lower layer than the transformation model, concepts can be modeled much more concise (thus less error-prone) than when the same concepts would be modeled in the transformation model. The reason is that languages in lower layers offer more problem-oriented (less abstract) language constructs (Kleppe03);
- because prototypes reside in the same layer as the input and output models, they are very readable and understandable for modelers. After all, modelers tend to think in terms of their input and output model when modeling a transformation model;
- when the modeler keeps the prototype and the transformation model separate, the prototype can be used as a plugin. Then, the prototype can be replaced with another (resembling) prototype, without having to change the transformation model. This enables the modeler to reuse the same transformation, even as a black box. Because only models in the layer of the input and output model have to be changed, it means that even users of the transformation who are not familiar with modeling transformations can personalize the transformation without knowing of its implementation.

Separating the prototype from the transformation model has also a disadvantage. In the transformation model, there is no connection with the prototype through bound variables (see also Section 2.5). Elements will have to be bound again after creation in order to use them, which might come down to a slight overhead in patterns that use elements of the prototype.

As an example, suppose a transformation with a Class Diagram as output model. (The input model is not important for this example.) A transformation rule is implemented that creates a package containing a simple parent-child relationship. A parent of type Parent has a two-way association with a child of type Child. In Figure 4.1, it is implemented as an ordinary transformation pattern. This pattern gets quickly very verbose, unreadable and error-prone. There are a lot of elements with a `<<create>>` stereotype, which account for the creation of the two elements and their association.

In Figure 4.2, a prototype implementing the package is shown. It is concise and takes a lot less time to implement. It is a lot less error-prone and a lot more readable, as this

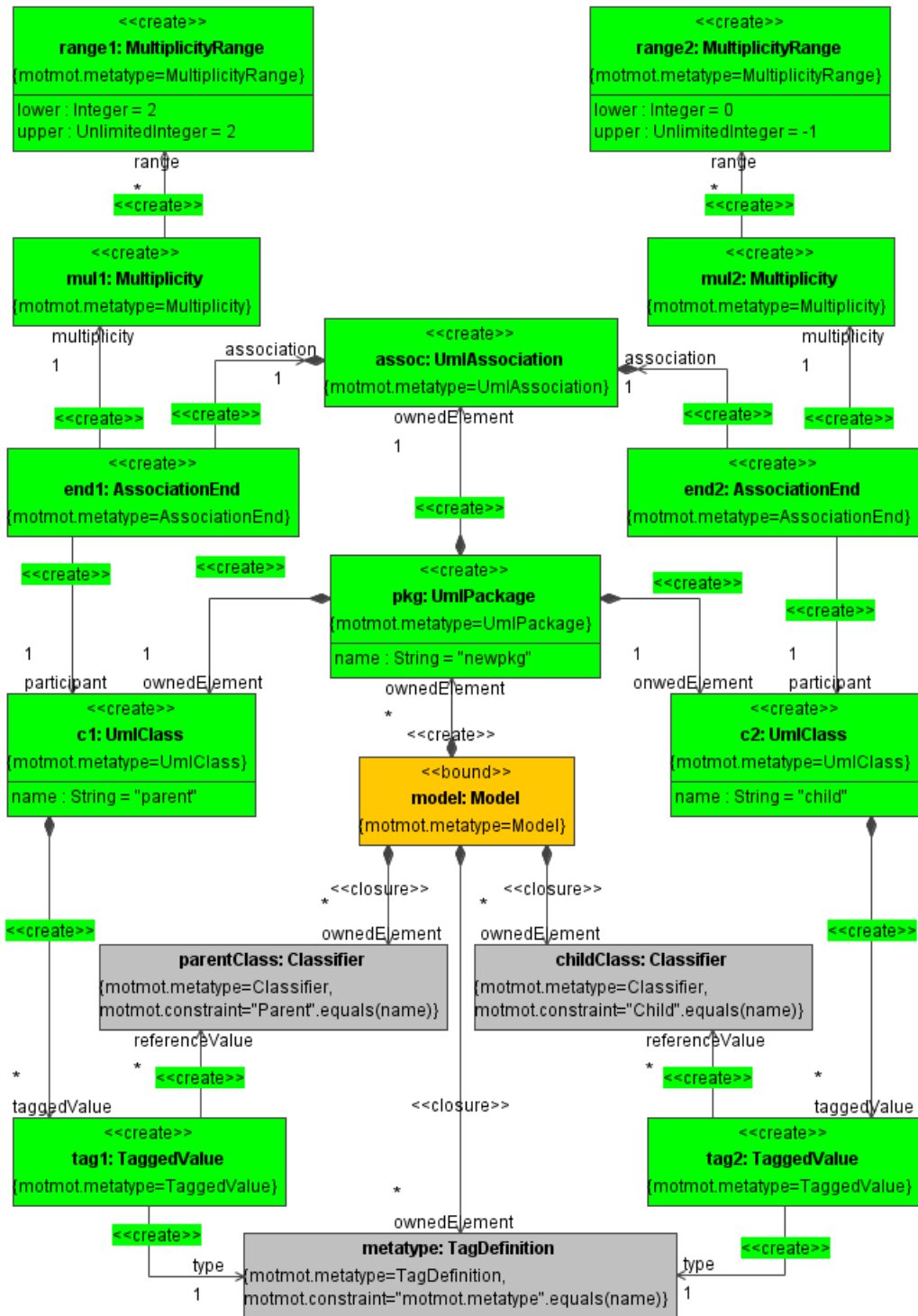


Figure 4.1: A transformation rule that creates a parent and a child and their association

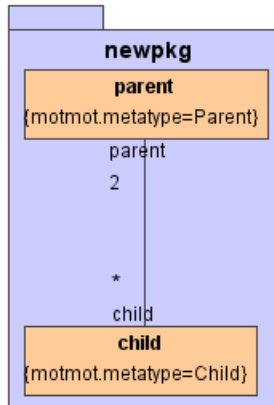


Figure 4.2: A prototype of the transformation rule

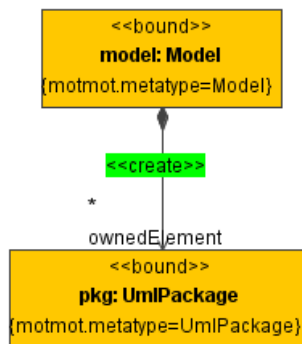


Figure 4.3: The integration of the prototype into the transformation rule

prototype is modeled using the metamodel of the output model. This metamodel is the true mental context of this transformation (together with the metamodel of the input model) as this prototype is the result that the modeler aimed for with the pattern of Figure 4.1.

Figure 4.3 shows the integration of the prototype of Figure 4.2 into the transformation. In fact, it is the replacement of the pattern of Figure 4.1. The package `pkg` is the copy of the package in the prototype. It is copied previously (i.e. in a previous state which performs the special operation of reading and copying), so therefore it is *bound*. The only thing left to do is putting the dangling copied prototype into the model.

The idea of using a combination of prototypes and higher order transformations is briefly mentioned though not elaborated in (Van Gorp08a) as an improvement of the higher order transformation that supports an SDM extension for specifying *copy* operations more concisely.

By extensively using prototypes, pattern matching with LHS and RHS as in AToM³ (see also Section 2.7) can be mimicked as follows. All LHS's and RHS's are prototypes (in other words, rules are implemented as prototypes). The transformation model implements the behavior of pattern matching in AToM³: try to match the one prototype (which represents the LHS) and then apply another prototype (which represents the RHS). There has to be a form of traceability between elements of the LHS and RHS, which can be easily achieved by naming. The transformation model also implements the scheduling of the rules². Using a LHS and a RHS is, in terms of readability, one of the strengths of AToM³ (and graph transformation in general), which come down to the same advantages of using prototypes.

4.4 A higher order transformation and prototypes as implementation of ND-SDM

This section provides an implementation of ND-SDM in MoTMoT. The instances of the new language constructs (*motmot.transprimitiveND* and *«nextPriority»*) are converted to equivalents written in the Core profile of the SDM language, a strategy suggested in the

²In theory, the scheduling of rules can also be implemented in a prototype. This means that the transformation model is generic and can perform any transformation. In other words, the transformation model is now a transformation tool itself, and the according transformation language consists of certain conventions in the prototypes. This is interesting, because it proves that Story Diagrams powerful enough to model a transformation tool, with a transformation language. This illustrates the concept of *getting rid of the layers* (Kleppe03) (see also Section 2.2).

conclusion of Chapter 3. This conversion is done by means of a higher order transformation. This methodology can be called "preprocessing", "desugaring" or "flattening" a language. In Chapter 3, the equivalents were already presented. However, it turned out that there can be many interpretations for a nondeterministic state, which may vary from modeler to modeler. But they all came down to breaking up the nondeterministic state into a state for each Story Diagram or Story Pattern, and a (in many cases iterated) nondeterministic choice between these states. Because there are many variants which all share several characteristics, prototypes are used as plugins to allow each nondeterministic state to transform to the necessary variant.

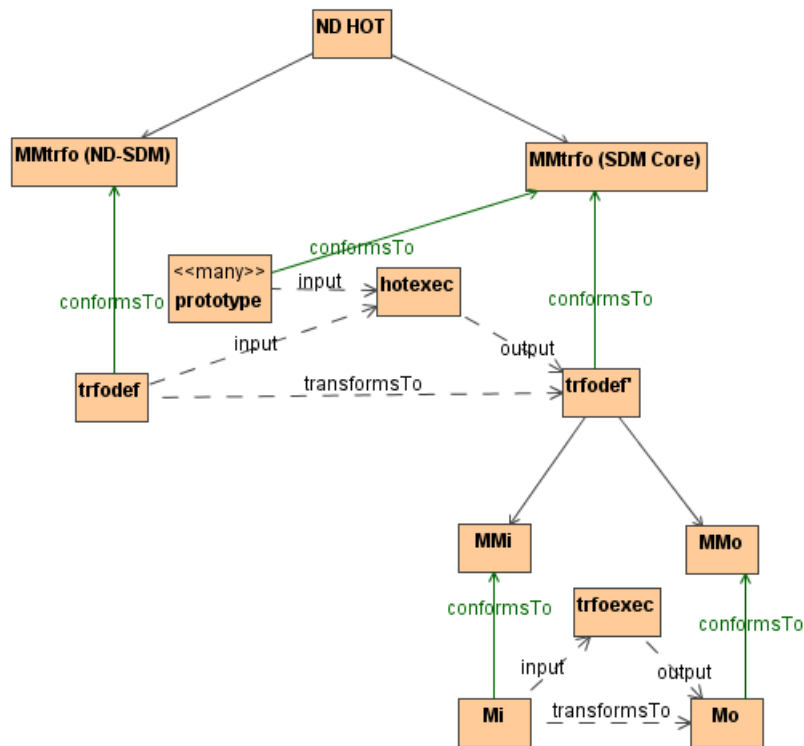


Figure 4.4: The architecture of the ND-SDM implementation

Figure 4.4 shows the architecture of the implementation using a HOT and prototypes. The main idea of the HOT is the transformation of a transformation model `trfodef` conform to the ND-SDM language to a transformation model `trfoedef'` conform to the SDM Core profile. `trfoedef'` can then be executed to transform a model `Mi` conform to a metamodel `MMi` to a model `Mo` conform to a metamodel `MMo`. In the HOT execution, multiple prototypes are used. In this context, `Mi` and `Mo` are called first order models, and `trfodef` and `trfoedef'` are called first order transformation models.

4.4.1 Prototype contract

As concluded in Section 3.1.2, there are too many interpretations to put all in one ND-SDM implementation. Modelers might even design their own variants. Therefore, it is the aim of this implementation that it allows conversion of the nondeterministic state to any variant as a prototype. It must be possible that more than one variant is used on the nondeterministic states of the same model. As suggested in Chapter 3, stereotypes or tagged values can be used on the nondeterministic state to denote the difference between variants.

For example, the nondeterministic state of the household problem of Section 3.1.2 was annotated with a `<<loop>>` stereotype, while the nondeterministic state of the restaurant problem was not. In this case, the presence of the `<<loop>>` stereotype can be called the *application condition* of the variant used in the household problem. Of course, the presence of a `motmot.transprimitiveND` tag is also a requisite for a nondeterministic state. If there is a `<<loop>>` stereotype on a nondeterministic state, apply variant 1, else apply variant 2. Of course, this particular application condition is designed for usage with these particular variants. Usually the application condition is chosen in a way that all the available variants can be distinguished intuitively. In other words, each variant has its own application condition. Therefore, a prototype model must contain a representation of its application condition. This can be done in a simple way, by adding a dummy state annotated with the concerning stereotypes and/or tagged values. Then, if the annotations of a nondeterministic state match those of the dummy state, the prototype in question must be applied.

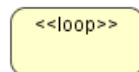


Figure 4.5: The application condition of the variant of the household problem

The application condition of the prototype of the household problem, is given in Figure 4.5, and the actual prototype Story Diagram, is given in Figure 4.6. Together, the diagrams represent the full prototype model.

As said, the application condition shown in Figure 4.5 is simply a nameless state with a `<<loop>>` stereotype. This means that for all nondeterministic states with a `<<loop>>` stereotype this variant will be used.

The actual prototype Story Diagram is given in Figure 4.6. As documentation, the code script is attached as a note to each `<<code>>` state. A `<<code>>` state is a state containing

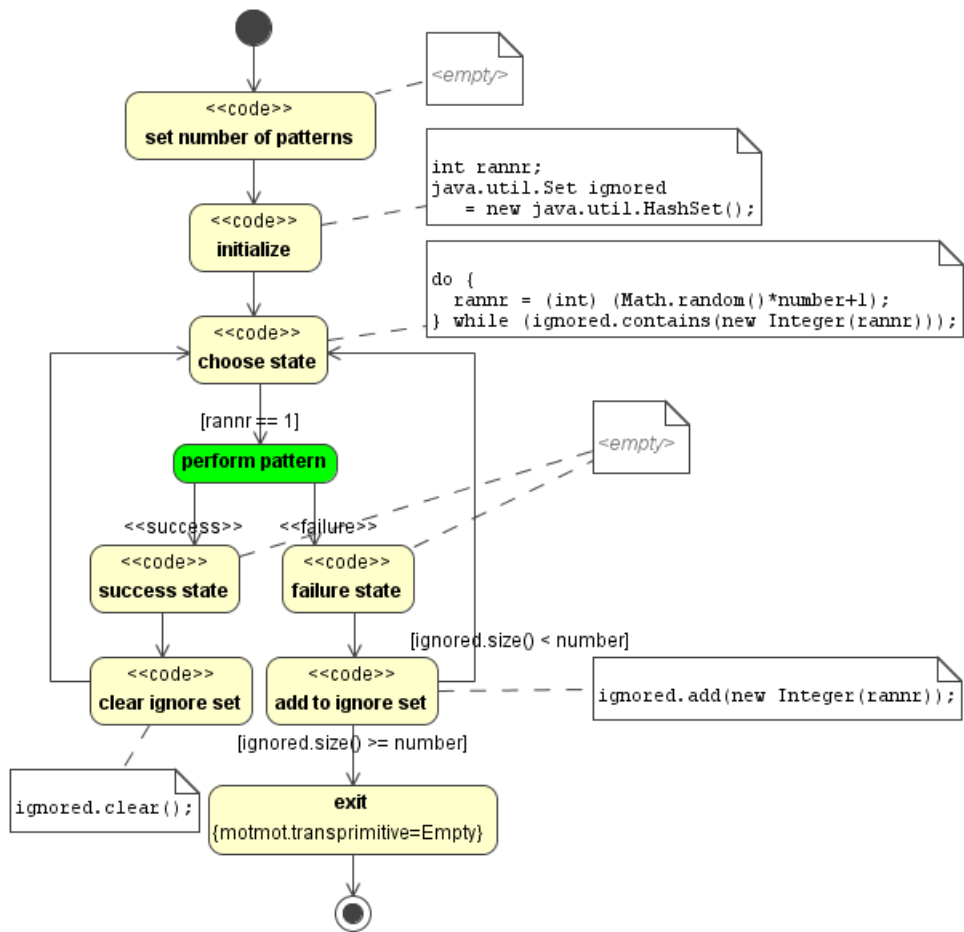


Figure 4.6: The actual prototype of the variant of the household problem

some Java code instead of referencing a Story Pattern. As said in Chapter 3, this variant means *keep matching rules until all failed to match*. Note the similarities with the equivalent of the nondeterministic state in the bottom diagram of Figure 3.6. The details of this prototype and prototypes in general are explained below.



Figure 4.7: The application condition of the variant of the restaurant problem

The prototype model suited for the variant of the restaurant problem is presented in Figure 4.7 (the application condition) and Figure 4.8 (the actual prototype Story Diagram). As said, this variant has the meaning *match each rule once until one fails to match*. In this case, the return value of the nondeterministic state is important, as in the example of Figure 3.7 the nondeterministic state has an outgoing *«success»* transition and an outgoing *«failure»* transition. When all rules are executed, the nondeterministic state returns *true*, but if a rule fails, it immediately returns *false*.

The application condition shown in Figure 4.7 is empty. This means that this prototype is used for states without the *«loop»* state. The actual prototype Story Diagram shown in Figure 4.8 uses a variable `is_success` declared in the "initialize" state to define the return value. This *boolean* remains *true*, but is set to *false* in the state "flag unsuccessful" if a pattern did not match. At the end, the "exit" state with an empty pattern (which always matches) is evaluated according to its *motmot.constraint is_success*. The transition that is followed from this state will secure the return value of the prototype.

In order to allow modelers to build their own prototypes besides those that are presented above, certain conventions have to be established:

- a prototype is a separate XMI file (OMG05) (with extension `.xml` or `.xmi`) in a `prototypes/` subdirectory of the working directory. When executing the HOT, this file will be read, and loaded in the repository, which contains all current models. The file names impose the order on the prototypes: the application condition of `1.xmi` will be evaluated before the application condition of `2.xmi`, etc. In the case of the variants of the household and restaurant problem, the prototype for the household problem has to be named `1.xmi` and the prototype for the restaurant problem has to be named `2.xmi`, in order for the application condition of the household problem (Figure 4.5) to be evaluated first. Otherwise, all nondeterministic states would be transformed using the prototype of the restaurant problem, because they would all match its empty application condition of Figure 4.7;

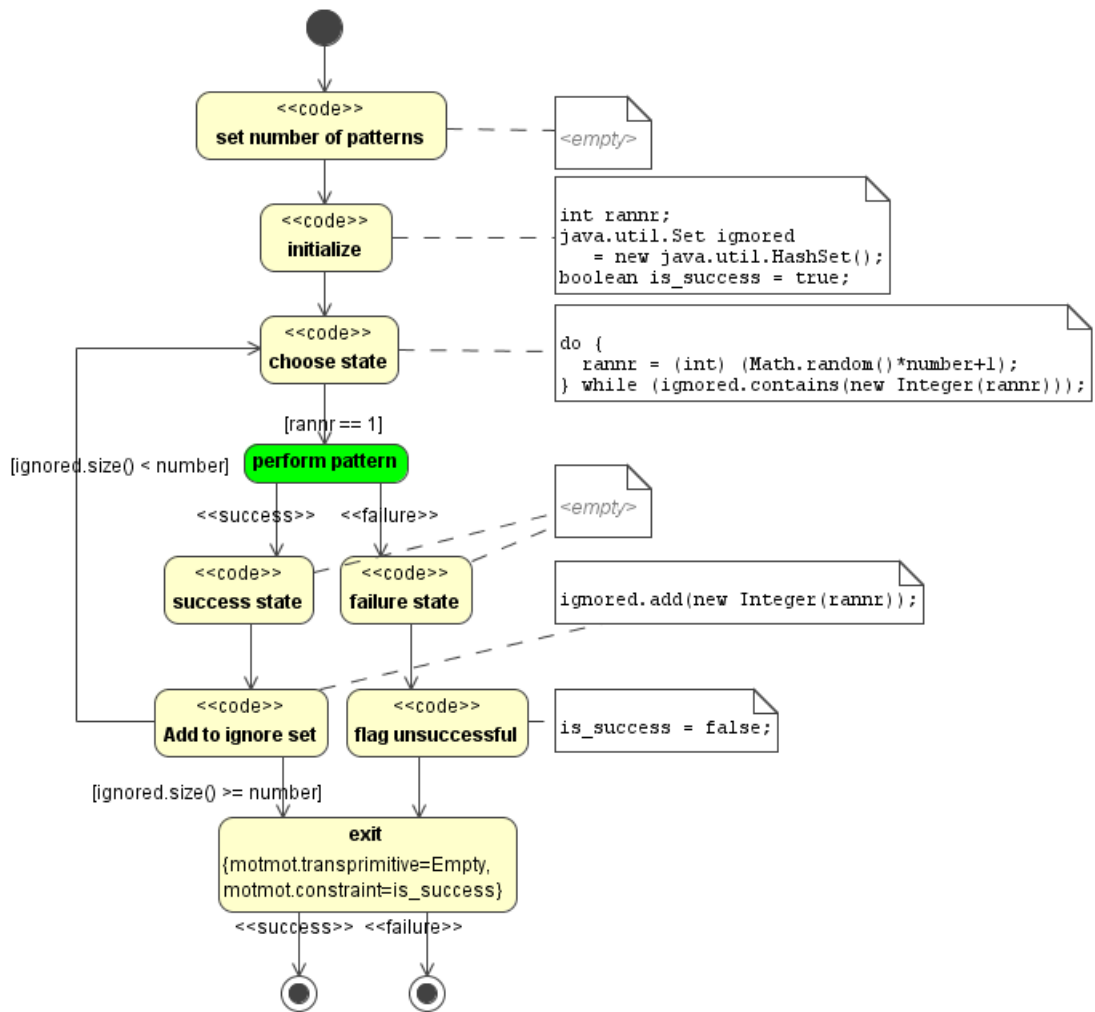


Figure 4.8: The actual prototype of the variant of the restaurant problem

- a prototype contains an Activity Diagram called "AC", containing the application condition. This comes down to a single nameless state, annotated with the stereotypes and/or tagged values in question;
- a prototype contains an Activity Diagram called "NondeterministicSDM_Proto", containing the actual prototype Story Diagram. This Activity Diagram has the following conventions:
 - it contains a `<<code>>` state called "set number of patterns". The `<<code>>` stereotype on the state denotes that the state corresponds to (Java) code instead of a transformation rule, denoted by a *motmot.transprimitive* tag. It must have a code script, but it can be left blank. The HOT will eventually put here a declaration in Java code that declares an *int* and assigns the number of rules in the nondeterministic state to it. This state is scheduled as the first state of the prototype. So all incoming transitions of the nondeterministic state will be reassigned by the HOT to incoming transitions of this state, thus preserving possible stereotypes or guards on these transitions;
 - it contains a `<<code>>` state called "initialize". Its code script is not empty, but it contains at least this declaration:

```
int rannr;
```

This *int* will be used at each iteration as the random number that indicates which rule will be evaluated next. Next to this declaration, other declarations must be made when *ignore sets* or *remember sets* are used (see Section 3.1.2). For example, when only an *ignore set* is used, the following declaration must be added to the code script:

```
java.util.Set ignored = new java.util.HashSet();
```

Other sets may be declared, as long as their names contain the string "ignored" or "remembered". Also, when the return value of the variant is important, the following declaration may be added:

```
boolean is_success = true;
```

In this case, *is_success* may be switched at any point, and another state with a tag *motmot.constraint=is_success* may be used later on in the prototype to enforce that the correct (`<<success>>` or `<<failure>>`) transition is followed. When this is done in the last state, it determines the return value. Finally, a counter *iter* may be declared for several purposes:

```
int iter = 0;
```

- it contains a `«code»` state called "choose state". Its code script contains the following:

```
rannr = (int) (Math.random()*number+1);
```

`number` represents the number of rules in the nondeterministic state in question. So this line of code assigns an integer between 1 and `number` to `rannr`. `rannr` represents the rule that is chosen this iteration. If *ignore sets* are used, patterns that are in the *ignore sets* can't be chosen, so the code script must look like this (in the case of one *ignore set*):

```
do {
    rannr = (int) (Math.random()*number+1);
} while (ignored.contains(new Integer(rannr)));
```

As said previously, for each rule of the nondeterministic state a *motmot.transprimitive* state will be created. In the HOT, a transition is created from the "choose state" to each of the *motmot.transprimitive* states, with a guard [`rannr==X`] with `X` a different number between 1 and `number` for each rule. In that way, according to `rannr`, one of the *motmot.transprimitive* states is entered (analogous to what was explained in Chapter 3, in particular in Figures 3.6 and 3.7);

- it contains states called "success state" and "failure state". This state will be the target of a `«success»` respectively `«failure»` transition from each state generated from a rule of the nondeterministic state. This state may be a `«code»` state, containing a code script that performs actions like *ignore rule next time* or *clear ignore set* (see also Figures 3.6 and 3.7), although such actions may be of course implemented in other states and scheduled as wanted;
- it contains a *motmot.transprimitive* state called "exit" pointing to an empty pattern. This state must be scheduled as the last state of the prototype. In that way, outgoing transitions of the nondeterministic state will be reassigned by the HOT as outgoing transitions of the "exit" state, similar to the "set number of patterns" state. This state may contain `{motmot.constraint=is_success}` when using the `is_success` variable described above. In that way, the correct behavior of the outgoing `«success»` and `«failure»` transitions of the nondeterministic state is maintained after having converted the first order transformation model.

Besides these obligations, the modeler is free to include the following in the Activity Diagram called "NondeterministicSDM_proto" (some of which are indeed included in Figure 4.6 and Figure 4.8):

- a state called "perform pattern". This state can be used to visualize the *motmot.transprimitive* states created from the rules of the nondeterministic state. Because it is only used for illustration and clarity, the state and its incoming and outgoing transformations are ignored when copying the prototype. It can however greatly help the modeler to grasp the workings of the prototype;
- states, possibly with SDM stereotypes and SDM tagged values. They will be copied accordingly;
- transitions, possibly with SDM stereotypes and guards. They also will be copied accordingly;
- patterns, if their package resides in the same package of the Activity Diagram called "NondeterministicSDM_proto". Because states with tagged values are also allowed, it is possible to use *motmot.transprimitive* states that refer to patterns in the prototype³;

These prototypes are integrated into the first order transformation model (i.e. the input model of the HOT) by the following HOT tasks:

- copy all states and transitions with the exception of the initial state, the final states, the "perform pattern" state, and all connected transitions;
- rename the names of all newly introduced variables of the copied prototype. This must be done for two reasons. First, suppose that a Story Diagram with two nondeterministic states is transformed. In this case, the nondeterministic states are transformed to their equivalents, resulting in a Story Diagram with states with the same name (for example "initialize"). However, in UML Activity Diagrams, state names must be unique (OMG01). Second, all variable names must be made unique in order to avoid double declarations in the Java code that is generated by MoTMoT. This last reason is however an issue of MoTMoT itself;
- reassign incoming and outgoing transitions of the nondeterministic state to the copy of the prototype. The prototype is now connected to the Story Diagram of the first order transformation model;

³However, because the prototype resides in a separate file, there is no direct access to the metamodel of the input model of the first order transformation. In the architecture of Figure 4.4, the metamodel that is meant here is *MMi*. Without access to the metamodel, no useful patterns can be modeled in the prototype. This makes sense, because otherwise the prototype would contain domain-specific information of a particular transformation. The prototype should be able to be used in any kind of model transformation regardless the model language it transforms. Nevertheless, when supposing that the prototype conventions would allow the prototype to be part of the transformation model, this dependence can be allowed, as in this case, the prototype can be considered transformation-specific. In this thesis however, a prototype is considered a variant of a nondeterministic state and it is part of the transformation language ND-SDM.

- create a *motmot.transprimitive* state for each rule of the nondeterministic state and create the three transitions similar to the incoming and outgoing transitions of the "perform pattern" state in Figures 4.6 and 4.8;
- remove the nondeterministic state.

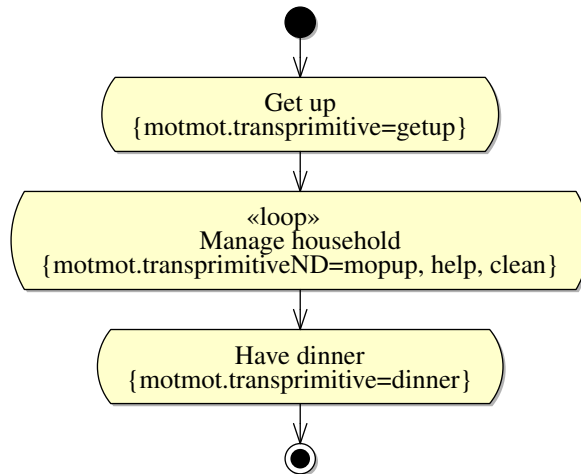


Figure 4.9: A household modeled using hybrid rule scheduling

While a vague equivalent to the nondeterministic state was shown in the bottom diagram Figure 3.6, a detailed equivalent conform to the SDM Core profile (thus usable by MoTMoT and other SDM tools) can now be constructed for each nondeterministic state and integrated into the Story Diagram. An execution of the HOT on the household problem (once again shown in Figure 4.9) results in the Story Diagram shown in Figure 4.10, using the application condition of Figure 4.5 and the prototype of Figure 4.6. Again, the code scripts of the `» states are included as notes for clarification. To conclude, this diagram has the same behavior as the diagram in Figure 4.9 envisioned, but this one can be executed by a tool that supports Story Diagrams.`

4.4.2 Overview of the higher order transformation

In this section an overview of the implementation of the HOT is given. The actions the HOT must perform were already described briefly in the previous section, but it was not yet explained how this is done. As said in Section 4.2, according to the transformation tool, the HOT resembles any other transformation. So it consists of Story Diagrams and Story Patterns.

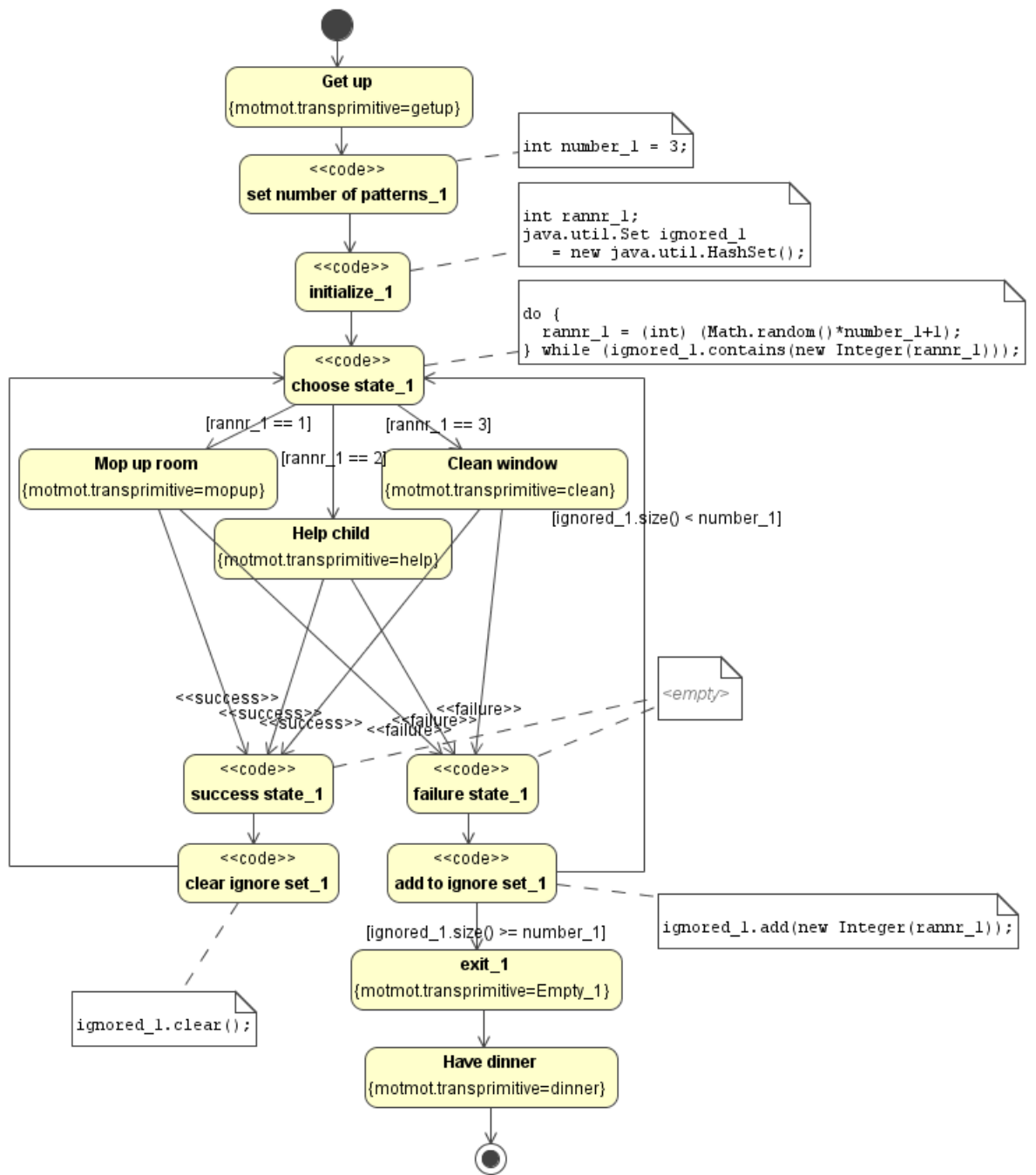


Figure 4.10: A household after execution of the HOT

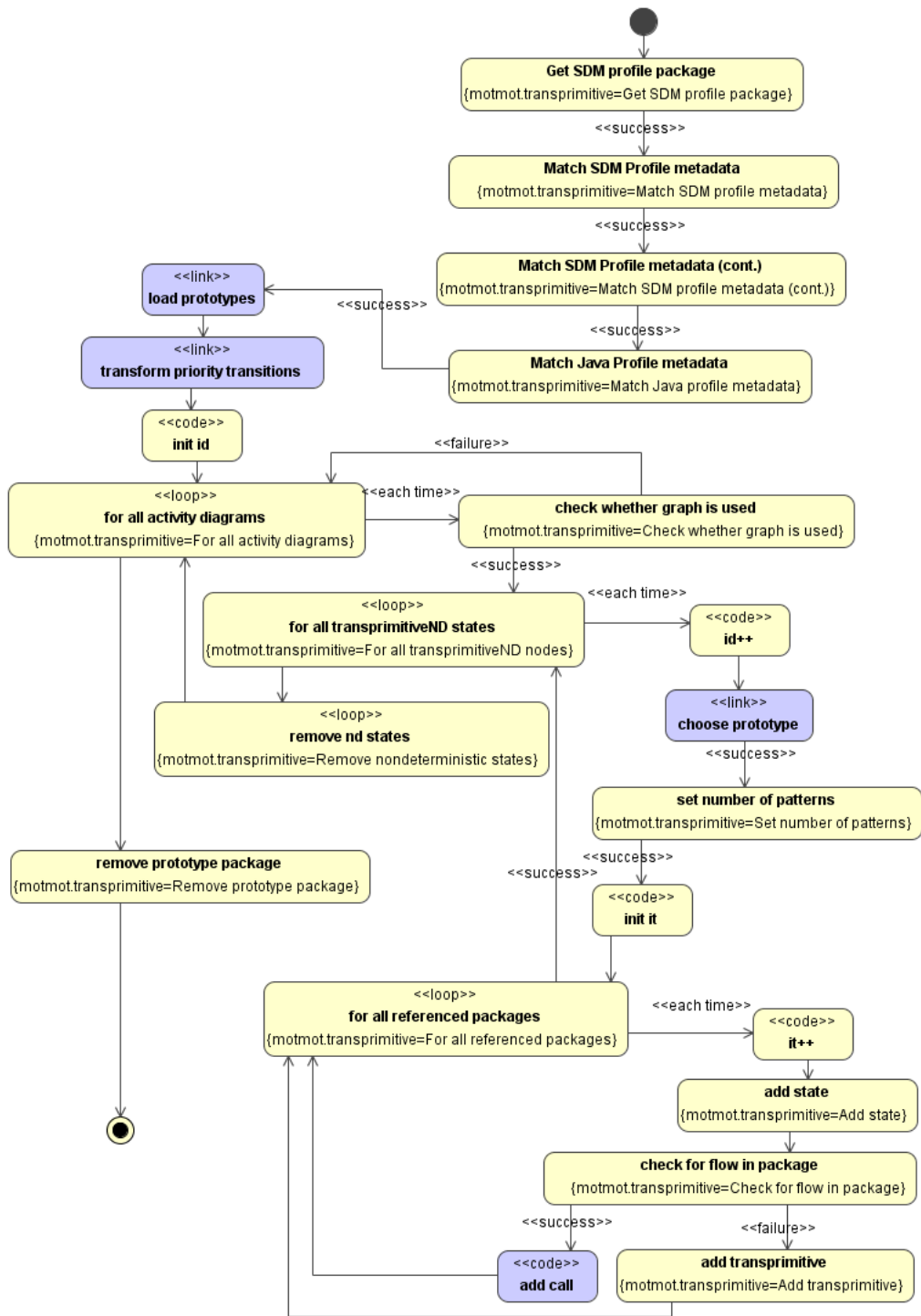


Figure 4.11: An overview of the ND-SDM implementation HOT

The main Story Diagram of Figure 4.11 gives an overview of the HOT. First, the package containing the SDM Core profile and ND-SDM extensions (i.e. the model of the ND-SDM language) is looked up by name in the transformation model. Also, a new package named `protoContainer` is created that will contain all the application conditions of the prototypes that are available. The pattern of this state is given in the Appendix in Figure A.7.

Then, data of the metamodel (i.e. the SDM model) is matched in the first three `Match` metadata states. Language constructs such as the `<<success>>` stereotype (part of the SDM Core profile) and the `motmot.transprimitivEND` Tag Definition (part of the ND-SDM extensions) are looked up and bound in these patterns. Also, some Java language constructs, such as the `boolean` type that are also needed are matched. The necessity of Java is discussed in the section about tool-independence (Section 4.4.7). The patterns are given in Figures A.8, A.9 and A.10.

All previous patterns must not fail, because the elements that are bound here are needed later on in the HOT. Failing one of these patterns causes the HOT to stop. This is not visualized in the simplified Story Diagram of Figure 4.11.

4.4.3 Prototype loading

In the next state, `load prototypes`, another Story Diagram is called, given in Figure 4.12. Its purpose is to read the application condition of each available prototype. Each application condition is then registered by copying it in the package `protoContainer`, created previously (see Figure A.7).

After having finished this Story Diagram, all application conditions are loaded. These application conditions can be consulted when a nondeterministic state is transformed, in order to choose the correct prototype.

Prototypes are stored in separate files. First, all files containing prototypes are listed in `list possible prototypes`. Then, a counter is initialized and increased. The state `For each found file` references an empty Story Pattern, so it iterates while the counter is smaller than `nFiles`, the number of available prototype files. For each file, a container is added. Then, the file is read and the application condition is loaded as an Activity Diagram. Next, the only state in this Activity Diagram, representing the application condition (for example, see Figures 4.5 and 4.7), is bound. The application condition consists of the present stereotypes and tag definitions on this state, so these are put into the container, created in `add container`. The patterns and code blocks are given in Figures A.11 to A.17.

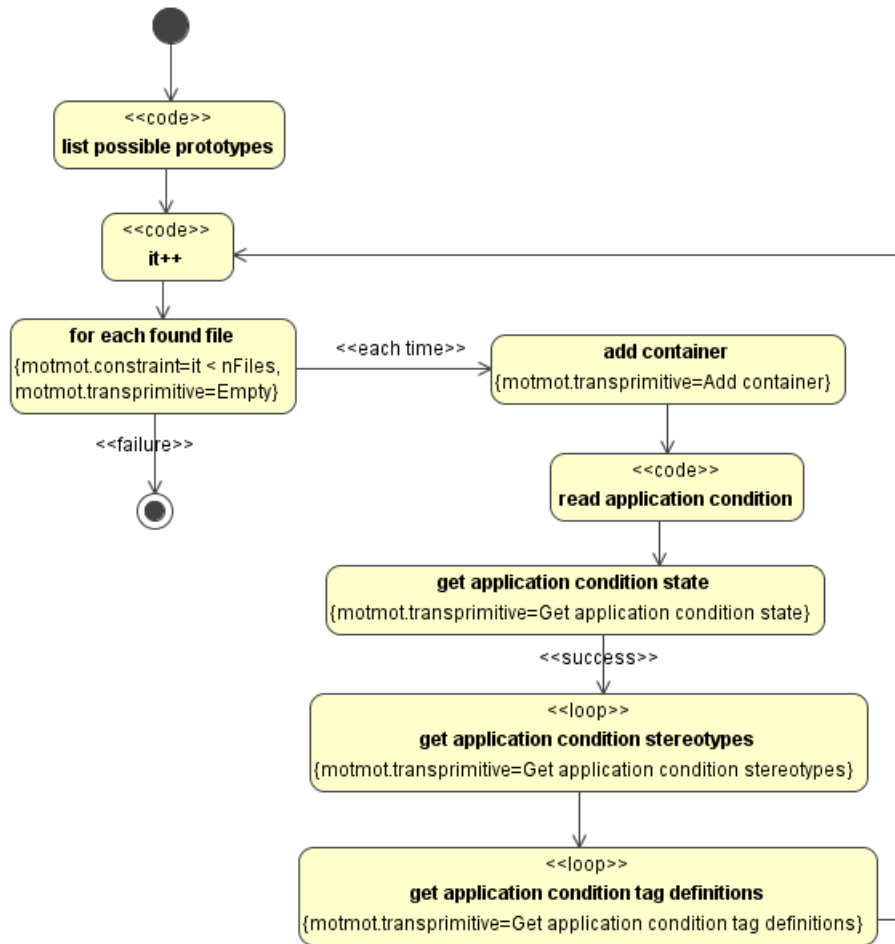


Figure 4.12: Loading the application conditions of the prototypes

4.4.4 Priority transitions transformation

After the prototypes are loaded, the transform priority transitions state of the main Story Diagram in Figure 4.11 is entered. It calls another Story Diagram, given in Figure 4.13.

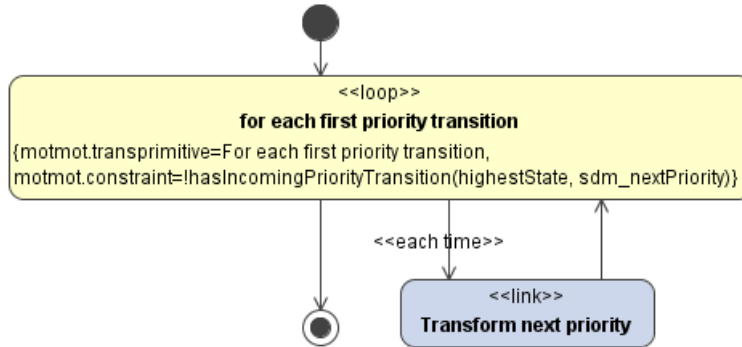


Figure 4.13: Transforming the *<<nextPriority>>* transitions

This Story Diagram will transform a *<<nextPriority>>* transition chain (i.e. successive states connected with *<<nextPriority>>* transitions) to its equivalent, as explained in Section 3.3. The first state of Figure 4.13 iterates over every *<<nextPriority>>* transition chain. Its pattern matches a state *highestState* with an outgoing *<<nextPriority>>* transition, but without an incoming *<<nextPriority>>* transition. In other words, a state is found that has top priority in the priority chain. From this state, a *<<success>>* transition is created from and to *highestState*, conform to Figure 3.9. The pattern is shown in Figure A.18.

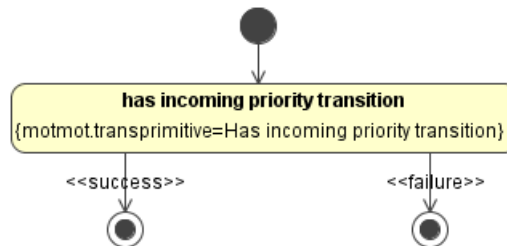


Figure 4.14: The Story Diagram that checks for incoming priority transitions

As said, the pattern may only match if *highestState* has no incoming *<<nextPriority>>* transitions. "Not having an incoming *<<nextPriority>>* transition" can not be expressed in a story pattern. Therefore, there is an extra constraint on the "for each first priority

transition" state. It says that the method `hasIncomingPriorityTransition` must fail for `highestState`. This method calls the Story Diagram given in Figure 4.14. Its pattern, shown in Figure A.19, matches if `highestState` *does* have an incoming `<<nextPriority>>` transition. If so, the `<<success>>` transition is followed to the final state, which causes the method `hasIncomingPriorityTransition` to return `true`. As a consequence, the constraint fails and thus the pattern `For each first priority transition` fails for this particular binding of `highestState`. If on the other hand, no incoming `<<nextPriority>>` transition is found in `has incoming priority transition`, the method returns `false` and the pattern `For each first priority transition` can match. Actually, this is an implementation of a *negative application condition* (see Chapter 6), as evaluation can only succeed if a sub-pattern *does not* match.

For each `<<nextPriority>>` transition chain, the Transform `next priority` state in Figure 4.13 is entered, which calls a Story Diagram given in Figure 4.15. The first thing that stands out is that this graph calls itself in the Transform `next priority` state. This is because the graph represents a recursive loop. Recursion is a good example of the expressive power of explicit rule scheduling in combination with the ability to call other Story Diagrams from states.

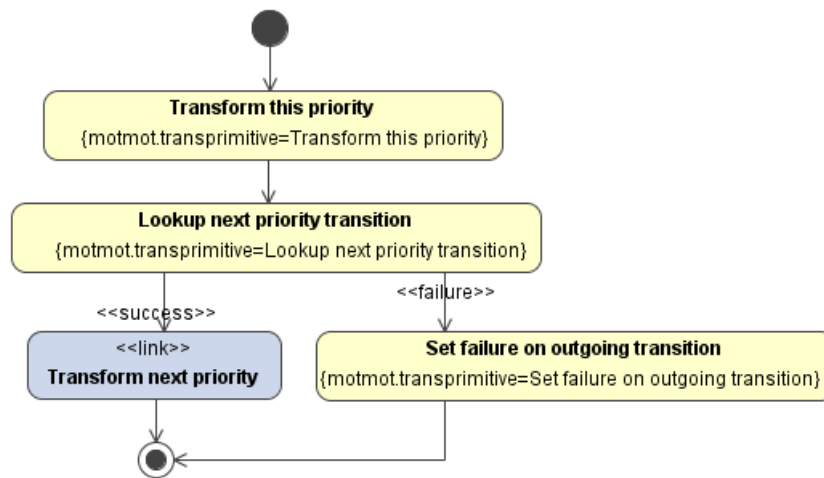


Figure 4.15: The Story Diagram that transforms the next priority

In Figure 4.15, the Transform `this priority` state is entered first. This state performs the actual transformation of one transition to a `<<success>>` and `<<failure>>`, according to Figure 3.9. The pattern is given in Figure A.20.

In the next state, the following `<<nextPriority>>` transition of the chain is bound, if there still is one. If successful, the Story Diagram is recursively called in the `Transform next priority` state, but with the new transition as parameter. As a consequence, all `<<nextPriority>>` transitions are transformed one by one until the end of the chain is reached. Then, the `Lookup next priority` transition state fails and a `<<failure>>` stereotype is put on the outgoing transition of the `<<nextPriority>>` transition chain, as shown in Figure A.22. After each `<<nextPriority>>` transition chain has been transformed, the loop in `for each first priority` transition in Figure 4.13 ends and the next state in Figure 4.11 is entered.

4.4.5 Prototype choice

After having transformed the `<<nextPriority>>` transitions, the code of state `init id` initializes a variable `id`. This variable will provide a unique ID for each nondeterministic state that is transformed. The value of this ID can be added to the names of states that are added during the transformation, to ensure that there are no states with the same name in one Activity Diagram. After all, as said previously, this is illegal in UML Activity Diagrams (OMG01). The code of this state is given in Figure A.23.

Next, in the `for all activity diagrams` (see Figure A.24), a loop is started that binds each Story Diagram one by one. For each diagram, it is tested whether the graph is actually used as Story Diagram in the transformation model. If not, the next Activity Diagram is looked up. The pattern is given in Figure A.25.

Then, an iteration starts over each nondeterministic state in the current Story Diagram (see Figure A.26). For each nondeterministic state, the ID is incremented (see Figure A.27). In the next state, `choose prototype`, the correct prototype must be chosen for this particular nondeterministic state.

The state calls the Story Diagram given in Figure 4.16. Each application condition that has been loaded in the Story Diagram of Figure 4.12 must be inspected. Note that, as previously said, the order in which the application conditions are checked is important when the stereotypes and tag definitions of one application condition are a subset of those of another application condition. In MoTMoT, the order is kept from reading the prototypes, so no extra ordering is needed⁴.

First, a boolean `found` is initialized to `false` in the `init found` state. Next, an iteration is started over each prototype. For each prototype, a boolean `nomatch` is initialized

⁴To have a fixed order in matching the application conditions, the way of matching patterns must be influenced. This would require to get round the usual pattern matching algorithm of Story Driven Modeling and Graph Grammars in general. This concept may also be useful in other contexts.

to *false*. Then, all stereotypes of the prototype are matched iteratively in the `for each stereotype` state. For each matched stereotype, the nondeterministic state is checked for a stereotype with the same name in the `match stereotype` state. If a stereotype was not matched at some point, this means that the current prototype does not match. `nomatch` is then flagged *true* in the `stereotype does not match` state, causing the `«loop»` state for each `stereotype` to fail for the next iteration. The `stereotype matched?` state is entered and returns *false*, so the `«failure»` transition is followed to the `for each prototype` state, where a new prototype is matched. If this time, all stereotypes match, the `stereotype matched?` state returns *true* and all tag definitions are evaluated in a similar way. If all stereotypes and tag definitions match, the matching prototype is found and can be created (see below). The boolean `found` is flagged *true*, in order to stop the iteration in the `for each prototype` state in the next visit⁵. The patterns and code blocks are given in Figures A.28 to A.37.

If all stereotypes and tag definitions of a nondeterministic state match those of a certain application condition, its prototype will be used. In the `create prototype` state, the prototype is read and integrated into the Story Diagram of the nondeterministic state. The state calls the Story Diagram of Figure 4.17. This Story Diagram mainly consists of five phases. In these phases, whenever necessary, variable names for states etc. are made unique, for the reason stated above.

1. the prototype is read;
2. the necessary states, transitions and patterns (conforming to the prototype contract of Section 4.4.1) are moved to the given Story Diagram;
3. the incoming and outgoing transitions of the nondeterministic state are reassigned to the first and last state of the prototype in order to integrate it into the Story Diagram;
4. annotations such as tag definitions and stereotypes are reassigned by name to those of the metamodel of the transformation model;
5. the now empty Story Diagram of the prototype is removed from the working repository.

After having finished the Story Diagram of Figure 4.16, the currently bound Story Diagram has been enriched with the prototype. The prototype conforms to the SDM metamodel and is connected to the other states in the Story Diagram.

⁵The boolean variables have to be used, as MoTMoT forbids a transition from for example the `match stereotype` state to the `for each prototype` state. After an `«each time»` transition, control must eventually end up in the `«loop»` state in order to properly exit the iteration.

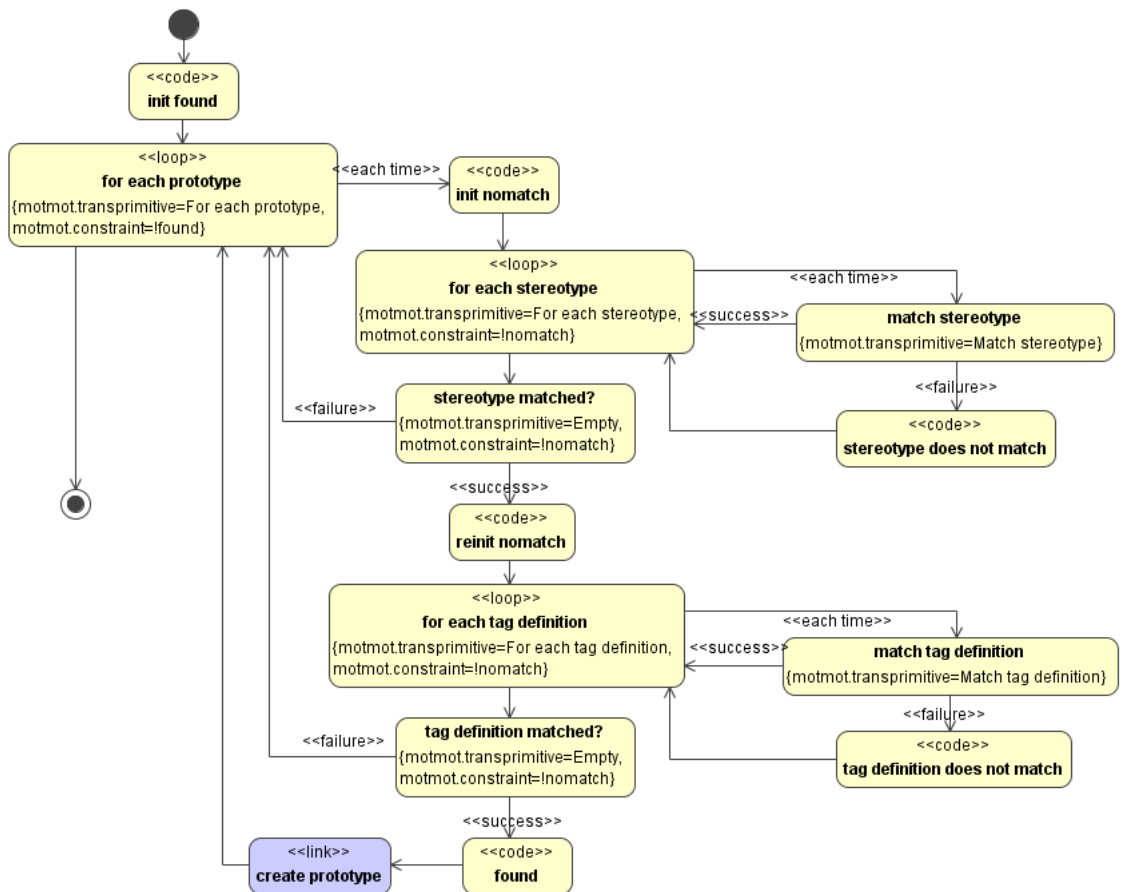


Figure 4.16: Choosing the right variant according to the application conditions



Figure 4.17: Reading and integrating the chosen variant

4.4.6 Nondeterministic state transformation

In the next few states of Figure 4.11, the added prototype is altered according to the Story Patterns and Story Diagrams that are referenced by the nondeterministic state. The integrated prototype will be shaped in such a way that the result conforms to the prototype contract (see Section 4.4.1), like Figure 4.10 for example. First, in the `set number of patterns` state, a declaration for the actual number of referenced packages is added in the code script of the freshly added `<<code>>` state, which was also named `set number of patterns` (see the section about the prototype contract, Section 4.4.1). The pattern is shown in Figure A.38. Then, a variable named `it` is initialized, which will be used as a counter, as shown in Figure A.39.

Next, each pattern is looked up one by one in the `for all referenced packages` state (see Figure A.40). For each package, `it` is increased (see Figure A.41). Then, in the `addstate` state, a state is added which will reference the current package, either with a `motmot.transprimitive` tag or with a method call. The incoming and outgoing transitions are also generated. The pattern is shown in Figure A.42.

If the referenced package contains a Story Diagram, a method call is added to the new state. Otherwise, a `motmot.transprimitive` tag is generated. The test for an Activity Diagram is done in the `check for flow in package` state, with its pattern shown in Figure A.43. The pattern where a `motmot.transprimitive` tag is created, is shown in Figure A.44. The other case, where a method call is added, is a bit more complicated and needs its own Story Diagram, shown in Figure 4.18.

If a method call to a Story Diagram must be added to the new state, the method itself must also be created. In the first state of Figure 4.18, the method and the call are created (see Figure A.45). The Story Diagram that will be called may contain some patterns with some `<<bound>>` elements. These bound elements must be added as parameters in order to make sure that they can be used in the called Story Diagram. This is done in the next state by iteratively executing the pattern in Figure A.46. After all parameters have been generated, the parameter list of the method call is closed in `close param list` (see Figure A.47).

Again in the main Story Diagram of Figure 4.11, all referenced packages are taken care of like described above, for each found nondeterministic state in the currently bound Story Diagram. The only thing left to do is remove the nondeterministic state itself, as it is now transformed to an equivalent. This is done in the `remove nd states` state, with its pattern

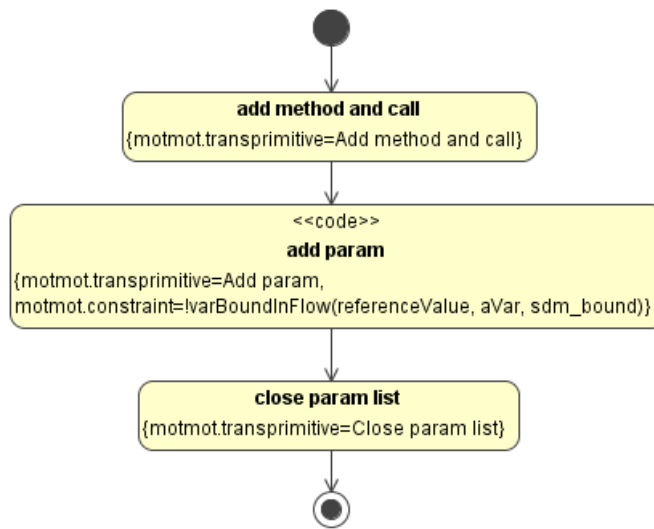


Figure 4.18: Adding a method and a call to this method

given in Figure A.48⁶.

After all Activity Diagrams in the model have been treated, the model is cleaned up by removing the package containing the application conditions in the `remove prototype` package state (see Figure A.49). Then, the higher order transformation is finished. Now, the transformation model is usable by the SDM tool because all ND-SDM constructs are transformed to their equivalents.

It can be noted that many of the states in the higher order transformation can be scheduled themselves using nondeterministic scheduling. For example, when loading prototype application conditions in Figure 4.12, the copying of stereotypes and tag definitions can be modeled in one state, as it is conceptually the task *get all state annotations*. Also in many other cases, for example when reading the variant in Figure 4.17, using nondeterministic states would make the Story Diagram more concise and readable.

4.4.7 Tool-independence

This higher order transformation is modeled as a Story Diagram with Story Patterns, just like the input- and output model it reads and generates. This is done in order to minimize

⁶The nondeterministic states are removed all together outside the loop of the `for all transprimitiveND states` state. If this was done inside the loop, the Java iterator generated by MoTMoT that is generated for the `for all transprimitiveND states` state will get erroneous after deleting the nondeterministic state. This is because the code for deleting the state is `stateContainer.remove(state)` instead of `iterator.remove()`.

the need for proprietary infrastructure. The applicability of higher order transformations for language extension was discussed by Olaf Muliawan (Muliawan08).

This approach can be realized on any tool for SDM, like MoTMoT, Fujaba 3, Fujaba 4, etc. Moreover, the same concepts can be used on other tools and frameworks that are not related to SDM. Combining a very limited set of language constructs as proposed in (Van Gorp08b), and extending the core language by using higher order transformations, can overcome interchangeability problems between transformation tools.

While the fundamental idea of this implementation makes it tool-independent, there are still some minor problems with respect to tool-independence. Indeed, the prototype has many `<<code>>` states containing Java code. So, apart from the UML, this implementation is also dependent of Java. In other words, the transformation language of the higher order transformation consists of both the UML and Java.

A first idea to avoid the dependency of Java is to avoid Java code at all. Following Figure 4.10, this would mean that for example the `ignored_1` set would become a UML *Package*. Elements representing the nondeterministically scheduled Story Patterns or Story Diagrams can then be added and removed from the package when they must be ignored next time or not. However, this idea comes with a serious problem. The package contents must be able to change during execution of the first order transformation. So in order to be able to use the package in an execution of the transformation, it can not be modeled as part of the transformation model. It must be added at some place in the first order model, as this is the transformation's working space. Consequently, it must be either conform to the metamodel of this first order model to blend in, or it can have its own metamodel, which must also be added to the first order model. However, changing the first order by the higher order transformation is conceptually impossible, for the same reason as explained in the footnote on page 54. The *model* entity of the first order input model has to be looked up, which requires knowledge of the input model's metamodel. It is very important that this metamodel is not visible to the HOT, because the HOT must be independent of the kind of transformation it takes as input.

The HOT must be independent of Java code, without needing access to the first order input model. This can be done by using some kind of profile for generic containers. This profile would be an interface for constructs such as a set and a node. The modeler can use this interface in the same way as the UML. An according mapping to source code, in the case of MoTMoT, Java code, can then be implemented. In that way, the first order input model does not have to be touched while still obfuscating the fact that MoTMoT generates

Java code. Other useful constructs and operations can be added to this profile, like reading a model from a file or URL (this is different for each transformation tool, because it depends on the transformation language), constructs for boolean expressions, integers, the size of a collection, strings and string manipulation. These are all constructs and operations that are now used as plain Java code in the higher order transformation. Also, constructs for traceability between input and output models can be added to this profile. A disadvantage of this approach is that the tool itself has to be extended with this profile. However, the SDM profile (for transformation concepts) and this profile (for utilities and simple operations) form a mature, versatile, usable and extendable core for a transformation language. On top of that, it is also a stimulus for standardization of these language concepts.

When using the generic containers, the architecture of the higher order transformation, previously given in Figure 4.4, would look like Figure 4.19. The metamodels of the transformation models (i.e. the SDM Core profile and ND-SDM) are enriched with the structures and operations provided by the generic container. Since in this thesis MoTMoT would be used as transformation tool, a Java implementation is used.

4.5 Conclusion of this chapter

Nondeterministic scheduling is implemented in SDM as a higher order transformation, which transforms nondeterministic constructs to their equivalents in the SDM Core profile. By using a higher order transformation, the transformation language is extended without extending the transformation tool. This means that this approach can be used on any tool supporting the concept of higher order transformations and SDM. In addition, the ability to add and remove variants for nondeterministic scheduling is also achieved by using prototypes. With prototypes, the higher order transformation does not have to be altered when adding or removing a new variant. Moreover, when following the prototype contract, these prototypes can easily be built by users of ND-SDM, as it is in the same conceptual layer as the transformation models they build. Besides this, users can schedule rules implicitly in an explicitly scheduled diagram, but also the other way around. This is because it is possible to reference a whole Story Diagram in a nondeterministic state. Therefore, the language ND-SDM can be truly called hybrid with respect to rule scheduling. Summarized, the technique of higher order transformations and prototypes can be used to extend the transformation language further and further in an elegant and tool-independent way.

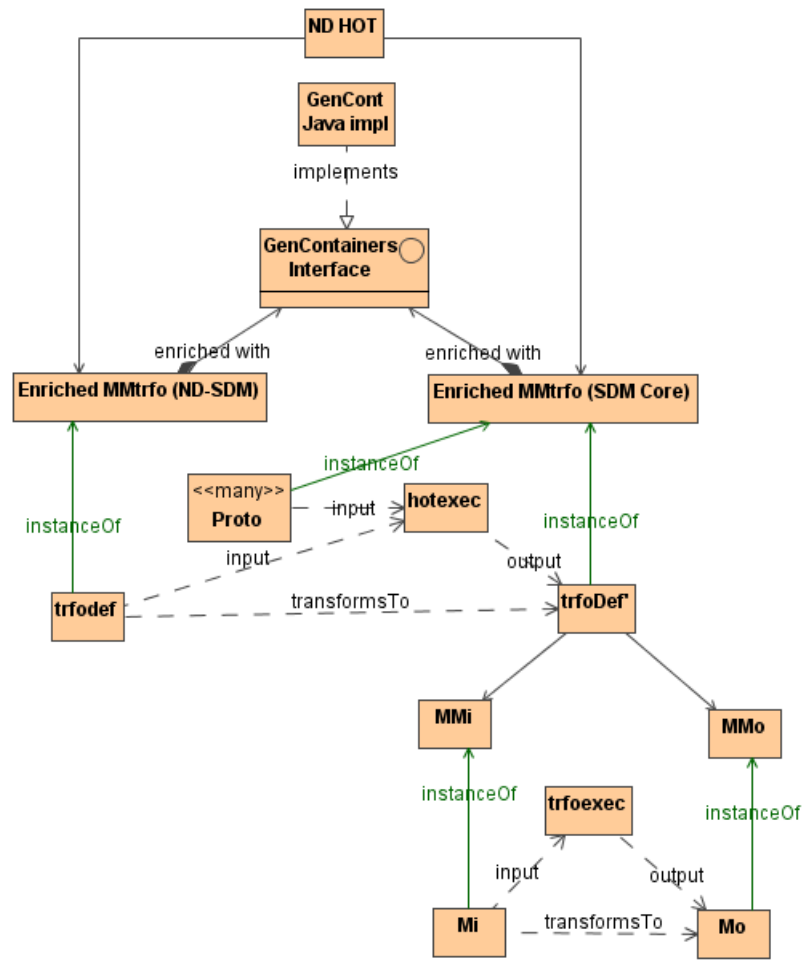


Figure 4.19: The architecture of the ND-SDM implementation using generic containers

Case studies in the usage of ND-SDM

This chapter contains three case studies of example transformations that use the featured language constructs for nondeterministic scheduling of ND-SDM. The first two example transformations start with an implementation either in a language that supports only explicit rule scheduling, such as MoTMoT, or in a language that supports only implicit rule scheduling, such as AToM³. The weaknesses of these implementations are discussed, and a better alternative in ND-SDM is implemented. The third example illustrates the usage of another nondeterministic variant and priorities.

5.1 The UML to RDB transformation

5.1.1 UML to RDB

For model transformations, the *Conceptual Data Model to Relational Data Model* is a common case study (for instance in (Bézivin05)). In this case, we will use a variant of UML Class Diagrams as Conceptual Data Model. This is particularly interesting in the context of this thesis, as UML Class Diagrams are also part of the transformation language. In fact, it illustrates a property of MDE: metalanguages such as UML Class Diagrams can also be used as first order model languages.

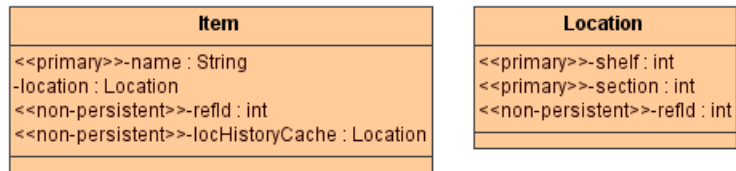


Figure 5.1: A storehouse inventory system in Class Diagrams

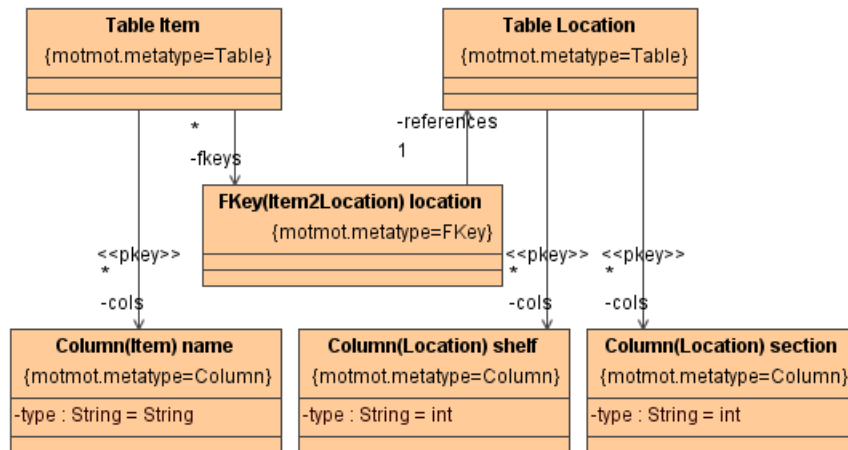


Figure 5.2: A storehouse inventory system in a UML Relational Data Model

Figures 5.1 and 5.2 represent an example of the execution of such a transformation. Because the modeling tool that is used in this thesis, Magic Draw, supports the UML, the metamodel of both diagrams is the UML. However, certain stereotypes and tagged values are used as annotations to describe domain-specific language features. The UML metamodel is shown in the appendix, in Figures A.1 to A.6.

In this respect, in the Class Diagrams, attributes can have a `<<primary>>` or a `<<non-persistent>>` stereotype. `<<primary>>` means that this attribute represents the unique identifier for this class. A `<<non-persistent>>` attribute is an attribute for which its value does not have to be stored permanently. Methods and associations are omitted in this simplified example.

The metamodel of Relational Database Models is a bit more complicated. Tables, columns and foreign keys are modeled as classes, and their type is defined by the value of the `motmot.metatype` tag. A table can have many columns and foreign keys, modeled by an association. A foreign key references one table, denoting the type of the foreign key, also by means of an association. A column is a primary key of a table if there is a `<<pkey>>` stereotype on the association between them.

All classes from the Class Diagram of Figure 5.1 become tables in the Relational Database Scheme of Figure 5.2. Attributes are transformed to either columns or foreign keys, depending on whether the type of the attribute is a basic datatype such as `int` or `String` or an instance of a class that is part of the model such as `Location`.

5.1.2 Implementation in MoTMoT

Figure 5.3 represents a simplified model transformation that transforms a Class Diagram to a Relational Database Scheme. Rules are scheduled using SDM Story Diagrams, the original transformation language of MoTMoT. The first rule, `Match metadata`, looks up the metamodel of the Class Diagrams and the Relation Database Schemes. Then, the input model is checked on the presence of tables, as this would mean that the transformation already has been executed. If a table can be found, the `<<success>>` transition is followed and the algorithm ends. If not, the actual transformation of class diagrams is started, and first, each class is transformed to a table in the `<<loop>>` state `Transform classes to tables`.

Next, class attributes must be transformed. As explained before, it turns out that there are two kinds of attributes that need two separate transformation rules. More detailed, attributes can be of a simple data type or of an object type, which is represented differently

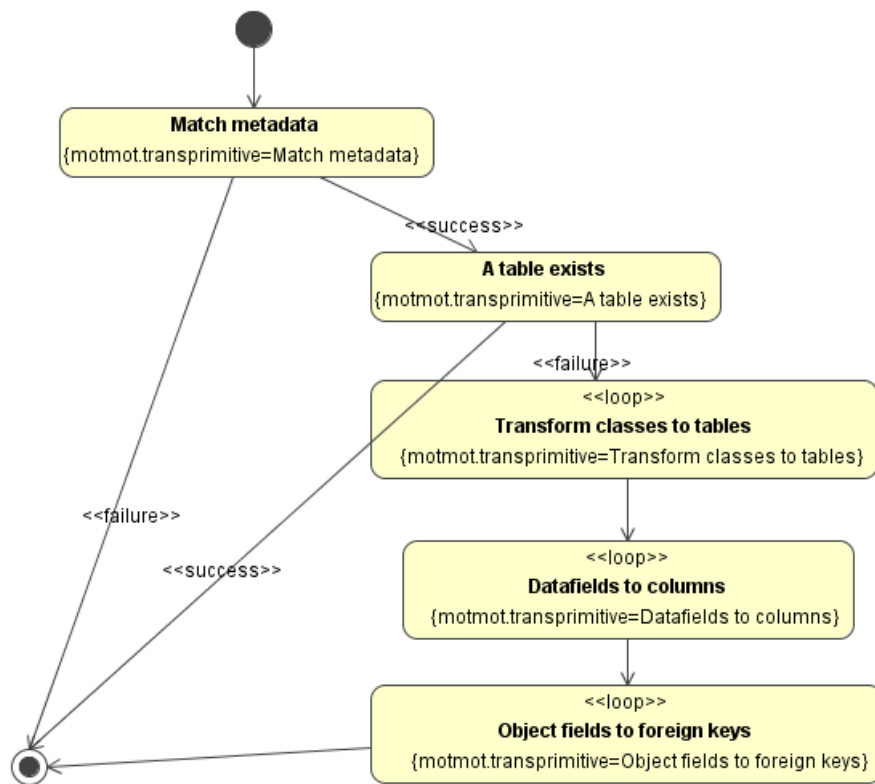


Figure 5.3: A transformation from Class Diagrams to a Relational Database Scheme

in databases. Attributes of a simple data type t become columns of type t . Object attributes of class T become foreign keys with references to table T , as is the case with the `location` attribute of the Class Diagram of Figure 5.1. In short, in Figure 5.3, all datafields are transformed, and then all object fields are transformed.

Having to express the transformation of attributes in two different rules decreases the quality of the transformation model in several ways. First, one has to impose an order on these two rules, which has no meaning. This is thus a clear case of overspecification. Second, modeling the transformation of all attributes as two different rules stresses the need for two separate transformation rules. This should not be stressed in the Story Diagram, as the Story Diagram is of a different level of granularity. Third, the transformation of all attributes is conceptually one action, and should be modeled as such in the Story Diagram. Fourth, the model becomes too verbose, which decreases readability and usability. This would especially be the case when a lot more than two rules are in this situation. Alternatively, for these two rules, implicit rule scheduling would be a good choice.

Suppose that the whole transformation were implemented using implicit rule scheduling in the first place, in for example, a tool such as *AToM*³. With implicit rule scheduling, the `Match metadata` and `A table exists` states (this kind of sanity checks are rather common at the start of model transformations) could not have been scheduled before the other rules without having to rely on hand-written code or other tool-specific approaches, such as a dedicated state variable. Contradicting what has been stated previously, this is a reason why modeling in a language using explicit rule scheduling is a good choice for this problem.

5.1.3 Implementation in ND-SDM

It turns out that both implicit and explicit rule scheduling are needed to model the example of Figure 5.3 in a decent way. Therefore, the ND-SDM language is very suited as it introduces a new language construct for the SDM language that allows implicit rule scheduling. In Figure 5.4, a nondeterministic state is used, following the variant *keep matching rules until all failed to match*, just like in the household problem (see Section 3.1.2). Its prototype contained an application condition consisting of a `<<loop>>` stereotype (present of course on the state in Figure 5.4), and the actual prototype was given in Figure 4.6.

When using the transformation model of Figure 5.4 as input model, an execution of the higher order transformation described in Chapter 4 generates the transformation model in Figure 5.5. This transformation can, in contrast to Figure 5.4, be executed by *MoTMoT*, but has the same behavior as the ND-SDM transformation in Figure 5.4.

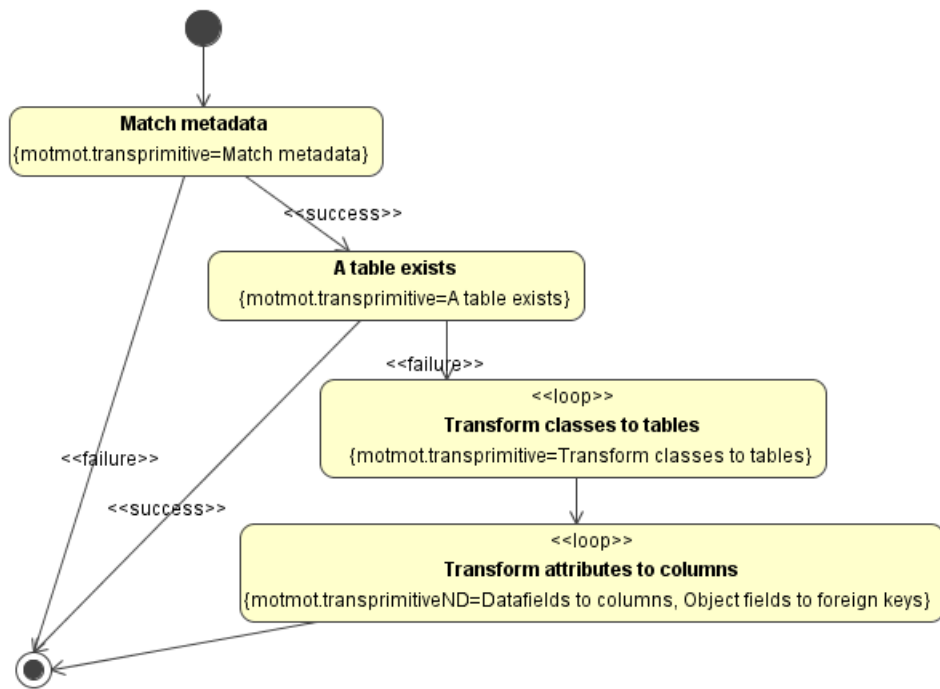


Figure 5.4: The UML to RDB transformation in ND-SDM

Using the nondeterministic state in Figure 5.4 solves the problems of the ND-SDM implementation of Figure 5.3. The two similar rules are scheduled as one state named *Transform attributes to columns*, which is conceptually a single action. The transformation model is less verbose, and it is now very clear at first glance that first tables and then attributes are transformed.

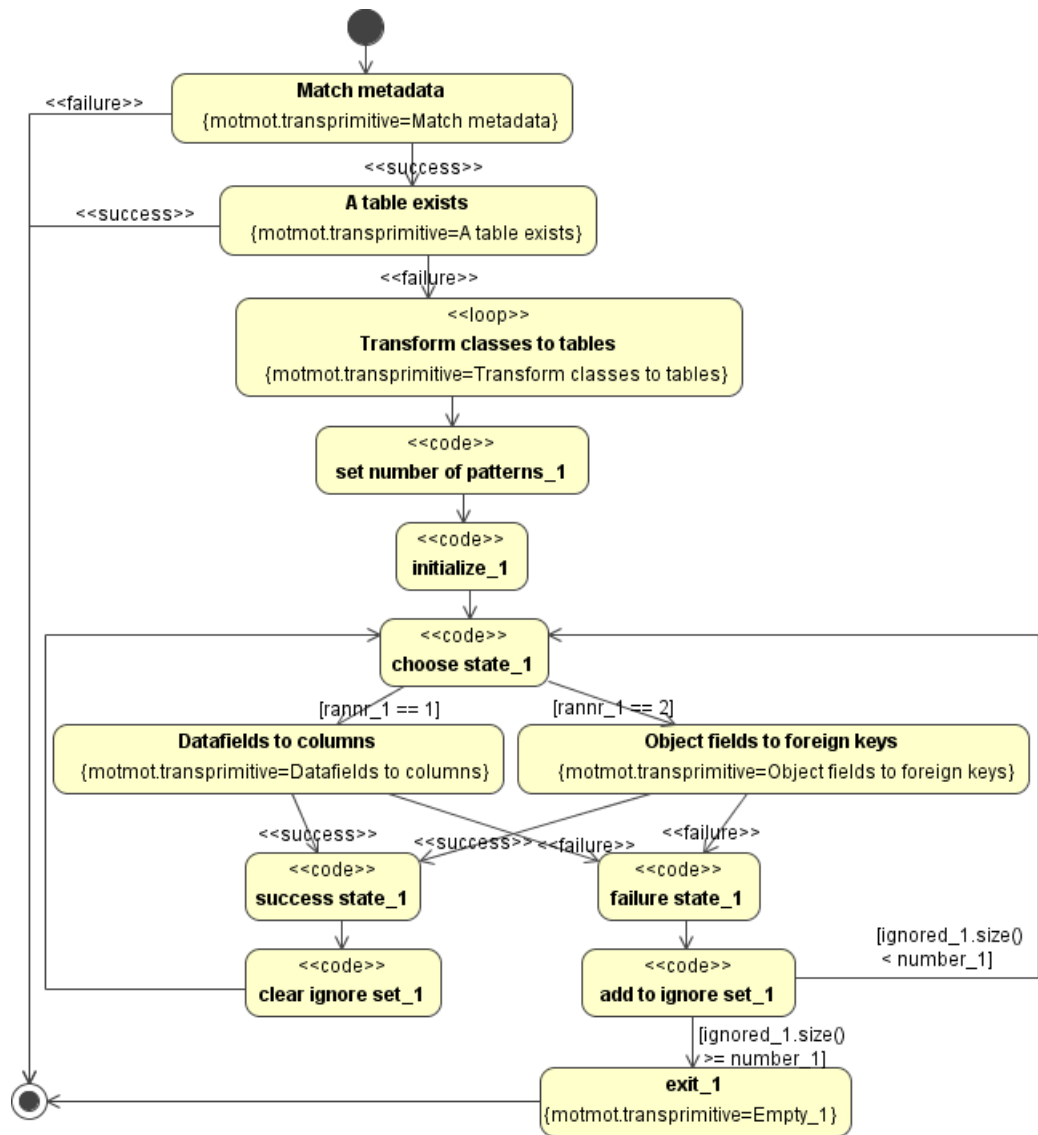


Figure 5.5: The flattened UML to RDB transformation

5.2 The Petri Net simulation transformation

5.2.1 Petri Net simulation

The next case is the simulation of the execution of Petri Nets (Peterson77). The execution of the Petri Net simulation transformation presented in this section preserves the structure of the Petri Net, but changes the marking of the Petri Net in such a way that it simulates firing an enabled transition, if there is one. Both the input and output model of the transformation are Petri Nets. This transformation is called endogenous (Mens06), since the input and output model have the same metamodel. Another difference with the previous case is that this time, the first implementation is in AToM³ (De Lara00), using implicit rule scheduling, rather than in MoTMoT, using explicit rule scheduling.

5.2.2 Implementation in AToM³

In this section a model transformation is introduced that implements the Petri Net simulation transformation of Petri Net models in AToM³ (see also Section 2.7). In AToM³, rules can be scheduled using priorities. Some advantages and disadvantages of using implicit rule scheduling will emerge.

As said previously, both the input and output model conform to the same metamodel, which is described below. The metamodel, the transformation model and the input model are all modeled using AToM³.

Petri Net access control example in AToM³

A marking of a Petri Net will be the input model of the transformation. As an example, Figure 5.6 models access control of one resource. Tokens in places represent processes in certain states (except for the tokens in the `access control` place: they represent available resources), and the firing of transitions triggers state changes. Three different processes can read the resource at the same time. A write action however can only happen if the resource is not being read. An execution of the Petri Net simulation transformation described below could perform for instance the firing of `t6`, where one token would move from `reader processing` to `ready to read`. Another execution could fire `t2`, which would empty the places `ready to write` and `access control`, and put a token in `writing`. Note that the resulting output model could serve as the input model of a new transformation execution, allowing chained transition firings.

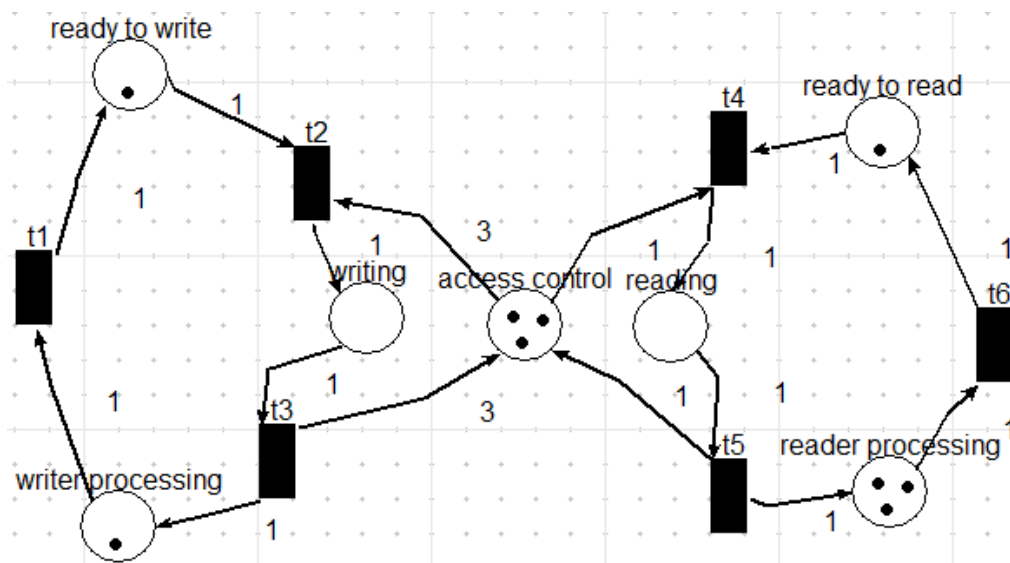


Figure 5.6: A Petri Net in ATOM³ that models access control

Petri Net metamodel in ATOM³

The Petri Net model of Figure 5.6 is conform to the metamodel of Figure 5.7. This metamodel is modeled and interpreted by ATOM³. It is an Entity-Relationship diagrams, one of the metalanguages provided by ATOM³. Besides using it as a metamodeling tool, ATOM³ will also be used as transformation tool below.

Figure 5.7 describes a metamodel of Petri Nets. PNPlaces and PNTransitions must be connected by arcs, either from a transition to a place (TransitionToPlace) or from a place to a transition(PlaceToTransition). A place can be connected to many transitions, and the other way around. Places can have tokens (0 by default), and a minimum (0 by default) and maximum (10000 by default) token capacity. Arcs can have a weight (1 by default) that depicts how many tokens are used when a transition fires.

The Petri Net simulation transformation in ATOM³

Tables 5.1, 5.2 and 5.3 present the rules of the Petri Net simulation transformation in ATOM³. For each rule, a priority is given (see Section 2.7). Each rule can have a LHS subgraph and a RHS subgraph, and some Python code for a condition and action for each rule is added.

The transformation goes as follows. First, with rules of the highest priority, some variables are initialized in the `initializing` phase. Once everything has been initialized, these rules won't match anymore, so the `StopInitializing` rule of priority 3 gets executed, which puts the algorithm in the `idle` phase. Then, it is tested whether the transitions can really fire. If

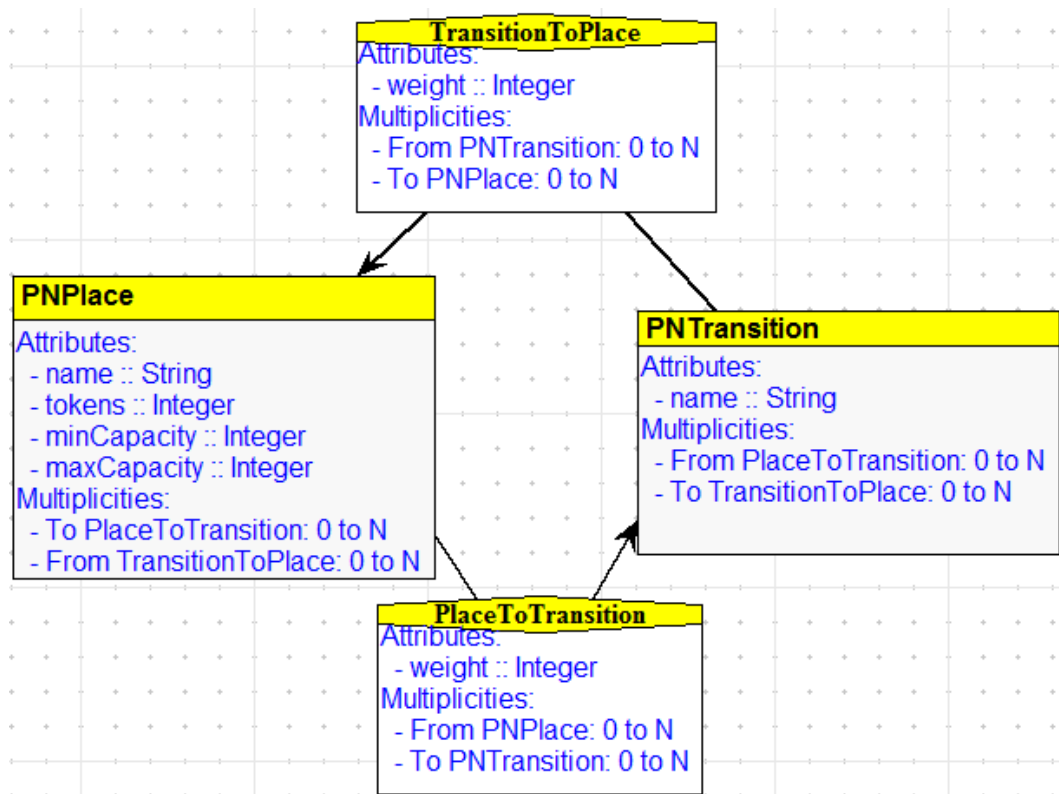



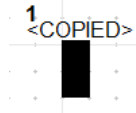
Figure 5.7: A metamodel of Petri Nets in AToM³

Priority 1 - Start

<i>LHS / Condition</i>	<i>RHS / Action</i>
<code>not hasattr(environment, "phase")</code>	<code>environment.phase = "initializing"</code>

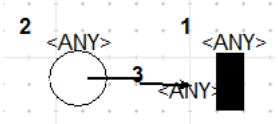
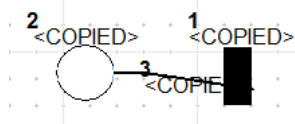
This rule starts the transformation by flagging that the algorithm is in initialization phase (by initializing the global variable `environment`). This rule does not need any graphs for pattern matching.

Priority 2 - InitializeTransition

<i>LHS / Condition</i>	<i>RHS / Action</i>
 <code>environment.phase == "initializing"</code> <code>and not transition.enabled == "unknown"</code>	 <code>transition.enabled = "unknown"</code>

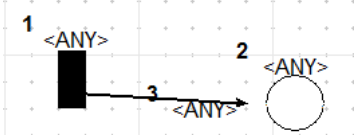
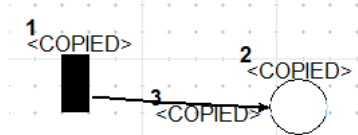
This rule, when in initialization phase, initializes a flag `enabled` to `unknown` for each transition.

Priority 2 - InitializeIncomingArc

<i>LHS / Condition</i>	<i>RHS / Action</i>
 <code>environment.phase == "initializing"</code> <code>and arc.fired</code>	 <code>arc.fired = False</code>

When in initialization phase, this rule initializes a flag `fired` to `False` on each arc that goes from a place to a transition.

Priority 2 - InitializeOutgoingArc

<i>LHS / Condition</i>	<i>RHS / Action</i>
 <code>environment.phase == "initializing"</code> <code>and arc.fired</code>	 <code>arc.fired = False</code>

When in initialization phase, this rule initializes a flag `fired` to `False` on each arc that goes from a transition to a place.

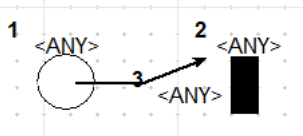
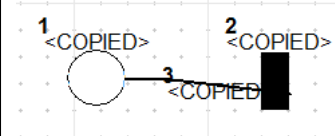
Table 5.1: Implementation of the Petri Net simulation transformation in AToM³

Priority 3 - StopInitializing

<i>LHS / Condition</i>	<i>RHS / Action</i>
<code>environment.phase == "initializing"</code>	<code>environment.phase = "idle"</code>

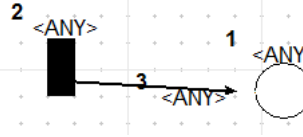
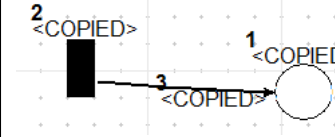
After everything has been initialized, the side-effects of this rule stop the initialization phase.

Priority 4 - DisableTransitionByInput

<i>LHS / Condition</i>	<i>RHS / Action</i>
 <pre> transition.enabled == "unknown" and environment.phase == "idle" and (place.tokens < arc.weight or place.tokens - arc.weight < place.minCapacity) </pre>	 <pre> transition.enabled = "no" </pre>

This rule rules out transitions that are supposedly enabled, but turn out to be not, when checking (one of its) incoming states. This can only happen in idle phase, otherwise a transition can be falsely disabled while firing.

Priority 4 - DisableTransitionByOutput

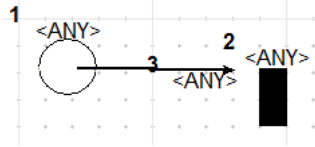
<i>LHS / Condition</i>	<i>RHS / Action</i>
 <pre> transition.enabled == "unknown" and environment.phase == "idle" and place.tokens + arc.weight > place.maxCapacity </pre>	 <pre> transition.enabled = "no" </pre>

This rule rules out transitions that are supposedly enabled, but turn out to be not, when checking (one of its) outgoing states.

Table 5.2: Implementation of the Petri Net simulation transformation in AToM³ (cont.)

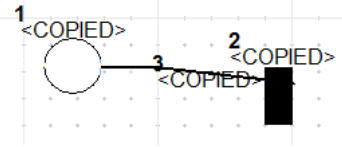
Priority 5 - DecreaseInput

LHS / Condition



`transition.enabled == "yes" and not arc.fired`

RHS / Action

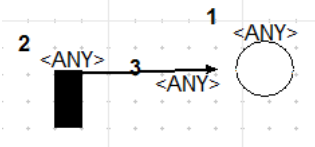


`place.tokens -= arc.weight`
`arc.fired = True`
`environment.phase == "firing"`

This rule decreases the number of tokens of the input states of the enabled transition.

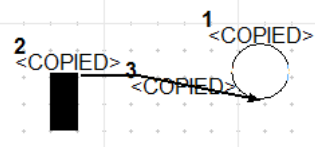
Priority 5 - IncreaseOutput

LHS / Condition



`transition.enabled == "yes" and not arc.fired`

RHS / Action



`place.tokens += arc.weight`
`arc.fired = True`
`environment.phase == "firing"`

This rule increases the number of tokens of the output states of the enabled transition.

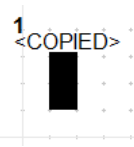
Priority 6 - ChooseTransition

LHS / Condition



`not environment.phase == "firing" and transition.enabled == "unknown"`

RHS / Action



`transition.enabled = "yes"`

A transition is chosen to try to fire.

Table 5.3: Implementation of the Petri Net simulation transformation in AToM³ (cont.)

a transition cannot, `DisableTransitionByInput` Or `DisableTransitionByOutput` disables the transition. Then, the next two rules will not match, because they require a transition to be enabled, and for now, some are disabled and for some it is unknown whether they are enabled. As a consequence, they are skipped, and the scheduling algorithm reaches the rule `ChooseTransition` with lowest priority. This allows a random transition that was not disabled to be enabled. Now, `DecreaseInput` and `IncreaseOutput` can match and consequently the number of tokens of the input place(s) is decreased and the number of tokens of the output place(s) is increased. Once this is done, none of the rules can match since the algorithm is in the `firing` phase and the firing is done, so the transformation ends. On the other hand, if all transitions would have been disabled by the `DisableTransitionByInput` and `DisableTransitionByOutput` rules, the transformation would end because no transition can be chosen in `ChooseTransition`.

In short, this transformation transforms a Petri Net marking into another marking where one transition has fired. The transformation can easily be extended so that it keeps firing transitions until there are no possibilities left. This would be done by adding a rule with lowest priority that does nothing but putting the algorithm back in the initialization phase. This is not done here, because one execution of the transformation is the firing of a single transition, if there is one enabled. Note that in the case of the access control example, this causes the transformation execution to never terminate.

Advantages and disadvantages of using AToM³

As a conclusion, we sum up the advantages and disadvantages of the use of AToM³, a transformation language using only implicit rule scheduling:

- overall, the transformation is very concise and elegant. This is due to the concept of declarative programming that is associated with implicit scheduling. On the scope of rule scheduling, it is modeled "what" must happen, and not "how";
- unfortunately, the modeler is obliged to annotate the rules with some Python code. This code is not visible in the LHS and RHS graphs. This is in contrast with the visual graph grammar language;
- unfortunately, in order to allow a correct termination of the transformation if there is no enabled transition in the input model, a string flag `enabled` must be used. This complicates the whole transformation model¹;
- some loops over one rule can be modeled using only that rule;

¹Note that in a way, due to the `enabled` variable as well as the `phase` variable, the transformation model gets suspiciously many characteristics of a state machine, which is deterministic.

- similar rules can be somewhat bundled by giving them the same priority. However, they are still separated, as any two other rules;
- there are three phases in the algorithm: the initializing phase, the phase where a transition is fired, and an idle phase. This is not immediately recognizable in this transformation model. Instead, unfortunately, the artificial usage of a global variable `environment.phase` is required.

5.2.3 Implementation in ND-SDM

This section introduces a model transformation that implements the Petri Net simulation transformation of Petri Net models in ND-SDM. Both implicit and explicit rule scheduling will be used. The disadvantages that emerged from the implementation in AToM³ will turn out to be solved.

Petri Net access control example in ND-SDM

Figure 5.8 shows the same Petri Net with the same marking as the access control example of the AToM³ implementation in Figure 5.6. Since the Petri Net metamodel is now written in the UML, the model looks a bit different graphically, but it still models resources with access control. Tags are used to model the number of tokens, the minimum and maximum capacity (both not visual) of a place, and the weight of an arc.

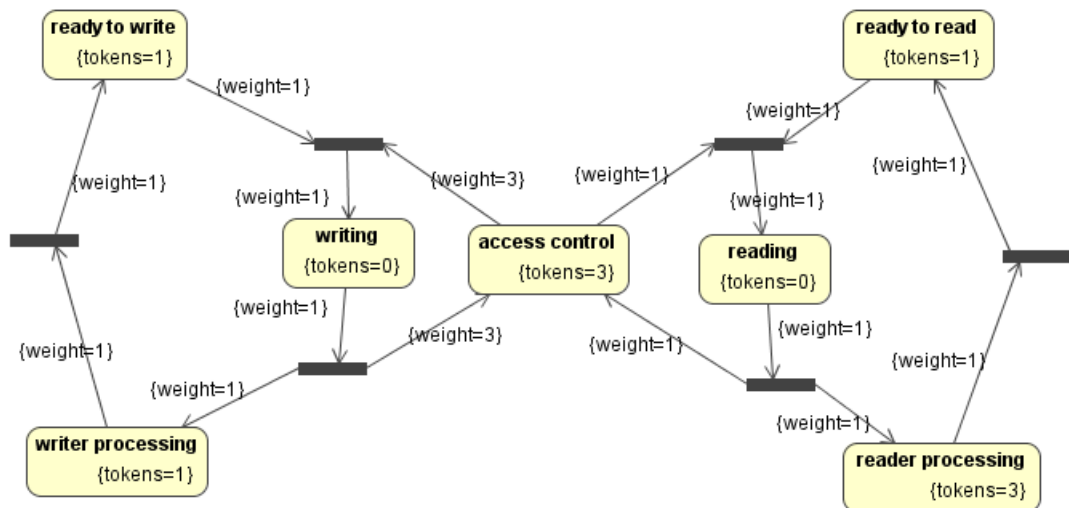


Figure 5.8: A Petri Net in the UML that models access control

Petri Net Metamodel

The metamodel of the ND-SDM implementation is UML Activity Diagrams. The metamodel for Activity Diagrams is shown in the Appendix in Figure A.6. *ActionStates* are used to model places, *PseudoStates* (i.e. forks and joins) are transitions, and *Transitions* model arcs.

The Activity Diagrams are enriched with some *TagDefinitions*, to model the Petri Net constructs such as weights, tokens and capacities. Tags are part of the so-called extension mechanisms in the UML, and their metamodel is shown in Figure A.3. Tag definitions are declared for *weight*, which are applicable on *Transitions*, and *tokens*, *minCapacity* and *maxCapacity*, which are applicable on *ActionStates*. As said, *minCapacity* and *maxCapacity* are marked to be not visual in their *ActionState*. (In the access control example, all minimum capacities are 0 and all maximum capacities are infinite.)

The Petri Net simulation transformation in ND-SDM

Both rule scheduling constructs of AToM³, namely nondeterminism and priorities, are also available in ND-SDM. Therefore, it is possible to mindlessly copy the scheduling of the rules of any AToM³ transformation. Although this would be interesting when one wants to transform transformation models automatically from one tool to another, doing so does not necessarily results in the most elegant result, as the availability of constructs for explicit rule scheduling is entirely ignored.

In the context of the Petri Net simulation transformation, while discussing the AToM³ implementation, it has been suggested that some constructs for explicit rule scheduling could be quite useful. The three different phases can be scheduled successively. The Story Diagram of the ND-SDM transformation is shown in Figure 5.9. In the *Start* state, some metadata is matched, like in the UML to RDB transformation in Figure 5.3. It is called *Start* to emphasize the similarity to the *Start* rule of the AToM³ implementation. After all, both are rules that initialize the environment.

Next, all transitions and arcs are initialized in the *initialize* state. This is done in the same way as in the AToM³ rules. *initialize* references two rules. Because UML's *Transitions* are used to model both incoming and outgoing arcs, there is no need for two separate *InitializeIncomingArc* and *InitializeOutgoingArc* rules. Next, transitions that are in fact not enabled are marked as disabled by checking their input and output places.

Then, a transition that is not disabled is chosen in the `ChooseTransition` state. If no enabled transition can be found, the transformation is terminated. Otherwise, the marking of the Petri Net is changed by changing the number of tokens of the input and output places of the chosen transition. It turns out that the referenced packages `DecreaseInput` and `IncreaseOutput` do not just contain patterns, but they are actual Story Diagrams. This can be considered as a case of explicit rule scheduling (i.e. the Story Diagram) inside an implicitly scheduled environment (i.e. the nondeterministic state). The nondeterministic state itself is of course a case of implicit rule scheduling inside an explicitly scheduled environment. Indeed, both are possible, making implicit rule scheduling and explicit rule scheduling virtually equal in hierarchy². When the `fire transition` state is finished, the transformation terminates.

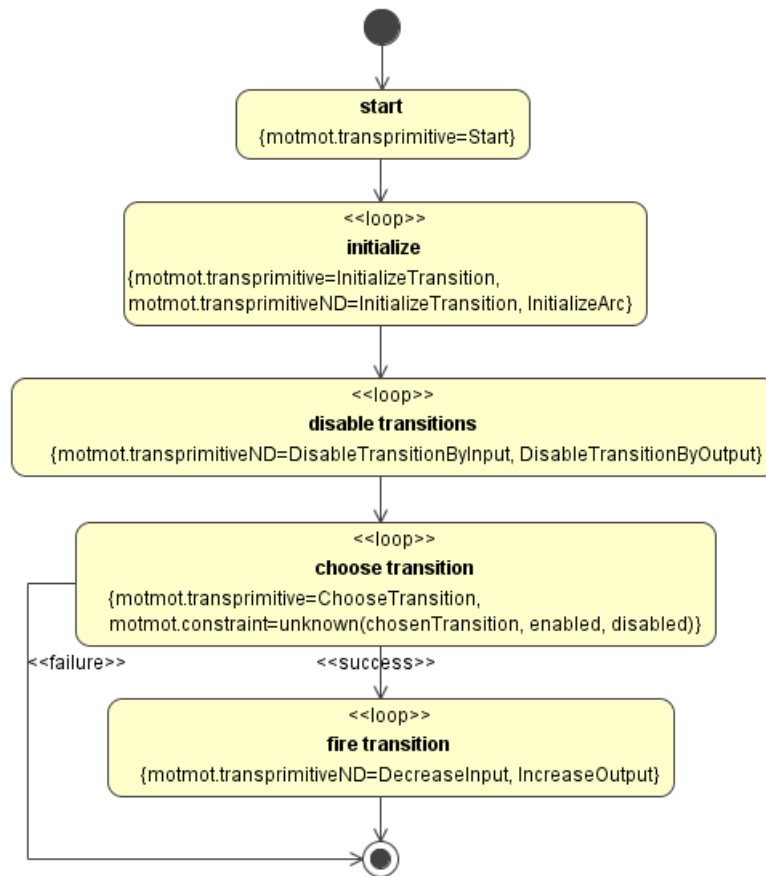


Figure 5.9: The Petri Net simulation transformation in ND-SDM

²If one wants to use only explicit rule scheduling in MoTMoT, one is still obliged to surround the implicitly scheduled rules with an initial state and a final state, so there still remains a small notion of explicit rule scheduling.

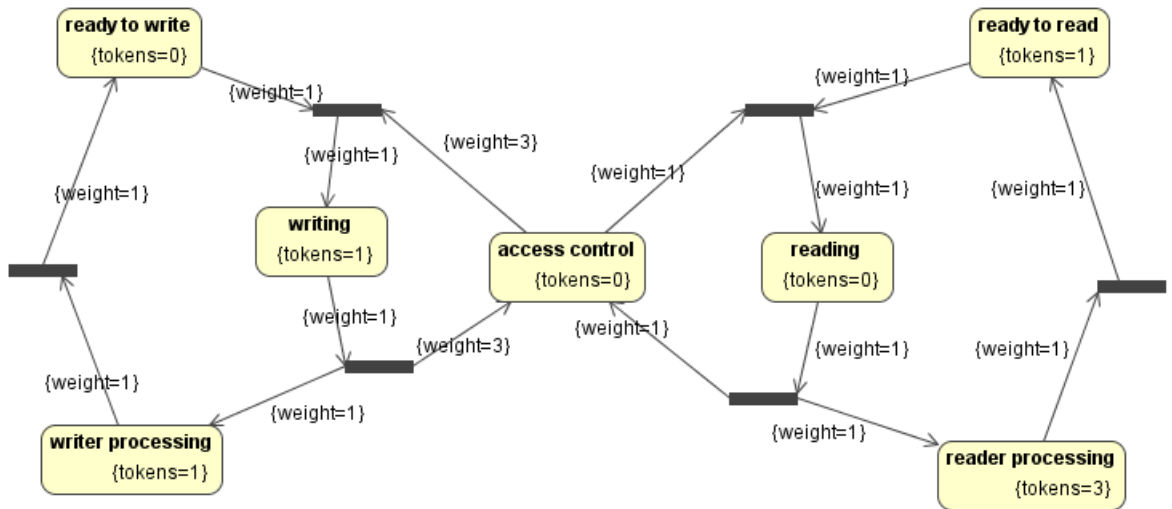


Figure 5.10: A possible marking after the execution of the transformation

An execution of the transformation possibly yields the marking given in Figure 5.10, where a process starts to write on the resource. Executing the higher order transformation described in Chapter 4 on the ND-SDM model, results in a MoTMoT compatible transformation model with the same behavior, given in Figure 5.11. Three nondeterministic states are transformed, all using the variant of the household problem (see Figure 4.6). It turns out that, in terms of scheduling, this variant exhibits the same behavior as an AToM³ rule of the highest priority.

Advantages of using ND-SDM

By using the hybrid transformation language, the disadvantages of the language with implicit rule scheduling are solved, while the advantages remain (see Section 5.2.2 about the advantages and disadvantages of implicit rule scheduling):

- the Story Diagram of the transformation is even more concise and elegant since similar rules can be modeled in one state;
- the use of additional platform-dependent code is very limited to some conditions (as boolean expressions), which can be imitated in virtually any language because no new variables are declared in code, and no global variables are used. The use of platform-dependent code could be solved by using the profile described in Section 4.4.7;
- when there is no enabled transition, the transformation stops. This is visually modeled by the conditional rule `choose transition;`

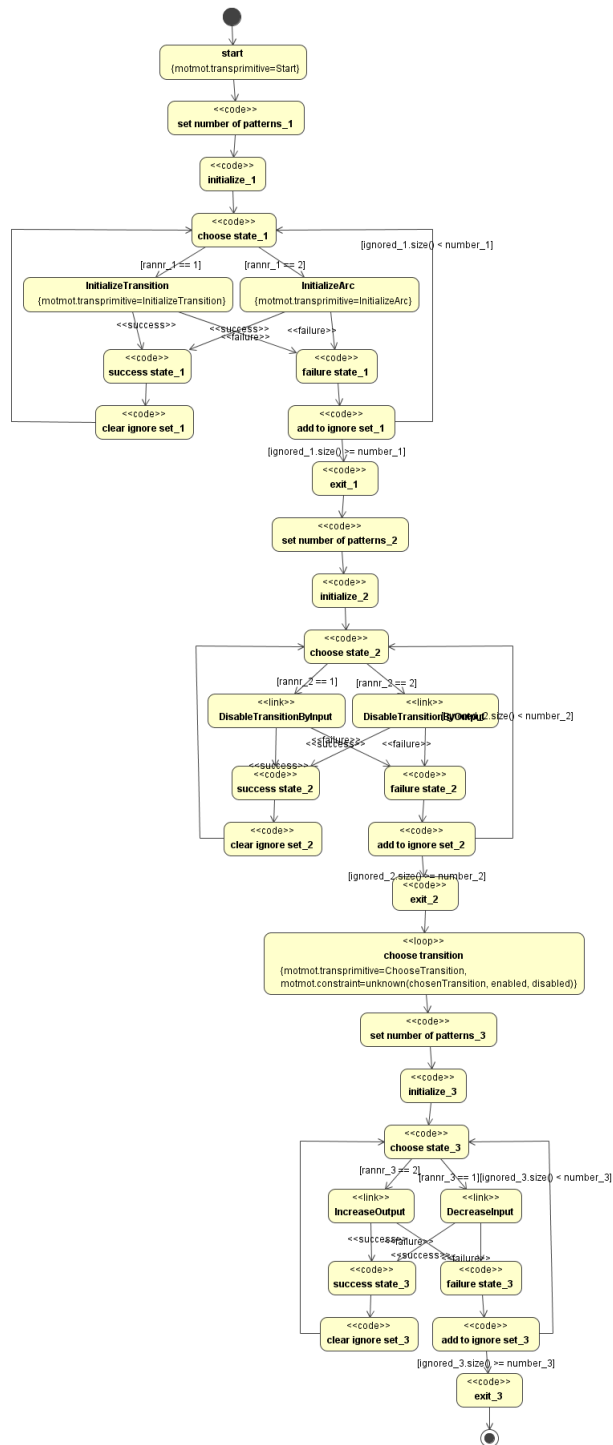


Figure 5.11: The flattened Petri Net simulation transformation

- iteratively evaluating rules, as in AToM³, can still be modeled when using the *«loop»* stereotype and the right variant of the nondeterministic state;
- similar rules are modeled in only one state. Conceptually similar actions are bundled, making the transformation model more readable and intuitive;
- the three phases of the algorithm can be modeled visibly and intuitively. No global variable needs to be used for modeling different phases;
- since priorities can still be used, their advantages remain. However, it turns out that this transformation can be better modeled without priorities.

5.3 The poker game simulation transformation

5.3.1 Poker simulation

Poker (Wikipedia09) is a card game based on a ranking system. In a game, each player gets a number of cards. For each player, the highest five card *hand* can be determined by the predefined ranking rules. The player with the highest hand wins. The ranking depends on the combination that can be formed from the *suits* (spades, hearts, diamonds and clubs) and *spots* (2 to 10, jack, queen, king, ace) of the cards. The ranking is given in Table 5.4.

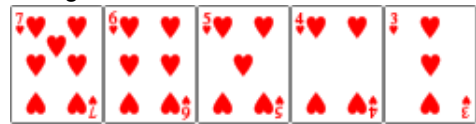
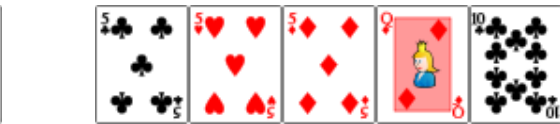
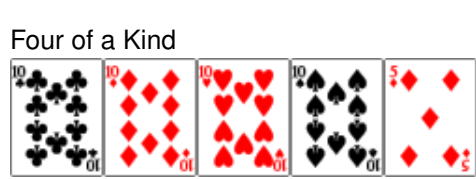
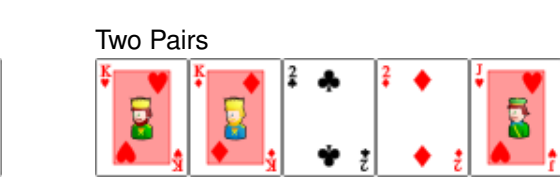
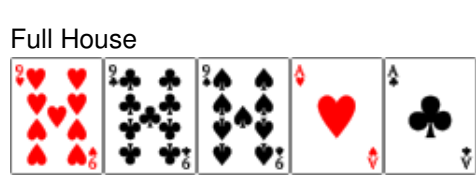
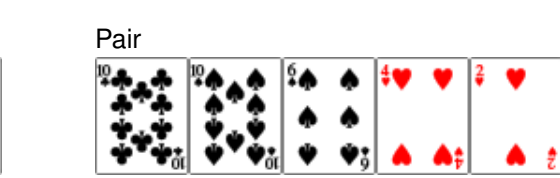

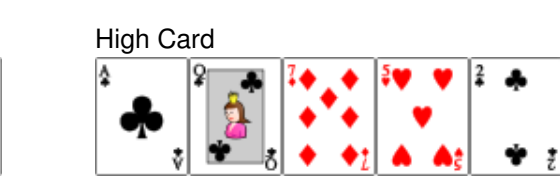

<p>Straight Flush</p> 	<p>Three of a Kind</p> 
<p>Four of a Kind</p> 	<p>Two Pairs</p> 
<p>Full House</p> 	<p>Pair</p> 
<p>Flush</p> 	<p>High Card</p> 
<p>Straight</p> 	

Table 5.4: The poker ranking with examples, in descending order (Wikipedia09)

The poker simulation transformation presented in this section follows the popular *Texas hold 'em* variant. Besides dealing cards for each player, *community cards* are dealt too, which can be used by all players to construct their best hand. Each player gets two cards, then three community cards are dealt (this is called the *flop*), then another one (the *turn*), and a final one (the *river*). The players can bet between these dealing phases (also called *betting rounds*). At the end, each player can construct the best hand using his two cards and the five community cards, and the player with the best hand wins.

The goal of the poker simulation transformation is to give each player its ranking and to point out the winner of the game. The poker simulation transformation will generate a deck of cards, deal cards to the players, and deal the community cards in the three betting rounds (the flop, the turn, and the river). Because in *Texas hold 'em* betting can be done after each round, it is useful to calculate after each round what is the best hand for each player. Once the river is dealt, the player with the highest hand is denoted as the winner. This transformation illustrates the use of priorities and another prototype as the ones used in the two previous transformation examples. This time, only the ND-SDM implementation is given.

5.3.2 Implementation in ND-SDM

Similar to the other transformations in MoTMoT, the metamodel of the poker game is the UML. This time, Class Diagrams are used. A card is a UML *Class*, and a player, the community cards and the deck are UML *Packages*, because they are all able to hold cards. To make a difference between different cards, *Stereotypes* are used. For example, the diamond queen is a class with two stereotypes: `diamond` and `Queen`. Players have a stereotype `Player`, and the board of community cards has a stereotype `Community Cards`. When the transformation is executed, a `Winner` stereotype is put on the player with the highest hand. Each player also gets a stereotype denoting its ranking, following the rankings of Table 5.4.

Figure 5.12 shows the input model of the poker game simulation transformation. The only thing that is modeled is which players will join the game. First, the card deck is generated, the board of community cards is created and each player is dealt two cards, as shown in Figure 5.13. It looks like Patrick has been dealt some pretty good cards: two queens. After the (imaginary) betting round, the flop is dealt and it seems that Frank is winning (also called *in front*), as he can construct a straight (kings to 9) from his cards and the community cards. Patrick is also doing OK, as his hand has three queens. Mary and Jane's hands both have only a pair for now. At the turn (Figure 5.15), a three of spades is dealt. Now Jane's hand has a three of a kind, but Mary's hand has 5 spades, forming a Flush, so she is in

front now. At the river (Figure 5.16), the last queen of the deck is dealt. So Jane can form a full house of threes over queens, but Patrick’s hand has now a four of a kind, so he is the winner. Figure 5.16 is the output model of the transformation, where the final hand rankings and the winner are denoted.

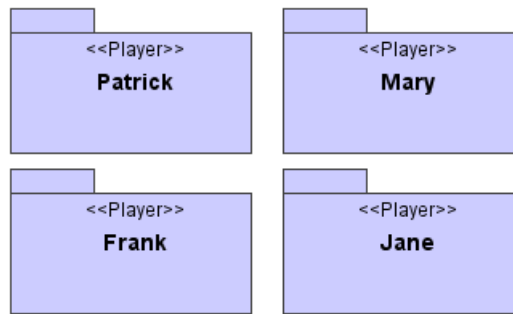


Figure 5.12: The input model of the poker simulation transformation

The Story Diagram of the ND-SDM transformation is shown in Figure 5.17. It starts with a series of sanity checks, which look up the metadata, such as the stereotypes for the different ranks (Full House, Pair, etc.), spots (2 to 10, Jack, etc.) and suits (diamonds, spades, etc.). An ordering is created for the different rank stereotypes by generating some associations between them (an association between High Card and Pair, between Pair and Two Pair, and so forth). The same is done for the spots, as they too have an ordering. These patterns are scheduled using a nondeterministic state with a *<<sanity check>>* stereotype. This variant of nondeterministic scheduling will try all the patterns once, until one fails. If all patterns match, the *<<success>>* transition is followed. If one fails, the *<<failure>>* transition is followed. Note that this is the same variant as in the restaurant problem of Section 3.1.2. The prototype was shown in Figure 4.8. Note that this time, the application condition is not an empty state, but a state with a *<<sanity check>>* stereotype.

Some of the patterns, such as Match ranks, are split into many patterns. This is done for performance reasons. After all, in order to match a whole pattern (or graph), a subgraph is built up by matching one node at a time. Therefore, for each node of the pattern, all possibilities can be tried. In this way, the search space for a pattern is exponential to its number of nodes. In other words, large patterns can be huge performance bottlenecks. By splitting up the pattern, the cumulative search space of the resulting smaller patterns is a lot smaller.

If the sanity checks were successful, the card deck and all 52 cards are generated. Then, each player is dealt two cards. In the deal player cards state a player is given a card as

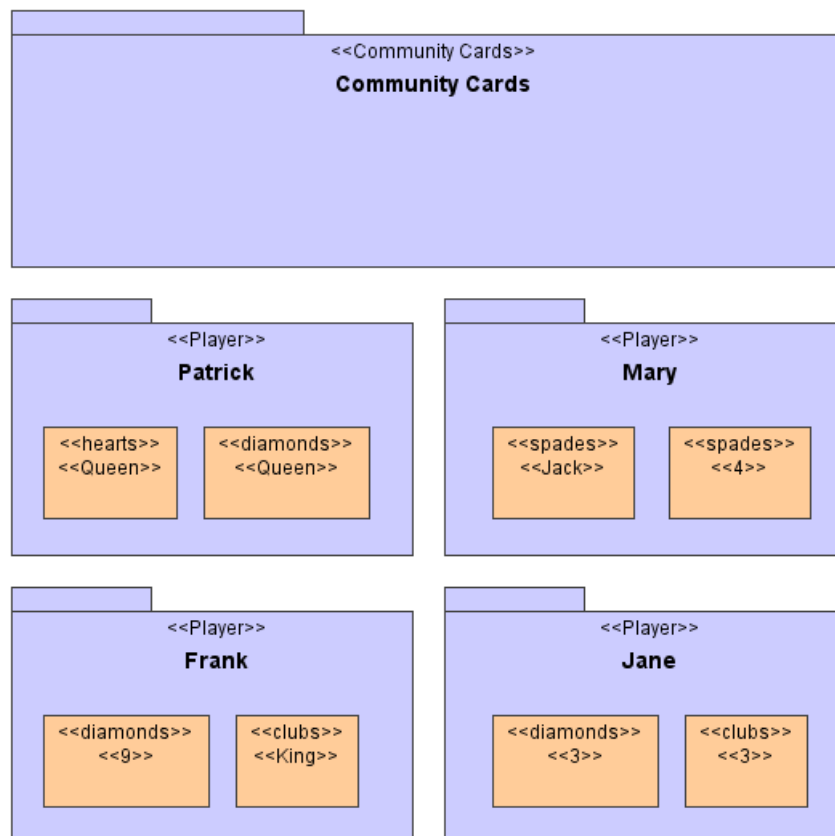


Figure 5.13: The players are dealt two cards each

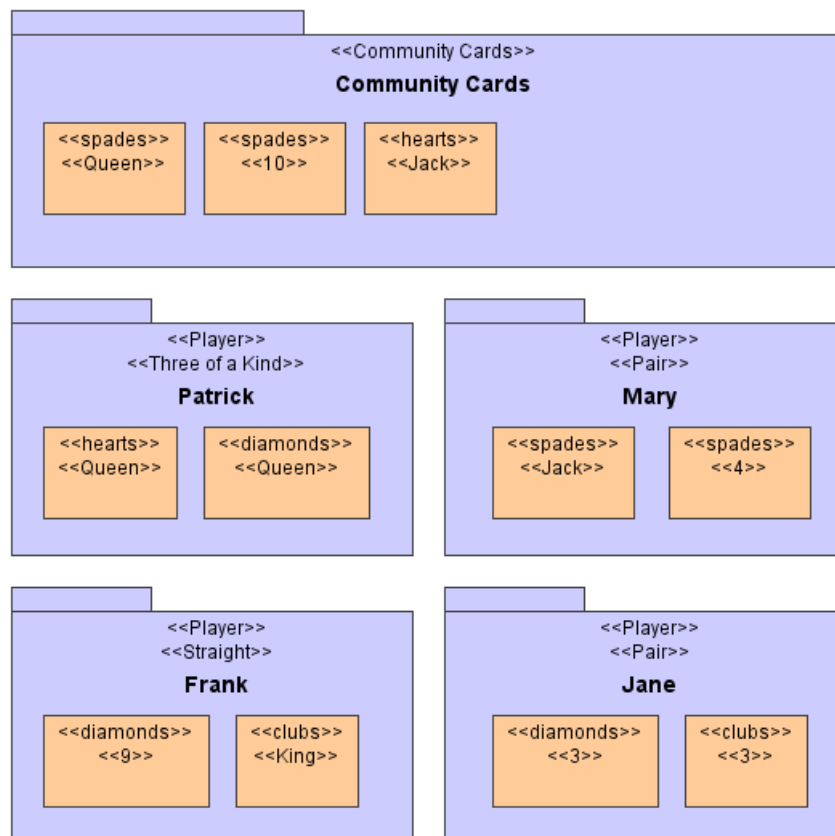


Figure 5.14: Frank is winning after the flop

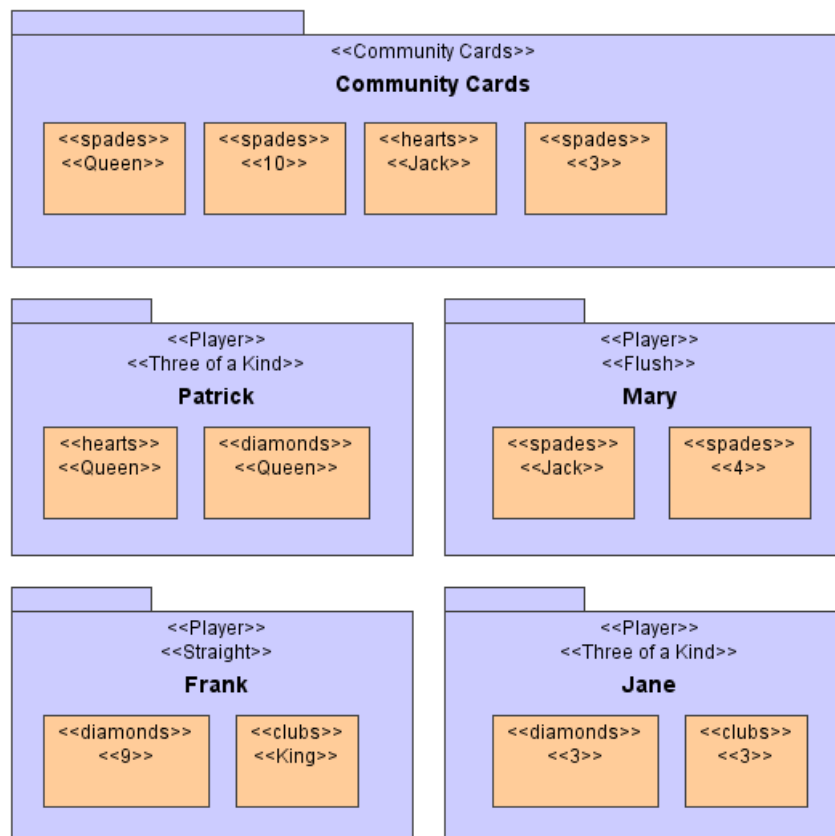


Figure 5.15: Mary is winning after the turn

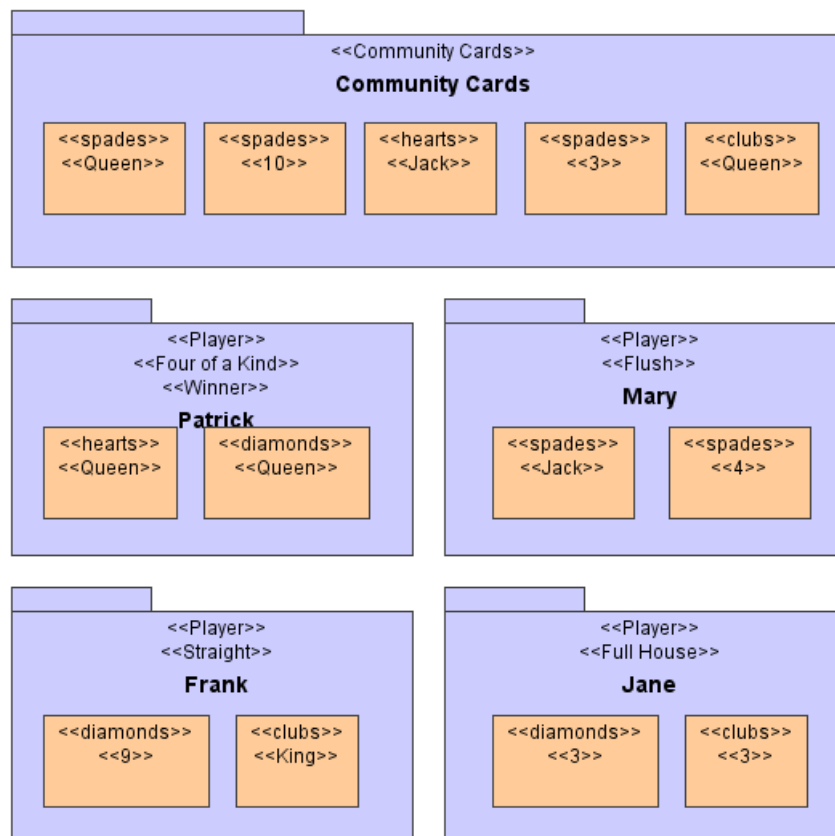


Figure 5.16: Patrick has won after the river

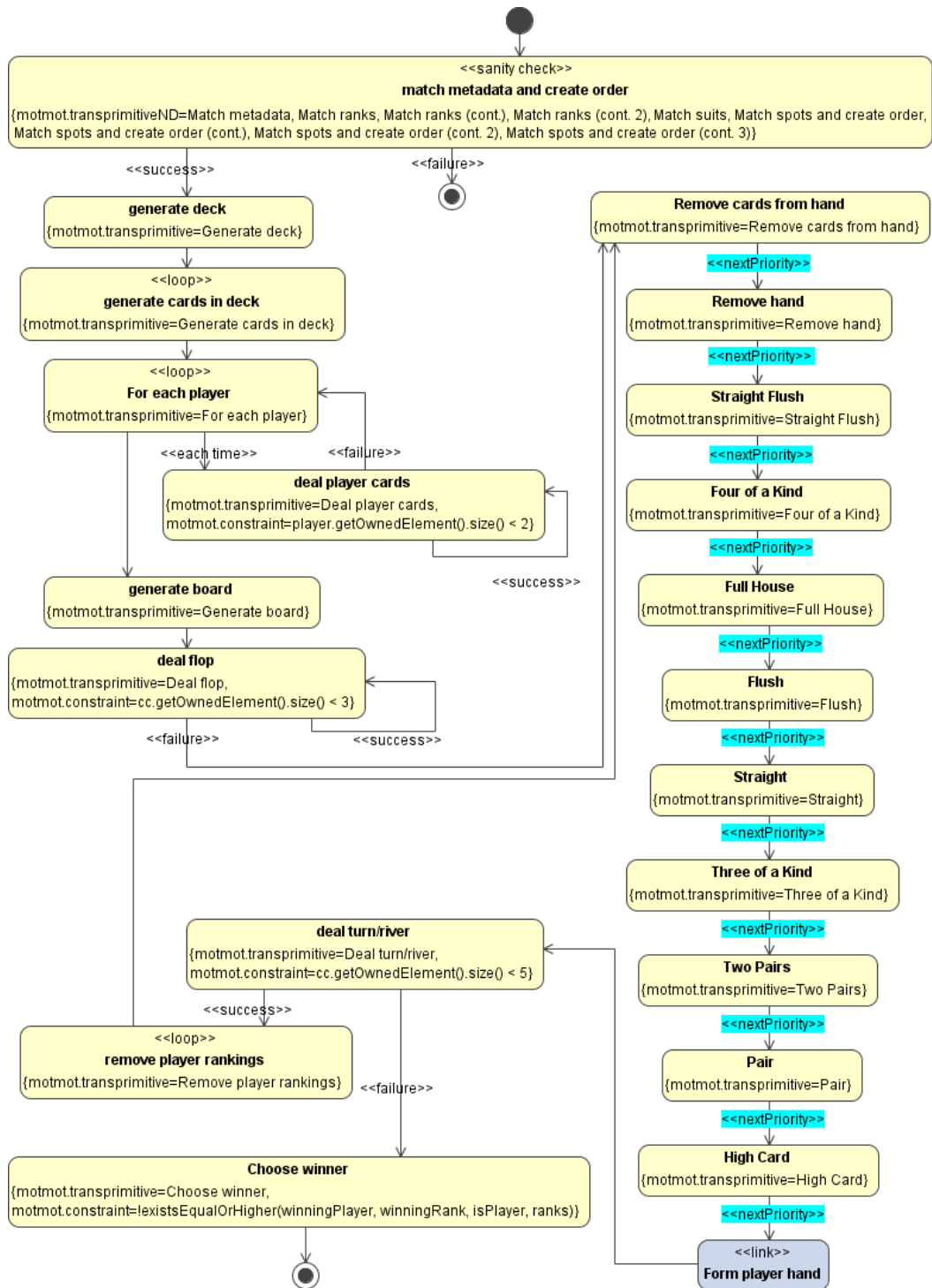


Figure 5.17: The Story Diagram of the poker simulation transformation

long as he has less than two cards. Next, the board of community cards is generated, by simply creating a new package with the `«Community Cards»` stereotype. Then, the three flop cards are dealt one by one in the `deal flop` state by putting them in the `«Community Cards»` package.

The next twelve states are scheduled using priorities, denoted by the blue `«nextPriority»` transitions. As discussed in Section 3.3, this means that patterns with higher priorities that match will be executed first. `Remove cards from hand` has the highest priority, and `Form player hand` the lowest. The first time, none of the patterns with higher priority match, so the call of the `Form player hand` state is made. In the called Story Diagram (which is not shown), a player is chosen and the player's hand is constructed. The hand consists of the cards that are available for the player to construct the highest hand. In other words, the hand of a player contains five cards at the moment: the cards of the player plus the three community cards (dealt in the flop).

Once the hand has been created, the highest ranking of the hand has to be found. The two states with highest priority can not match, as long as the player has not yet received his ranking for this betting round. However, one of the ranking states (`Straight Flush to High Card`) can match now. Because of the priority structure of the states, the highest possible ranking will match. When following the example above, the pattern of the `Straight` state will match for Frank. The `High Card` pattern would match too, but is not even evaluated because of the priorities. All these ranking patterns are rather similar.

For example, the pattern for matching a full house is shown in Figure 5.18. `fh_c1` to `fh_c5` represent the cards, which have to be part of the hand `fh_hand`. Three cards have to have the same spot `fh_spot`, and two other cards have to have the same spot `fh_spot2`. If the pattern matches, a stereotype `fullhouse` is put on the current player `fh_player`.

Once the best hand for the current player is found, the pattern of the `Remove cards from hand` state matches. In this pattern, a card is removed from the hand. This state keeps matching until all cards are removed from the hand. This way, a new hand can be constructed for the next player. Then, the `Remove cards from hand` pattern fails to match, so the `Remove hand` pattern matches and removes the hand. Because there is no hand, none of the patterns with higher priority match, and the situation is the same as when first entering the priority flow. Hence, the call from the `Form player hand` is made and a new unranked player is chosen, and the same path is traversed for this player.

After all players have been ranked, the call from `Form player hand` will return `false`, so the priority chain is left, as none of the patterns in the priority chain matches. Next, the `deal`

turn/river state is entered. In its pattern, one card is dealt to the community cards. Once the turn is dealt, the rankings are removed from the players so that the ranking can start all over again³. After each player is ranked once again, the river is dealt, and all players are ranked once more. When the deal turn/river state is entered after the river, its pattern fails, and the player with the highest hand is denoted as winner.

To summarize, different constructs of explicit and implicit rule scheduling can be mixed up in the Story Diagram. Note that in the priority chain of this diagram, not only patterns, but also a whole Story Diagram, are scheduled implicitly. This is again a case of explicit rule scheduling inside an implicitly scheduled environment (see also the Petri Net transformation of Section 5.2). Finally, Figure 5.19 shows the diagram that results from executing the HOT with the poker transformation as input.

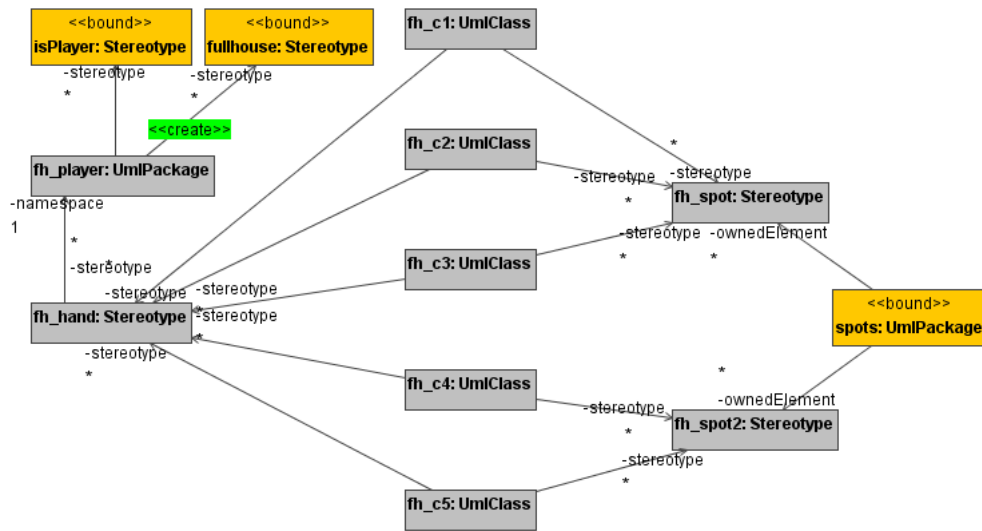


Figure 5.18: The pattern for matching a full house

³Removing the rankings seems to make the ranking of the players after the flop useless, but in practice, the current rankings can be printed, or the transformation can be paused to check the rankings.

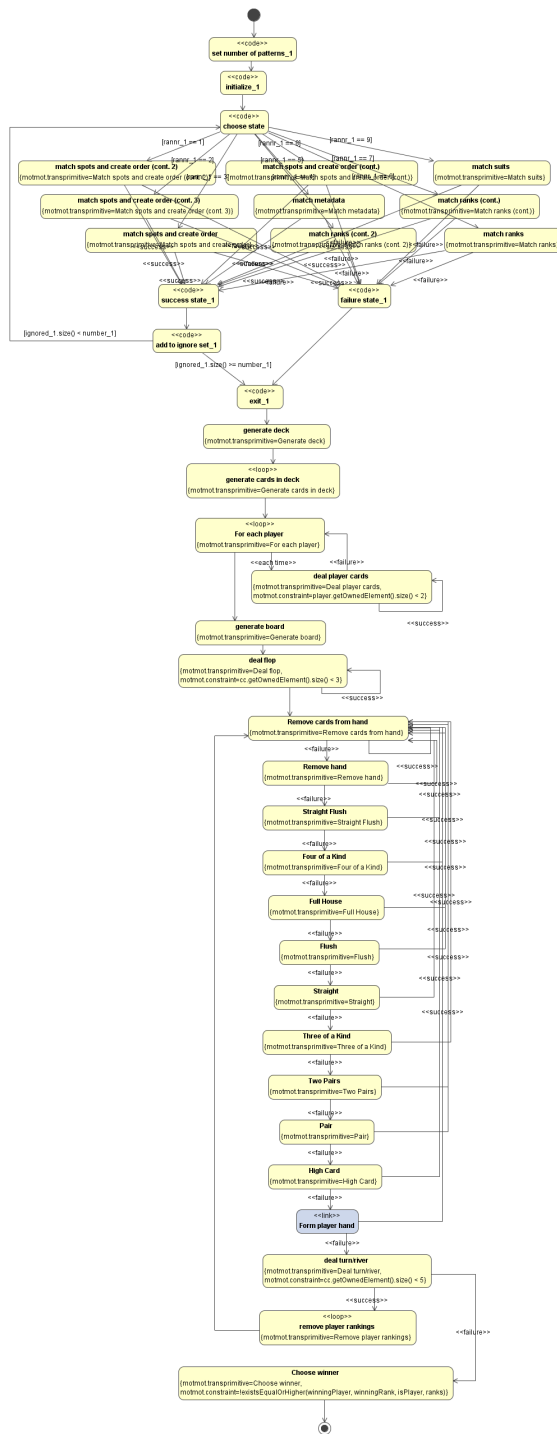


Figure 5.19: The Story Diagram after the execution of the HOT

Negative application conditions in SDM

In Chapter 4, we implemented hybrid rule scheduling as a higher order transformation. It turns out that many other language extensions can be implemented using this technique. To illustrate this, we implement negative application conditions (NACs) as a HOT. NACs are another useful language extension that is not available in the MoTMoT transformation language.

6.1 Negative application conditions

Negative application conditions in graph grammars were first presented by Habel et al. (Habel95). A NAC is a condition of a graph that *must not* be matched in order to match the graph. For example, a pattern that matches persons that do not have children can be elegantly expressed using a NAC. The NAC that would express that a person that *does* have a child *can not* match.

The example above only describes one type of NAC. In my opinion, four kinds of NACs can be useful in SDM:

- a NAC on a state: in contrast to a usual state, if the pattern of the state matches, *false* is returned. If the state does not match, *true* is returned;

- a NAC on an association in a pattern: a subgraph matches the pattern if there is no association that matches the NAC association between the matched elements. Table 6.1 presents a simple graph example to illustrate this;
- a NAC on an attribute of an element of a pattern: a subgraph matches the pattern if the value of the NAC attribute does not equal the value of the attribute of the matching element. This is very similar to a constraint on the element saying `attr != value`;
- a NAC on a subgraph of a pattern: a subgraph can match the pattern if it is not connected in the same way with any subgraph that matches the NAC subgraph. The implementation of this type is discussed in this chapter, as it is the most complicated type of NACs. Table 6.2 presents a simple graph example to illustrate this type of NAC.

In his description of Story Driven Modeling, Zündorf describes so-called negative graph elements (Zündorf02). They have the same behavior as NACs on a subgraph, but only one element can be the NAC instead of a whole subgraph.

A NAC on a subgraph can be used in the example of the childless persons. A NAC on an association can not be used, as using a NAC on an association would imply that at least one child must exist, which is not the intent of the pattern.

As said, an implementation of NACs on subgraphs of patterns is presented in this chapter. The evaluation (the examples of Table 6.2 follows these semantics) of a pattern takes three steps:

1. the subgraph without the NAC subgraph must be matched. If no matching subgraph can be found, the pattern fails to match. If a matching subgraph can be found, continue with the next step;
2. the NAC subgraph is evaluated. If no match can be found, the pattern matches. If a matching subgraph is found, continue with the next step;
3. the edges connecting the matched NAC subgraph with the other matched subgraph are evaluated. If not all of them match, the NAC passes and the pattern matches. If they all match, this subgraph fails to match and another one is tried in the first step.

This type of NAC is used inside a Story Pattern, thus NACs, as language construct, differ a lot from nondeterministic scheduling, where everything happens at the level of the Story Diagrams. To illustrate NACs, the UML to RDB example of Section 5.1 is revisited. It turns out that the patterns of this transformation (not shown in Section 5.1) are very suitable

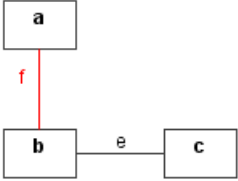
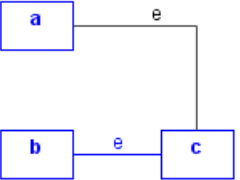
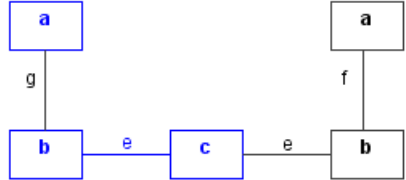
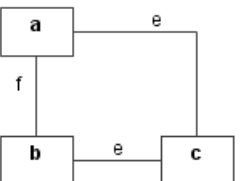
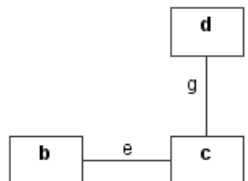
<p>Pattern</p> 	
<p>The pattern has a NAC on the edge f between a and b.</p>	
<p>Pattern match</p>  <p>A subgraph of this graph matches the pattern as a, b, c and the edge e match, while there is no edge f between a and b.</p>	<p>Pattern match</p>  <p>A subgraph matches, as there is no edge labeled f between the two left nodes a and b.</p>
<p>No match</p>  <p>No subgraph matches, as there is an edge f between the only occurrences of a and b.</p>	<p>No match</p>  <p>No subgraph matches, as there is no node a.</p>

Table 6.1: An example of a NAC association

<p>Pattern</p> <pre> graph TD a[a] --- f --- d[d] a --- g --- b[b] b --- f --- c[c] style a stroke:#f00 style d stroke:#f00 </pre>	
<p>The pattern has a NAC covering the nodes a and d, and the edge f in between. Note that the NAC would exhibit the same behavior if the edge g was included.</p>	
<p>Pattern match</p> <pre> graph TD e[e] --- h --- b[b] b --- f --- c[c] style b stroke:#00f style c stroke:#00f </pre> <p>A subgraph of this graph matches the pattern as the NAC subgraph can not be matched.</p>	<p>Pattern match</p> <pre> graph TD a[a] --- g --- b[b] b --- f --- c[c] style b stroke:#00f style c stroke:#00f </pre> <p>A subgraph matches, as the NAC subgraph can not be matched.</p>
<p>Pattern match</p> <pre> graph TD a[a] --- f --- d[d] a --- h --- b[b] d --- g --- c[c] b --- f --- c[c] style b stroke:#00f style c stroke:#00f </pre> <p>A subgraph matches, as there is no edge g between node b of the matching subgraph and node a of the matching NAC subgraph.</p>	<p>No match</p> <pre> graph TD a[a] --- f --- d[d] a --- g --- b[b] d --- h --- c[c] b --- f --- c[c] </pre> <p>No subgraph matches, as there is an edge g between node b of the matching subgraph and node a of the matching NAC subgraph.</p>

Table 6.2: An example of a NAC subgraph

for using NACs. Since nondeterministic scheduling can now be considered a part of the transformation language, we can start from the diagram with the nondeterministic state.

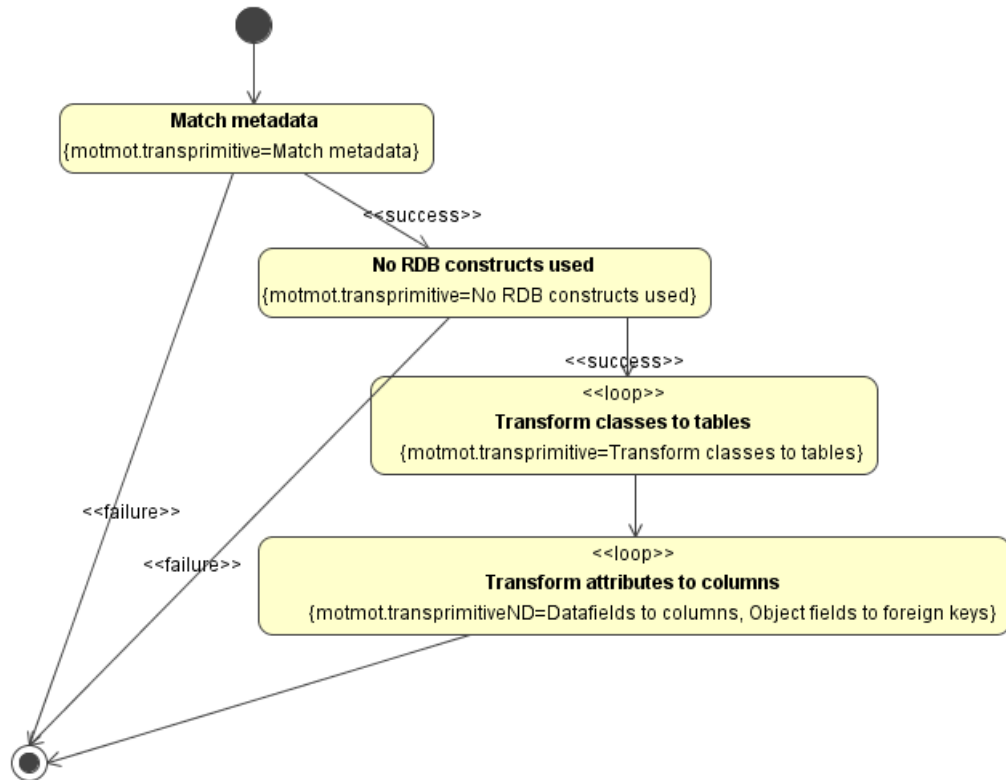


Figure 6.1: The UML to RDB transformation in ND-SDM

The Story Diagram shown in Figure 6.1, is however slightly different from the one of Figure 5.4. Instead of the `A table exists`, there is now a state called `No RDB constructs used`, and the outgoing `<<success>>` and `<<failure>>` transitions have been switched. This state is much more intuitive, as we are not really trying to match tables, but we are checking if there are no RDB instances in the model. Before, the transformation would terminate if `Match metadata` was not matched. Then, it would terminate if `A table exists` was indeed matched, which can be confusing. Now, both patterns simply have to match in order to continue.

The first state in Figure 6.1, `Match metadata`, binds some useful metadata that will be used in the following patterns. Then, the `No RDB constructs used` state is entered. The pattern is presented in Figure 6.2. The pink UML `Package` with the `<<NAC>>` stereotype immediately stands out. This is how NACs will be visualized. This way, the transformation language stays conform to the UML, as this is still a requisite of course. The pattern

matches if no class *c* can be found which has a *sdm_metatype* tag (i.e. the tag named *motmot.metatype*) that references a class that resides somewhere in the RDB metamodel. In other words, no instance of RDB may exist. Note that the element *c* has to be inside the `<<NAC>>` package. If not, the pattern states that a class must be found in *pkg*, and this is not what we want to express. On the other hand, it does not matter whether bound elements are inside or outside the `<<NAC>>` package, as they match anyway.

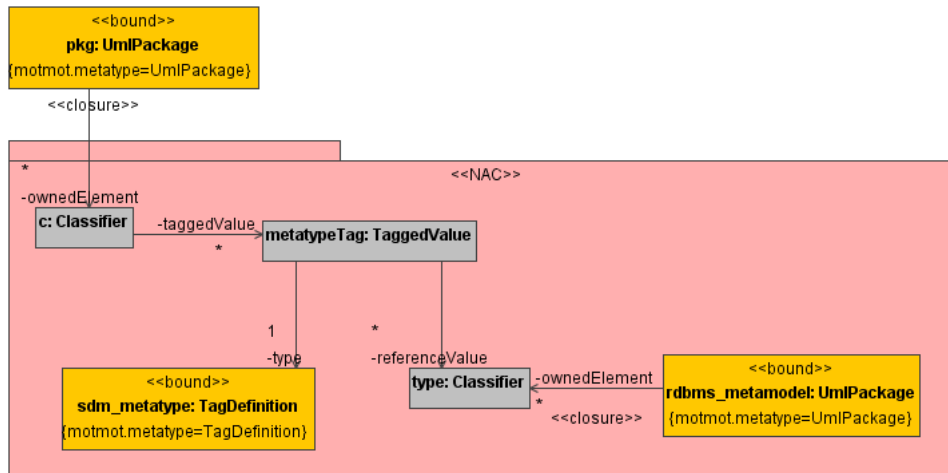


Figure 6.2: This pattern matches when no RDB constructs can be found

If no RDB constructs can be found, the actual transformation can start with the iterative Transform classes to tables state, shown in Figure 6.3. In this pattern, for a class *c* in *pkg*, a table *newTable* is created (with *rdbms_Table* as *motmot.metatype*). The tables and classes are both modeled as UML *Classes* (see Figures 5.1 and 5.2), so special care must be taken that newly created tables are not transformed as well. This is expressed by the NAC, which is very similar of the previous one, stating that *c* can not be an instance of RDB. This time, *sdm_metatype* is outside the `<<NAC>>` package, and there are two associations that connect the NAC to the rest of the pattern. After each class has been transformed, the attributes are transformed in the Transform attributes to columns state. This is the nondeterministic state referencing two patterns, `Datafields to columns` and `Object fields to foreign keys`.

The pattern `Datafields to columns` is rather large, so in order to present it in this thesis it is split up into two views¹, the *match* view (shown in Figure 6.4) and the *create* view

¹Using views fits very well into the larger context of Model Driven Engineering. Making views does not alter the transformation model in any way, as there is still only one `Datafields to columns` pattern. In fact, all diagrams shown in this thesis are just views. An actual model is a tree containing the elements.

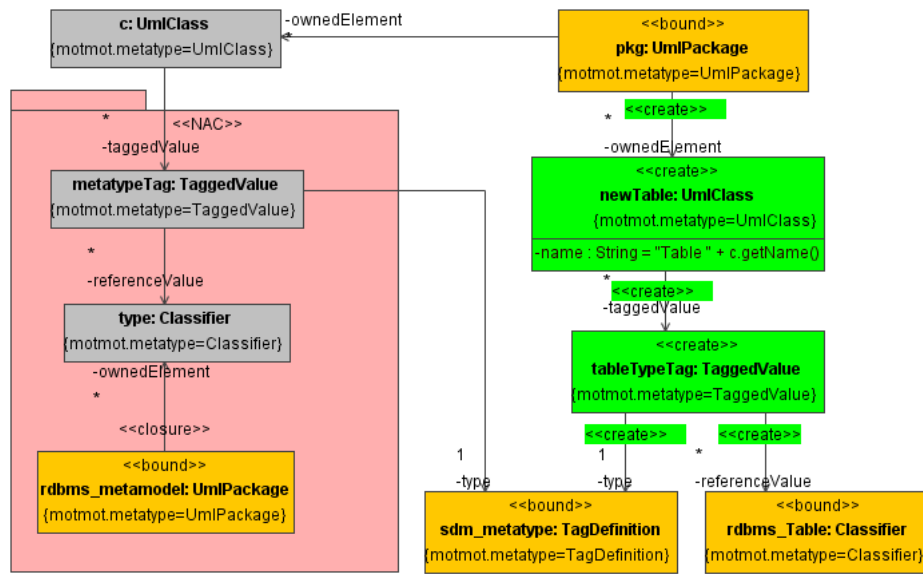


Figure 6.3: This pattern transforms classes to tables

(shown in Figure 6.5). In the *match* view, it is shown that an attribute `datafield` matches, if the following holds:

- it does not have the `class_nonpersistent` stereotype (the upper NAC). Non-persistent attributes do not become persistent columns. Note that in this case, a NAC on the association between `datafield` and `class_nonpersistent` would have the same meaning;
- its type `type` resides in the package `javaPkg` (i.e. `java.*`), as this means that the attribute is of a "simple" type, and must indeed be transformed to a column, not to a foreign key;
- its class `classWithDataField` is not an RDB instance (the middle NAC, again, similar to the one shown in Figure 6.2);
- its class has an according table `tableWithDataField`², created in the Transform classes to tables pattern;
- the column is not already created (the bottom NAC).³

For an attribute that confirms to the conditions above, the new column `newColumn` is created as shown in Figure 6.5. This includes creating a UML *Class* for the column with an

²Note that `tableWithDataField` is matched by name in the constraint. With traceability mechanisms, this could be expressed much more elegantly by using for example a traceability link between `tableWithDataField` and `classWithDataField`, created in the Transform classes to tables pattern.

³This too could be expressed more elegantly with a NAC on a traceability link.

attribute representing the type of the column, as well as the UML *Association* between the column and the table.

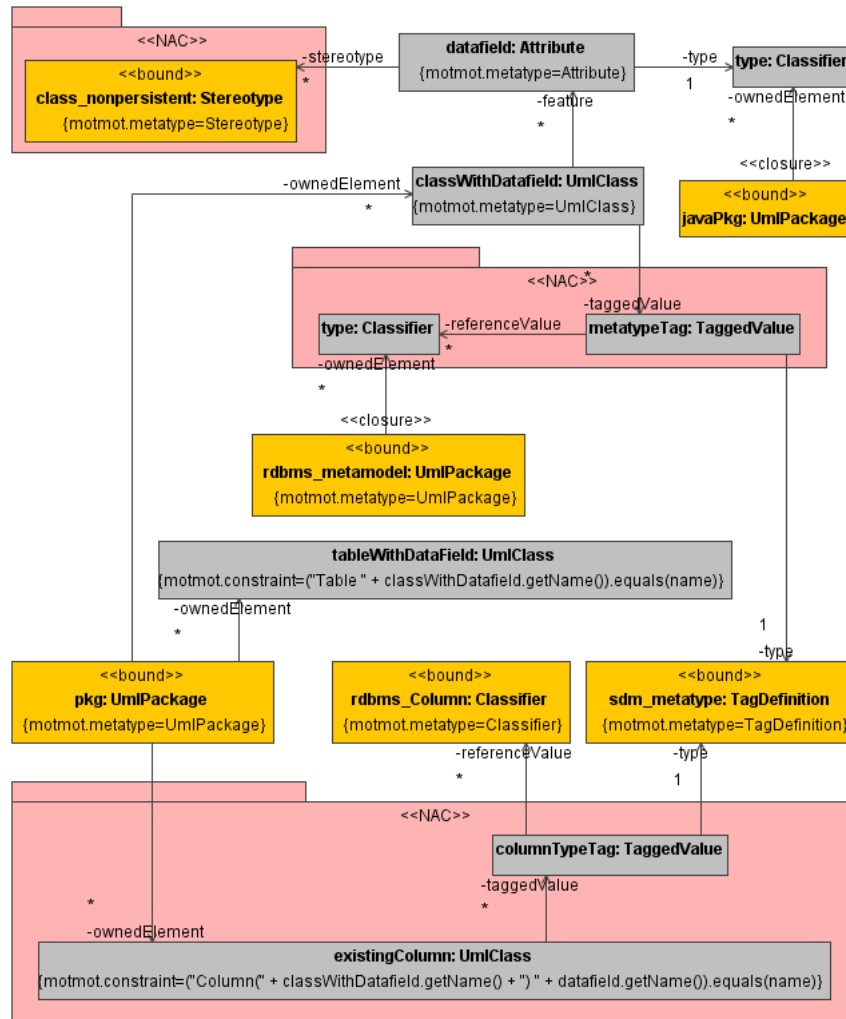


Figure 6.4: The *match* view of the pattern that transforms datafields

The Object fields to foreign keys pattern is very similar and is therefore not shown. It also uses three NACs, similar to those of the match view of Figure 6.4. A new *UmlClass* must be created, but this time two associations must be created: one from the table where the foreign key is part of, and one to the table it references (i.e. the type of the foreign key) (see Figure 5.2). Once all attributes are transformed, the transformation is finished.

To summarize, negative application conditions are very useful in numerous occasions. A pink package with a <<NAC>> stereotype is visually very recognizable and makes a clear

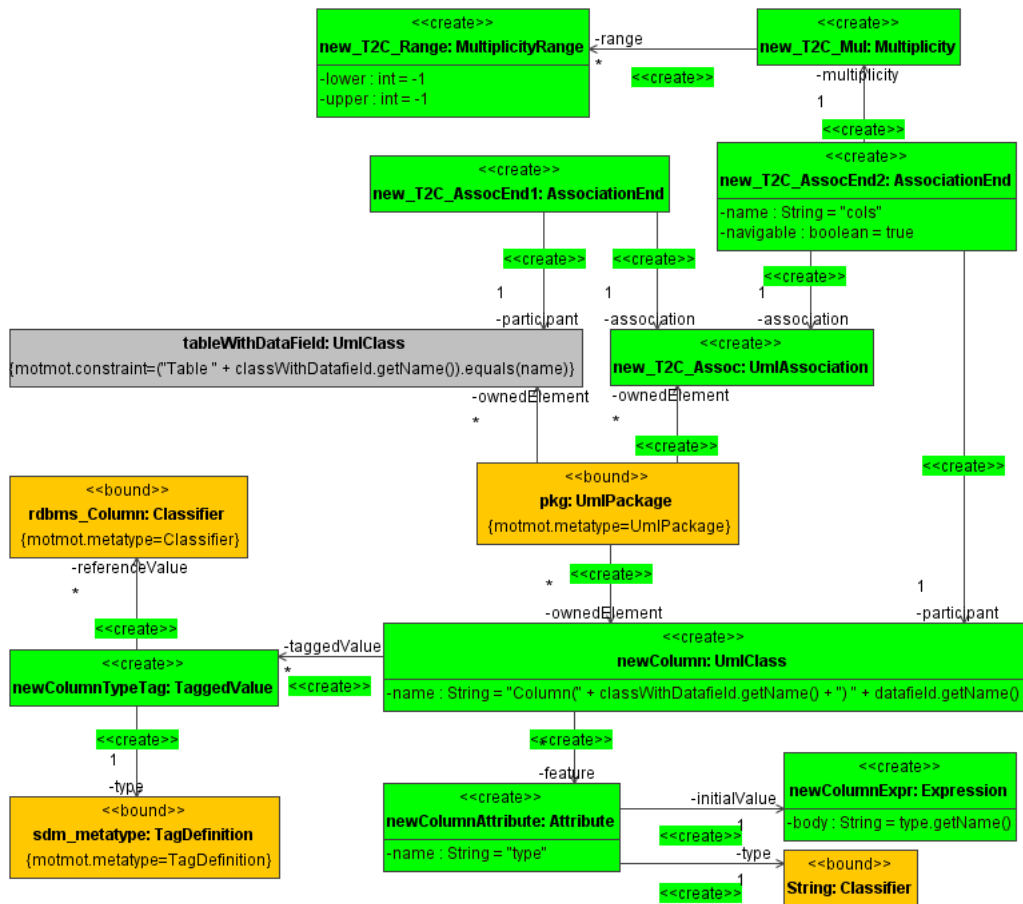


Figure 6.5: The *create* view of the pattern that transforms datafields

distinction between the subgraph that must be matched and the subgraph(s) that must not be matched. Because of their usefulness, negative application conditions are implemented in many transformation tools, such as AGG (Tántzer06), VIATRA (Csertán02) and GReAT (Balasubramanian06). Progres (Schürr09b) and Fujaba (Nickel00) support NACs that are only applicable on one element instead of entire subgraphs (see also (Zündorf02)). However, NACs have not yet been implemented in a tool-independent way.

6.2 The negative application conditions implementation

In this section we present the implementation of negative application conditions. Similar to the nondeterministic scheduling implementation, a higher order transformation will be used that transforms the transformation models written in the extended language to transformation models written in the language that is well-known to MoTMoT. The first thing to do is once again to find an equivalent of the NAC construct that has the same behavior, but is written in the Core profile of SDM (without the `<<NAC>>` stereotype), or ND-SDM.

6.2.1 The MoTMoT equivalent

NACs are conditions that must not be met. In the MoTMoT equivalent, a NAC can therefore become a boolean constraint (i.e. a *motmot.constraint*) on the pattern. More in detail, the constraint will contain a call to a newly created Story Diagram containing the NAC, which returns a boolean. This returned value will be negated, thus making sure that calls that return *true* will fail the match.

Figures 6.6 to 6.9 present the equivalent of the `Transform classes to tables` pattern of Figure 6.3. In Figure 6.6, the NAC is removed. Instead, a constraint is added to the state of the Story Diagram. Figure 6.7 shows the relevant part of the Story Diagram. It says that the return value of the call to a method `Transform_classes_to_tables_2` must be *false*. The implementation of this method is modeled by the Story Diagram of Figure 6.8. This Story Diagram simply returns whether or not the pattern of the `Transform classes to tables_2` state matches. This pattern is shown in Figure 6.9, and it consists of the NAC subgraph and the elements that were connected to it (the so-called context elements). These context elements are now bound, because they have already been matched in the class transformation pattern. Note that all bound elements are passed as parameters in the method call in Figure 6.7.

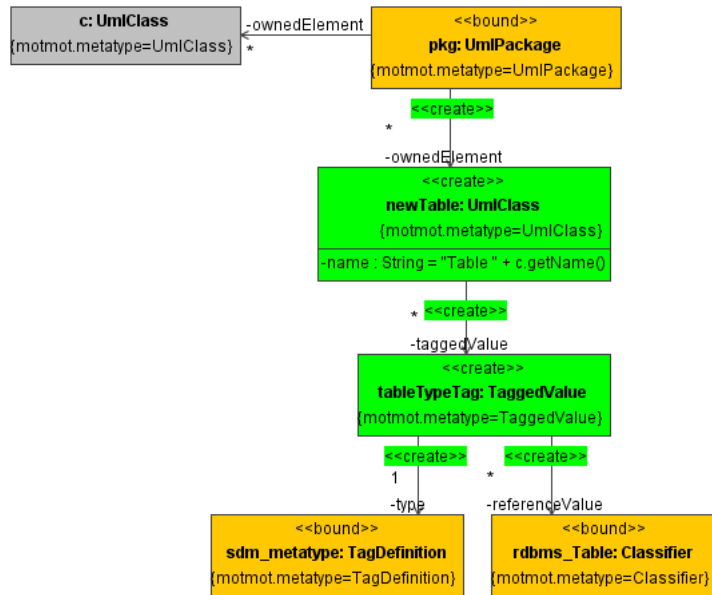


Figure 6.6: The class transformation pattern of the MoTMoT equivalent

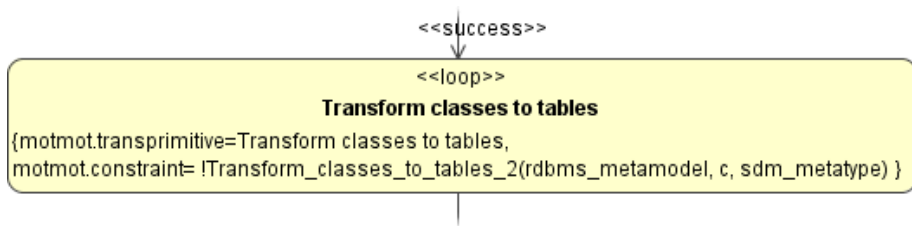


Figure 6.7: The Story Diagram of the flattened UML to RDB transformation

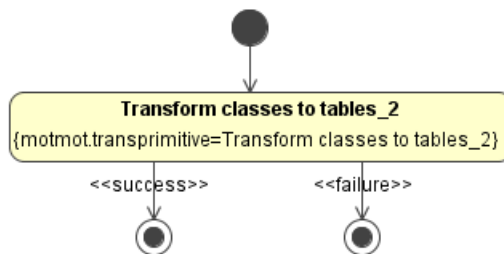


Figure 6.8: The Story Diagram of the NAC

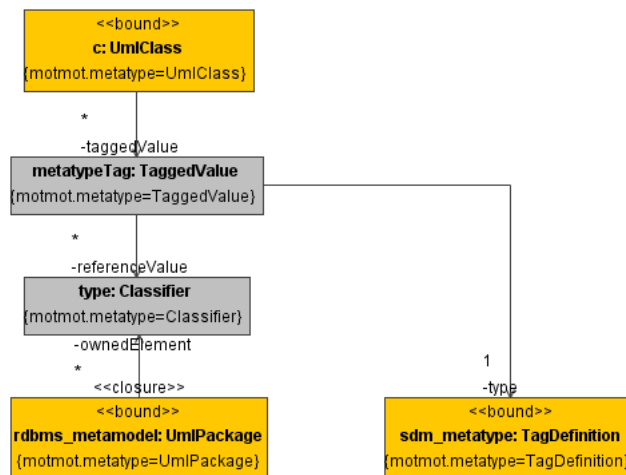


Figure 6.9: The Story Pattern of the NAC

If more than one NAC is used in the same pattern, the constraint is composed of the logical conjunction of the different negated method calls. This results in a tag of the form `motmot.constraint=!condition1 && !condition2 && !condition3`. Alternatively, if one wants to model the different NACs in disjunction, one has to put all NACs of the pattern in one package. In this way the NAC will only match if all NACs match. This means that the pattern can match as long as not all NACs match. The constraint would simply be of the form `motmot.constraint=!condition1`. This was also noticed by Habel et al. (Habel95).

6.2.2 The negative application conditions HOT

In the NAC HOT, occurrences of NACs are transformed to their equivalents, similar to the HOT for nondeterministic scheduling. In order to do this, the NAC HOT has to perform the following tasks:

- create a boolean method for the NAC;
- create a method call in the constraint of the patterns' state (similar to the constraint in Figure 6.7);
- create a Story Diagram (similar to the Story Diagram of Figure 6.8) that represents the implementation of the NAC method. This is the Story Diagram that is entered when the NAC method is called;
- separate the subgraph in the `<<NAC>>` package from its pattern (resulting in patterns similar to the patterns Figure 6.6 and 6.9). This is done by performing the following:

- create a new pattern for the NAC. Since patterns and NACs are both represented by packages, this is simply done by removing the `<<NAC>>` stereotype from the `<<NAC>>` package;
- copy all the context nodes from the original pattern to the NAC pattern, including their associations with elements from the NAC pattern, their tags and their stereotypes, and add a `<<bound>>` stereotype;
- remove all dangling associations from the original pattern.

Figure 6.10 shows the Story Diagram of the HOT. First, some metadata is matched and bound in the `match metadata` and `match java profile metadata` states. Then, an ID that will give each NAC equivalent a unique name is initialized. Next, each NAC is visited iteratively by the pattern of the `for each NAC <<loop>>` state. The `<<NAC>>` stereotype is immediately removed from the matched package. For each NAC, the ID is incremented, making it unique for this NAC⁴. Next, in the `copy each context element` state, each context element is copied as a bound element into the NAC pattern. For each context element, the associations with the NAC are also moved to the NAC package, and the stereotypes and tags are copied as well.⁵

Then, the transformation class is looked up in the `find main class` state. In the `add method` state, a method representing the NAC is created in this class. In the `create nac flow` state, the Story Diagram that will represent the implementation of the NAC method is created. Next, it is checked if there was already a constraint on the state referencing the pattern in question. If no tag is found, a constraint tag is created in the `add constraint tag` state. Then, the method call is added to the constraint value, preceded with logical AND ("`&&`") if necessary. In the `add params` state, each bound element of the NAC pattern is looked up and added to the method declaration, and in the `add param in constraint tag` state they are appended to the call string of the constraint. When all parameters are added, the parameter list of the method call is closed by simply appending `)`.

It turns out that NACs can be easily implemented with a HOT. The higher order transformations for both nondeterministic scheduling and NACs clearly show that many language extensions can be implemented using this technique. Our experience shows that once the first language extension has been implemented using HOTs, the implementations of the following HOTs come quite easy, as many insights can be taken along to new implementations.

⁴This ID is used in the method name, hence the `"_2"` in the name of the `Transform_classes_to_tables_2` method of the example above. Apparently, it was the second NAC that was transformed.

⁵Alternatively, copying an element can be done with the copy operator (Van Gorp08b). The associations, tags and stereotypes will then be copied automatically.

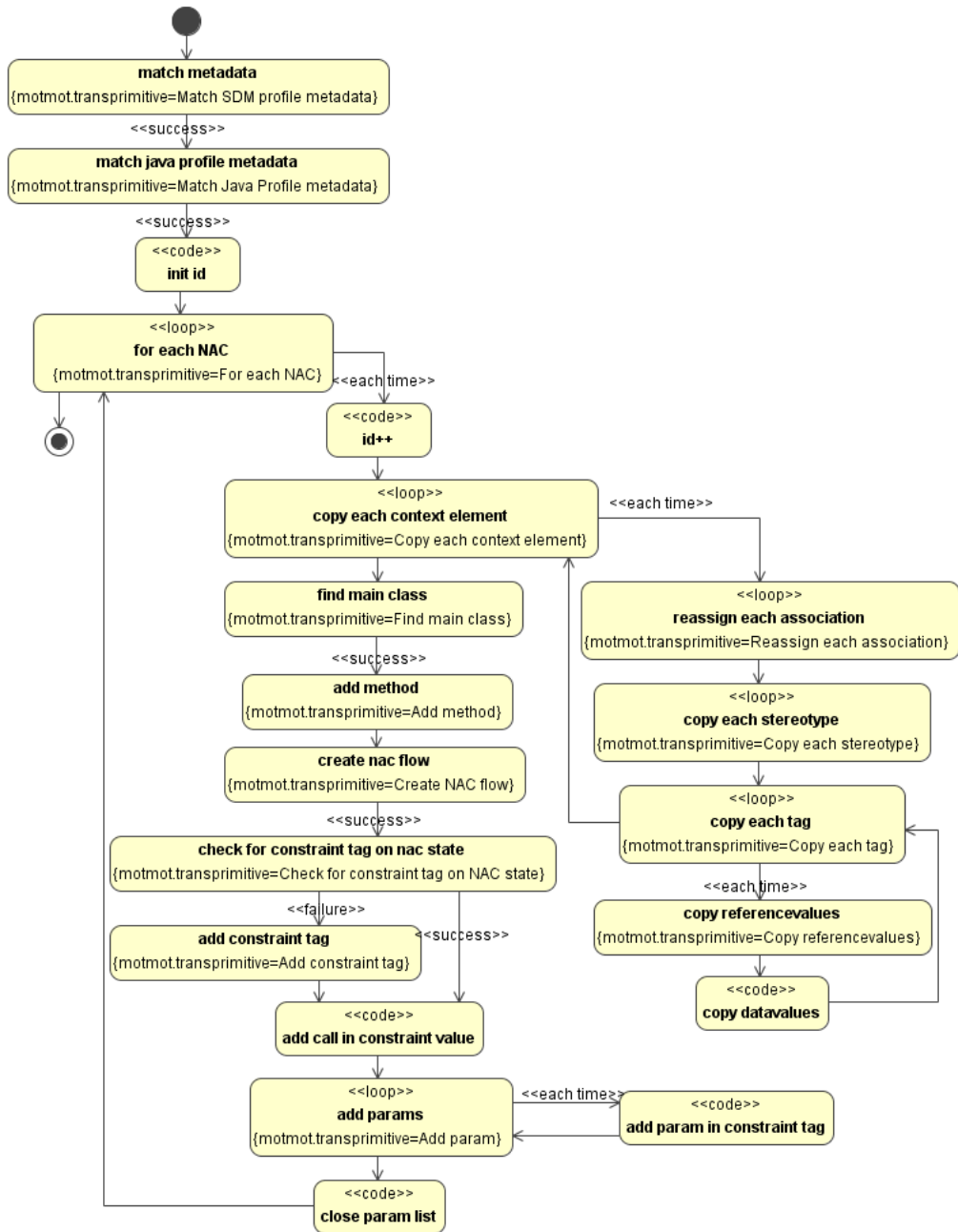


Figure 6.10: The Story Diagram of the NAC HOT

7.1 Related work

This thesis covers the two possible ways of handling rule scheduling: implicit and explicit. Blostein et al. mention a third one: event-driven graph rewriting (Blostein96). As explicit rule scheduling (or ordered graph rewriting) has an internally-imposed ordering of rules, event-driven graph rewriting has an externally-imposed ordering. The order is determined by interactions with the system, either by another program or a user. As said, AToM³ allows users during transformation execution to manually choose the subgraph to which the current pattern is applied. This is however not a case of an event-driven graph rewriting system, as it is the subgraph that is externally chosen, while the rule is still chosen by AToM³.

In GReAT (Balasubramanian02), rules can be scheduled explicitly using sequences and conditionals. Rules can also be connected in parallel, in which case the order of rule execution is nondeterministic (Balasubramanian06). GReAT does not offer a construct for priorities. However, GReAT relies on its own scheduling language, thus crippling the capabilities for portability of transformation models and transformation language features. On the other hand, its pattern specification language is UML compliant.

Syriani and Vangheluwe rely on DEVS (Bernard00) instead of relying on Activity Diagrams as a standard language for controlling the application of rewrite rules (Syriani07). Their MoTif implementation of DEVS in AToM³ offers excellent syntax to allow both implicit and explicit rule scheduling.

7.2 Future work

In the ND-SDM implementation in Chapter 4, we proposed a solution using higher order transformations (HOTs) that involved transforming the ND-SDM constructs to equivalents written in the imperative language known to MoTMoT. The current realization of nondeterministic scheduling aims at true nondeterminism by using random numbers. In some cases however, the order of the rules is irrelevant. More specifically, one may not care if the same order is used at all times. In these cases, the random number generator is nothing but a performance bottleneck. Therefore, the implementation can be extended with another, much simpler higher order transformation that imposes a particular order on the rules instead of guaranteeing randomness.

Alternatively, one could execute the rules in parallel. It seems intuitive to rely on UML *Fork* and *Join* elements to model the parallel nature of a transformation system explicitly. However, no SDM tool (Fujaba (Nickel00), MOFLON (Amelunxen06) or MoTMoT (Schippers04)) generates any code aimed at parallel execution, such as thread creation and thread synchronization. Therefore, a higher order transformation approach does not seem to be applicable. Instead, one would have to extend the core of a particular SDM tool. On the other hand, the use of standard UML elements does apply.

The technique of using higher order transformations to extend a language can be applied to many language extensions. The main idea is to come up with an equivalent expressed in the existing language. This has already been illustrated by the implementation of NACs in Chapter 6. As stated before in Section 6.1, NACs can also be applied to other types of elements besides parts of a pattern. These other types of NACs can also be implemented using HOTs. First, a NAC can be applied to a state. The equivalent would then be the same state without the `<<NAC>>` stereotype, but with the `<<success>>` and `<<failure>>` transitions swapped. Second, a NAC can be applied to an attribute of a node in a pattern. The equivalent would then use a *motmot.constraint* tag. Third, a NAC can be applied to an association between nodes. The equivalent would then be the same pattern, but with the `<<NAC>>` association removed, and an extra pattern will be introduced with a NAC on its

state containing the two (now bound) nodes of either end of the $\ll NAC \gg$ association. Note that in this way, previously implemented extensions are used in the implementation of a new extension.

Many other extensions can be implemented by HOTs. In Fujaba for example, *multi objects* are available. When a multi object is added to a pattern, the side-effects of this pattern must be executed for each occurrence of the multi object. *Maybe clauses* are also available in Fujaba. It is especially interesting in tools supporting so-called injective matching, like Fujaba. This means that different nodes in the pattern must match different objects in the input graph. MoTMoT is however not injective. An *Optional graph element* is another feature in Fujaba. With an optional node in a pattern, an exceptional possibility can be modeled in the same pattern as the expected scenario. For example, a stack push on a non-empty and on an empty stack can be modeled concisely in one pattern (Zündorf02). All these features can be implemented using the technique explained in this thesis.

Extensions for reuse can be implemented using HOTs. For example, a hierarchy can be established between patterns by adding inheritance to patterns. A pattern can inherit the behavior from its parent pattern. This would be useful when for instance adding a parent state for the rank patterns of the poker simulation, which are very similar (see Section 5.3.2), or when adding a parent state for the attribute transformation patterns in the UML to RDB transformation (see Section 6.1). In the equivalent of this feature, the elements of the parent patterns would be added to their child patterns, replacing duplicate elements. It would be useful if these parent patterns can be parameterized. Also, extensions for performance can be created using HOTs. For example, by splitting large patterns, the search space can be greatly reduced, as its size is exponential to the number of nodes in the pattern. This was done manually in some of the patterns of the poker example.

A very interesting extension is a traceability mechanism. Although this is implemented in many other tools, each tool has developed its own implementation. Therefore, it would be very useful to develop a tool-independent implementation of traceability. This can not be done using HOTs however, but the profile for generic containers and operations (as proposed in Section 4.4.7) can be used.

This brings us to the overall objective in the long-term. An ultimate profile for SDM can be developed, which can enable portability of models and especially language features. It turns out that many transformation tools share the same features (for example: NACs, traceability mechanisms, etc.). When this profile is implemented in these tools, tool builders can reuse implementations of others instead of building yet another solution for the same

problem. This ultimate profile can be divided into four profiles:

- a profile for the UML;
- a profile for a small selection of SDM operations usable as UML *Stereotypes* or *TagDefinitions*;
- a profile for generic containers and operations;
- an optional profile for new language constructs.

The idea for the first two profiles has already been introduced in (Schippers04) respectively (Van Gorp08b). They are used in MoTMoT. For true platform independence, a third profile is necessary in order to have access to some "helper" features, as explained in Section 4.4.7. Then, using the three previous profiles and higher order transformations, a fourth profile is available that is continuously extended with useful language constructs by the SDM community. It is the choice of the tool developer to make none, some or all of these language constructs available in his tool. Constructs of the fourth profile (like nondeterministic states or NACs) can be automatically shared between UML implementations, because the profiles ensure platform independence.

In order to implement traceability, the third profile can be used as follows. Generic nodes, links or annotations are used as language constructs for traceability. In Story Patterns, these nodes are typically used as UML *Classes*, links as UML *Associations*, and annotations as *Stereotypes* or *TagDefinitions*. The name is then of the form `traceabilityNode: GenericNode`, similar to how the UML profile is used in MoTMoT.¹ With this implementation of traceability, for example Triple Graph Grammars (Schürr95) and bi-directional transformations become possible.

When traceability becomes available, some of the elements of the profile for SDM can be eliminated and moved to the fourth profile, because they can now be implemented using higher order transformations. This makes the profile for SDM even more concise, making it more attractive to implement mappings to languages than Java. First, an equivalent of the *bound* annotation can be created using traceability constructs. A traceability link can be added between each bound element and its original. Second, *loops* on states and *each time* transitions can also be implemented using a HOT. The equivalent uses traceability annotations to mark the elements that have already matched, and a constraint is added to the state saying that only elements that are not annotated can match.

¹Such a *GenericNode* must roughly be able to reference elements, so a wrapper around a UML *Classifier* can be used as its implementation.

This implementation for *loops* has a very useful application in the immediate context of this thesis. In a nondeterministic state with a `<<loop>>` stereotype, the enumerating behavior of `<<loop>>` states can now be maintained (see footnote on page 26). A variant can be built that uses the traceability constructs in the same way as described above. The HOT implementing nondeterministic scheduling does not even have to be changed.

Extension	SDM equivalent
NAC on state	Swap <code><<success>></code> and <code><<failure>></code> transitions.
NAC on attribute	Use <i>motmot.constraint</i> .
NAC on association	Remove the NAC association from the pattern and add a second pattern with a NAC on its state containing the two (now bound) nodes on either end of the NAC association.
Multi objects	Use an additional <code><<loop>></code> state with a pattern containing the multi object node and the nodes it is directly connected to. Using an <code><<each time>></code> transition, the side-effects are executed for each iteration (Zündorf02).
Maybe clauses	Replace the pattern with a pattern for each possibility. These possibilities can be analyzed from parsing the maybe clause (Zündorf02).
Optional graph element	Replace the pattern with a pattern for each possibility (Zündorf02).
Inheritance	Add the elements of the parent patterns to their child patterns, and replace duplicate elements in the child pattern.
Reduce search space	Split large patterns.
Traceability	Annotate the model with traceability elements from the profile for generic containers and operations.
Bound nodes	Add a traceability link between each bound element and its original.
Loop states	Mark the elements that match and add a constraint saying that only elements that are not marked can match.

Table 7.1: Possible language extensions for SDM

All these extensions and a limited explanation of their equivalents are given in Table 7.1 as the summary of future work with higher order transformations. For some extensions, the profile for generic containers and operations has to be developed first. The proposed methodology of the four profiles will result in a tool-independent, highly modular and rich transformation language, with lots of plug-in possibilities.

A note must be made by presenting this methodology. Implementing features using higher order transformation comes with a performance cost. Performance is however very

important in transformation tools (Mohagheghi07). After all, the eventual goal of MDE tools is that they perform transformations on-the-fly between the different views of a system. In this context, it is still useful to optimize a transformation tool at the cost of modularity and portability.

7.3 Conclusion

In most transformation tools, transformation rules are scheduled either explicitly or implicitly. As a consequence, the user of the tool is forced to make a permanent choice between these scheduling paradigms. This means that in some occasions the user can't use the most intuitive scheduling mechanism.

The solution to this problem was given in this thesis as ND-SDM, a transformation language that supports both implicit and explicit - thus hybrid - scheduling. ND-SDM is an extension of SDM Story Diagrams, which already supports explicit rule scheduling. In other words, ND-SDM adds implicit rule scheduling to SDM, including nondeterministic scheduling, priorities and layers. It turned out that there is no clear-cut meaning of "nondeterministic scheduling of some rules". Therefore, the many variants have been analyzed.

The implementation proposed in this thesis is characterized by the use of two techniques, higher order transformations and prototypes. By using higher order transformations, SDM can be extended, without touching the SDM tool itself. In addition, in the case of MoTMoT, the implementation remains compliant to the UML profile, which is a major advantage of MoTMoT. By using prototypes, variants of nondeterministic scheduling can be plugged in without changing the implementation (i.e. the HOT), or even without knowing what a HOT is. The new language of ND-SDM is backed up by three examples, illustrating its various possibilities.

In addition, it is shown that higher order transformations can be used to implement many other language extensions, such as negative application conditions. With this knowledge, this thesis is an outset for a highly modular, tool-independent transformation language based on Story Driven Modeling.

Auxiliary diagrams

This chapter presents diagrams of the UML metamodel, based on the UML 1.4 specification (OMG01), and Story Patterns and code scripts of the higher order transformations for nondeterministic states and negative application conditions.

A.1 UML metamodel

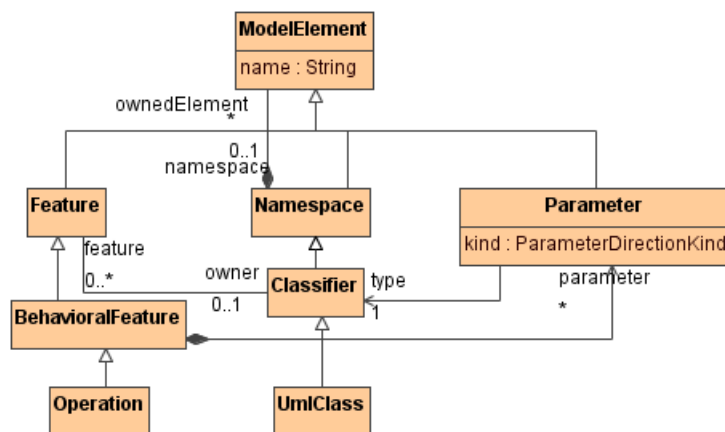


Figure A.1: UML core backbone (OMG01)

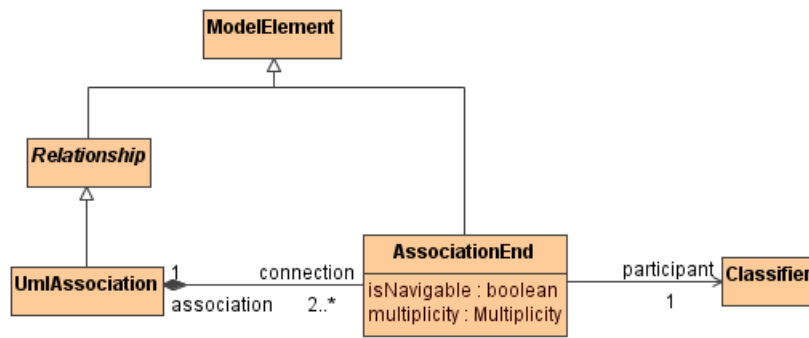


Figure A.2: UML core relationships (OMG01)

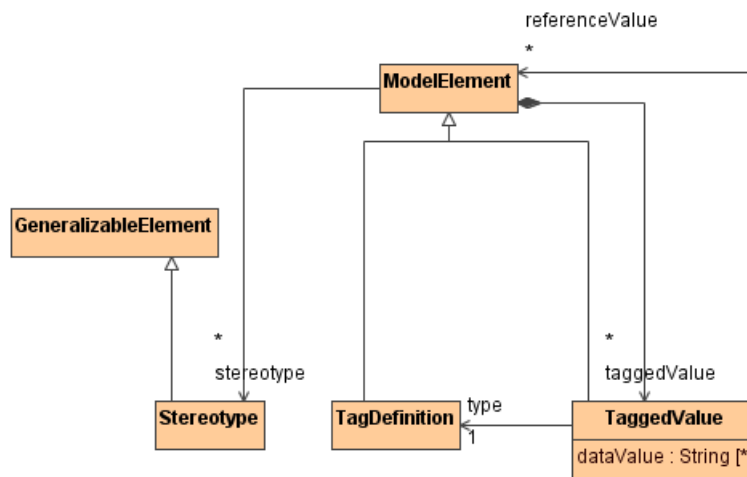


Figure A.3: UML core extension mechanisms (OMG01)

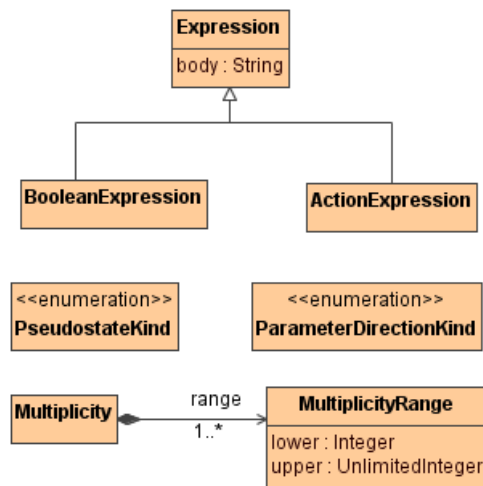


Figure A.4: UML data types (OMG01)

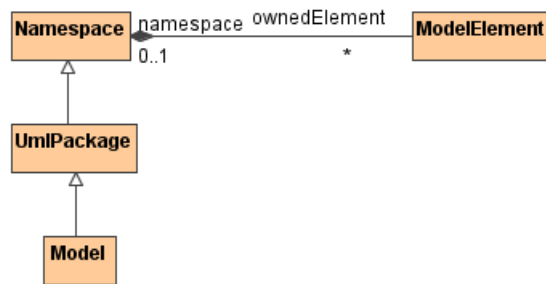


Figure A.5: UML model management (OMG01)

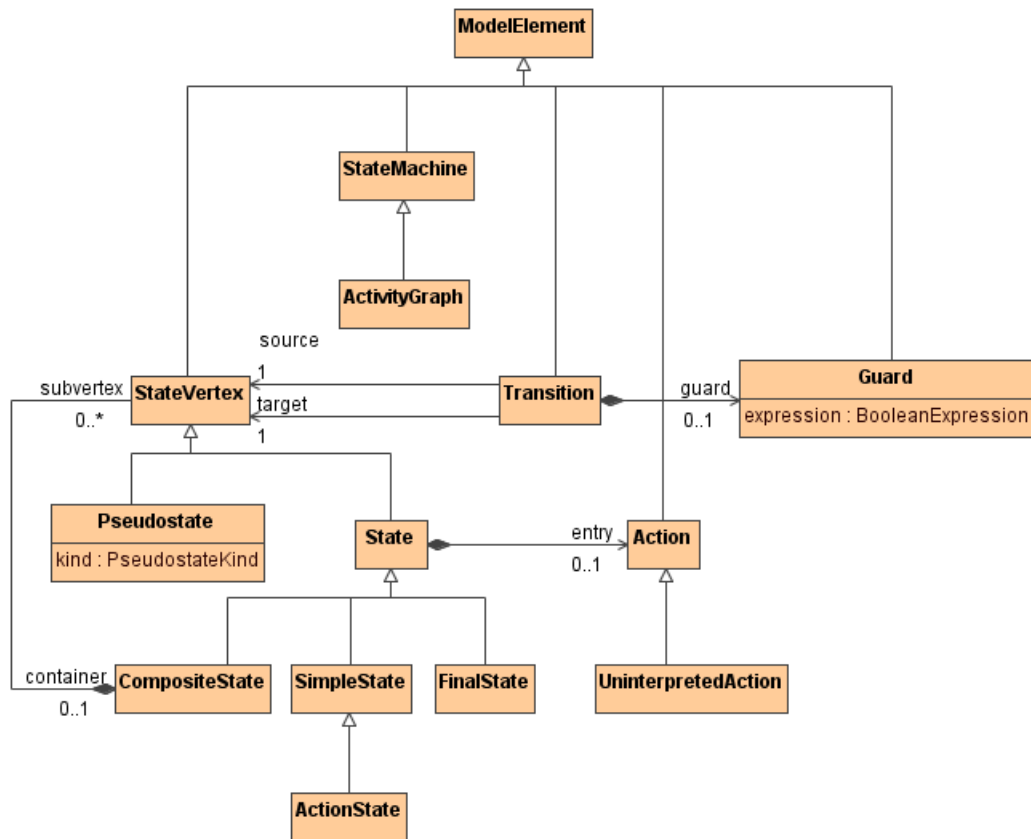


Figure A.6: UML Activity Diagrams (OMG01)

A.2 Story Patterns of the nondeterministic state HOT

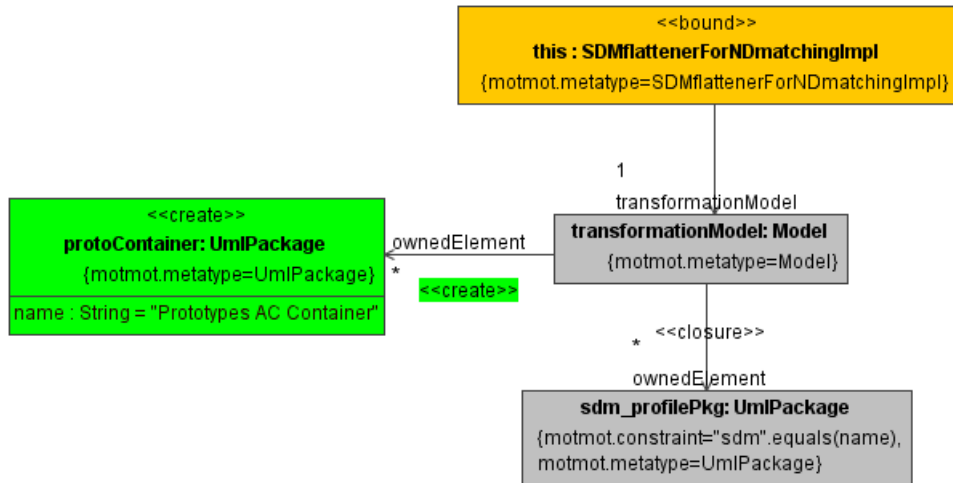


Figure A.7: Look up SDM profile package, create prototype application condition package

Only `this` is bound at the very start of the HOT, so the first pattern has to use this variable. The `Model` is looked up, which must have a package named `sdm` anywhere (denoted by the `<<closure>>` transition) in its hierarchy. If this pattern is matched, a new package is created into `transformationModel`.

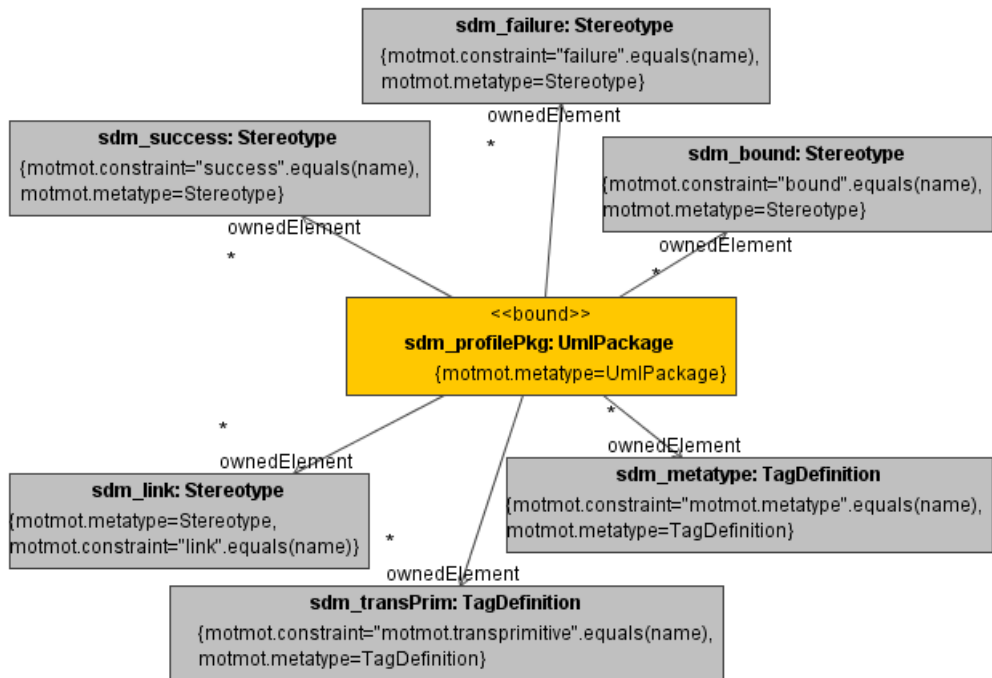


Figure A.8: Match SDM profile metadata

This pattern matches if `sdm_profilePkg` contains the given Stereotypes and Tag Definitions. These are part of the MoTMoT implementation of SDM, and are, in the context of this HOT, part of the input model's metamodel (i.e. the metamodel of the transformation model, in other words, the transformation language).

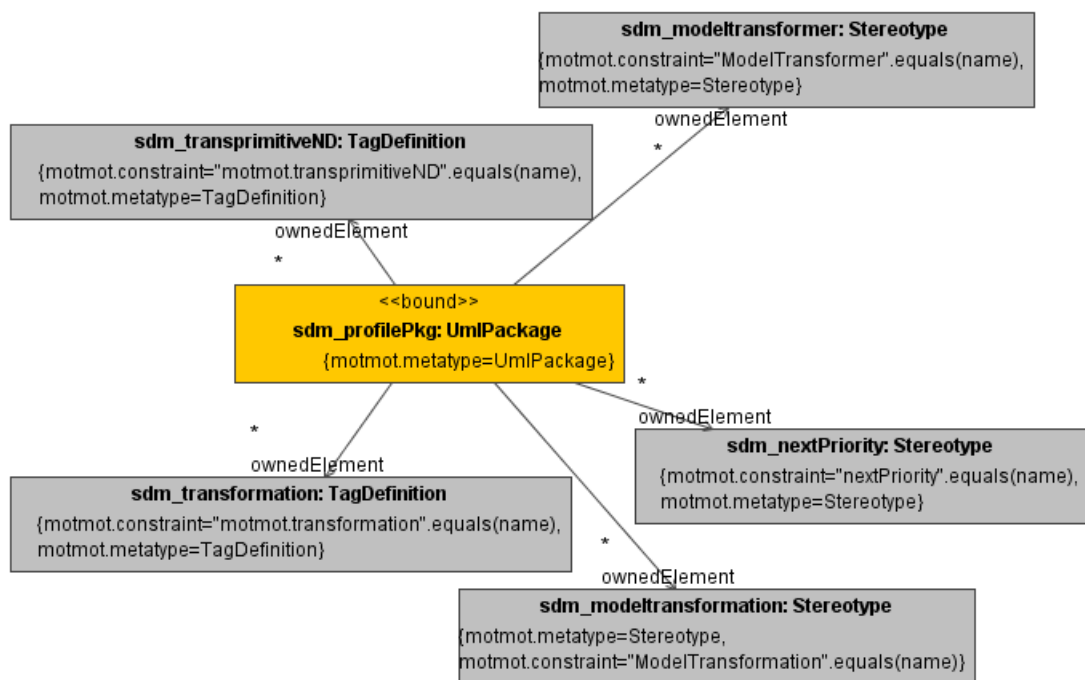


Figure A.9: Match SDM profile metadata (cont.)

This pattern matches if `sdm_profilePkg` contains the given Stereotypes and Tag Definitions. Some are part of the MoTMoT implementation of SDM, `sdm_transprimitiveND` and `sdm_nextPriority` are the new language constructs.

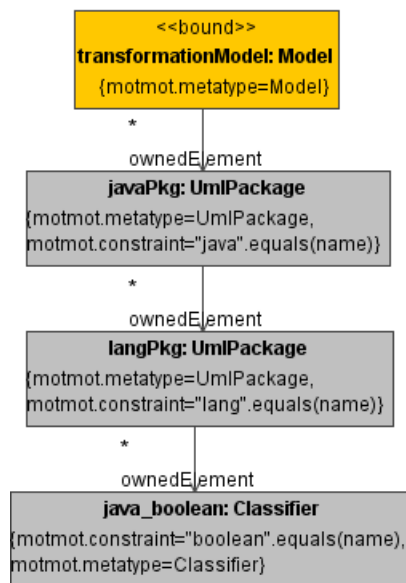


Figure A.10: Match Java profile metadata

Some Java language constructs are bound here. The reason why Java constructs are needed and how this relates to tool-independence is explained in Section 4.4.7.

```

String protoDir = "/prototypes/";
String absProtoDir = System.getProperty("user.dir")+"/target/classes" + protoDir;
java.io.FileNameFilter filter = new java.io.FileNameFilter() {
    public boolean accept(java.io.File dir, String name) {
        return name.endsWith(".xml");
    }
};
java.io.File[] files = new java.io.File(absProtoDir).listFiles(filter);
int nFiles = files.length;
int it = -1;
  
```

Figure A.11: List possible prototypes

The prototype files, which are .xml files residing in the prototypes directory, are listed in an array. A counter `it` is initialized.


```
it++;
```

Figure A.12: Increase counter

The counter is increased. This counter is used for iterating over each file, found in `list` possible prototypes (Figure A.11).

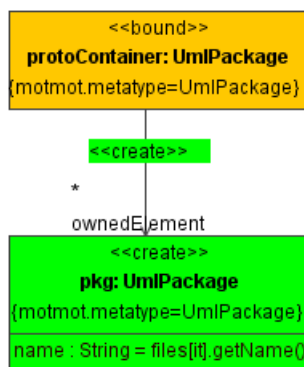


Figure A.13: Add container

Each application condition will be stored in its own package, which is a subpackage of `protoPackage`.

```

String modelUrl = "/prototypes/" + pkg.getName();
String graphName = "AC";
org.netbeans.api.mdr.MDRRepository repo
    = org.netbeans.api.mdr.MDRManager.getDefault().getDefaultRepository();
java.net.URL model_url = SDMflattenerForNDmatching.class.getResource(modelUrl);
java.net.URL metamodel_url
    = SDMflattenerForNDmatching.class.getResource("/M2_DiagramInterchangeModel.xml");
org.omg.uml.UmlPackage prototypeModel = null;
try {
    prototypeModel = (org.omg.uml.UmlPackage)
        be.ac.ua.motmot.lib.MotMotUtil.loadPrimitiveExtentFromXML(
            repo, model_url.toExternalForm(), model_url, metamodel_url,
            "UML", Class.forName("org.omg.uml.UmlPackage"), pkg.refOutermostPackage());
} catch (Exception e) logger.error("reading prototype failed: " + e.getMessage());
java.util.Collection allGraphs
    = prototypeModel.getActivityGraphs().getActivityGraph().refAllOfClass();
Iterator graphIt = null;
ActivityGraph ac = null;
boolean found = false;
boolean deleting = true;
while (deleting) {
    deleting = false;
    graphIt = allGraphs.iterator();
    while (!deleting && graphIt.hasNext()) {
        ActivityGraph aGraph = (ActivityGraph) graphIt.next();
        if (aGraph.getName().startsWith(graphName)) {
            ac = aGraph;
            ac.setName(modelUrl + "_" + ac.getName());
            found = true;
        } else {
            Namespace ns = aGraph.getNamespace();
            while (ns.getNamespace() != null) ns = ns.getNamespace();
            if (!aGraph.getName().startsWith("/prototypes/")
                && !ns.refMofId().equals(transformationModel.refMofId())) {
                aGraph.refDelete();
                deleting = true;
            }
        }
    }
}
}
}
}

```

Figure A.14: Read application condition

The application condition is read. First, a prototype model is loaded into the working repository `repo`. Then, the application condition diagram is searched for. However, other Activity Diagrams from the model (such as the actual prototype), must be removed from `repo`. After all, the actual prototype must be read again for every nondeterministic state that has to be transformed, as states and transitions are directly extracted from the prototype diagram, rendering it unusable for a next transformation of a nondeterministic state. If these diagrams are kept, name clashes occur.

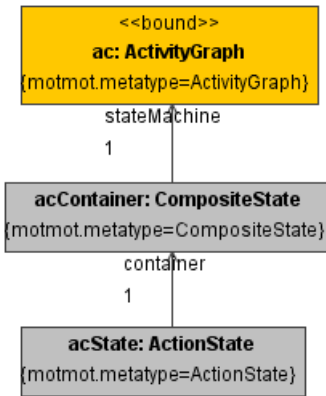


Figure A.15: Get application condition state

The only state of the application condition diagram is bound here.

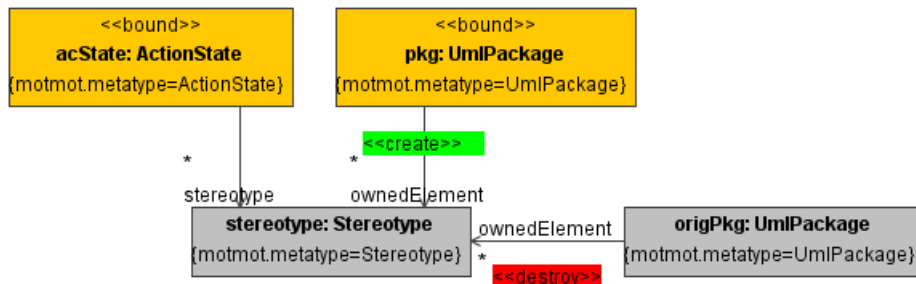


Figure A.16: Get application condition stereotypes

A stereotype on the application condition state is moved to protoPackage, the package that will contain the application condition.

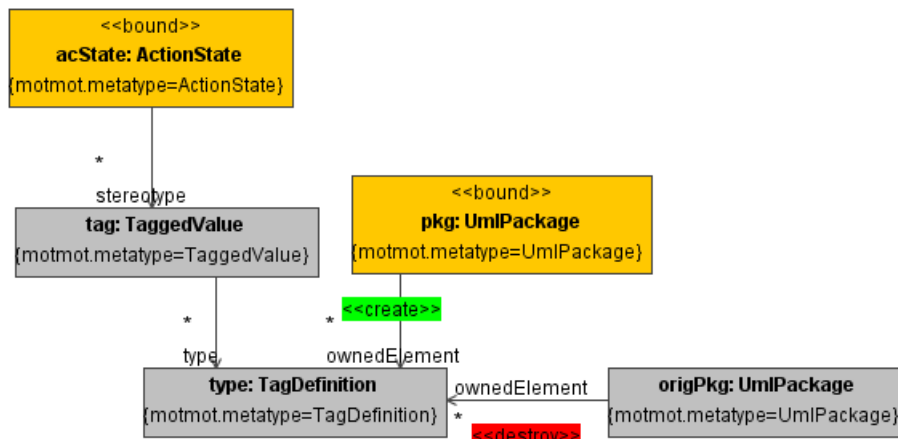


Figure A.17: Get application condition tag definitions

A tag definition of a tag on the application condition state is moved to `protoPackage`, the package that will contain the application condition.

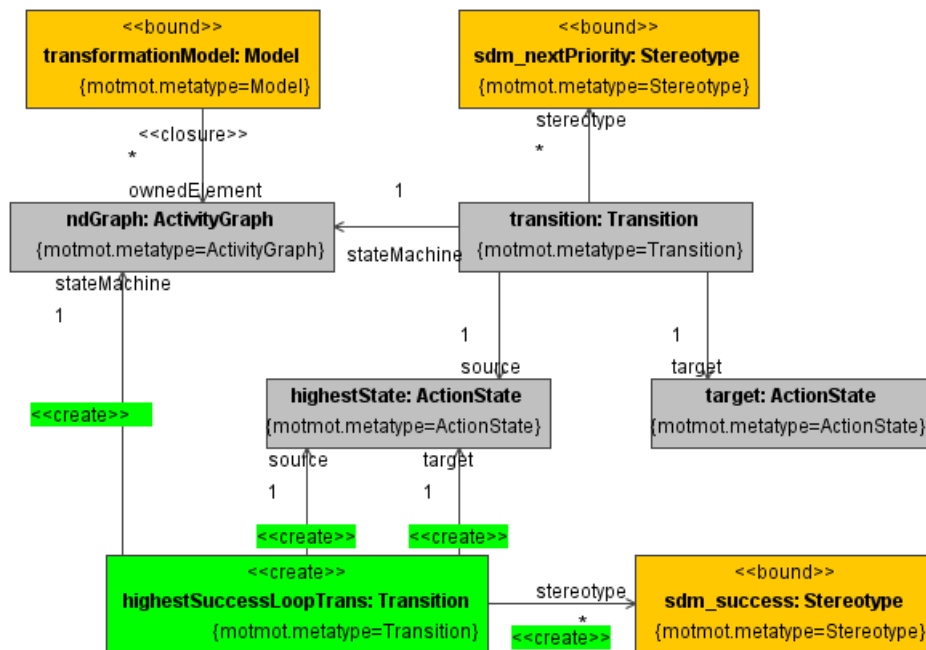


Figure A.18: Match a `<<nextPriority>>` transition and create a `<<success>>` loop

A `<<nextPriority>>` transition and a `<<success>>` transition is created as the first step to transform a `<<nextPriority>>` transition chain (described in Figure 3.9). As shown in the Story Diagram of Figure 4.13, there is an additional constraint on matching this pattern.

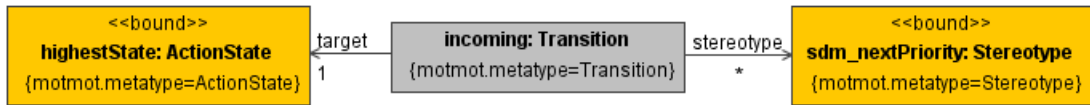


Figure A.19: Check whether source has an incoming `<<nextPriority>>` transition

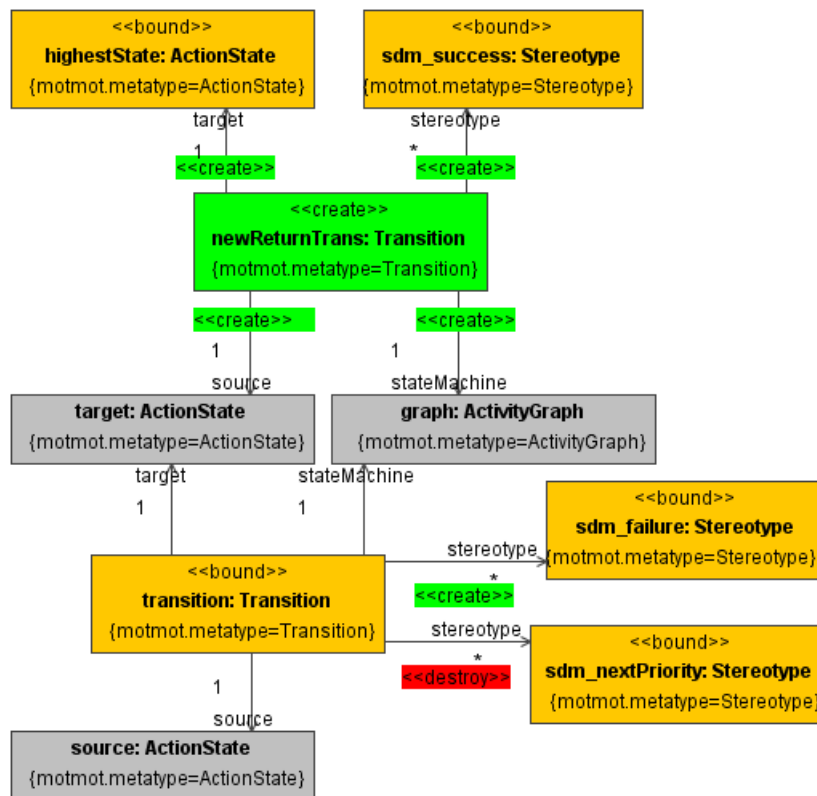


Figure A.20: Transform a `<<nextPriority>>` transition

This pattern transforms a `<<nextPriority>>` transition in a `<<success>>` and `<<failure>>` transition as proposed in Figure 3.9. A new `<<success>>` transition is created with the current state target as source, and the state with top priority as target. Also, the `<<nextPriority>>` transition becomes a `<<failure>>` transition.

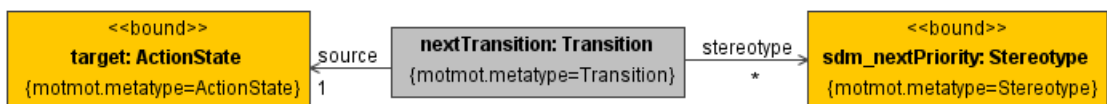


Figure A.21: Find the next `<<nextPriority>>` transition

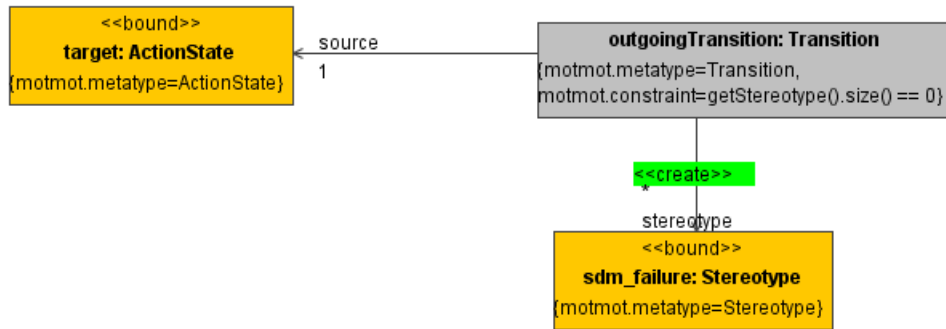


Figure A.22: Set a *failure* stereotype on the outgoing transition

According to Figure 3.9, the outgoing transition of a *nextPriority* transition chain must have a *failure* stereotype.

```
int int_id = 0;
```

Figure A.23: Initialize ID

An ID is initialized. This ID will make sure that the names of the new states created during the transformation are unique.

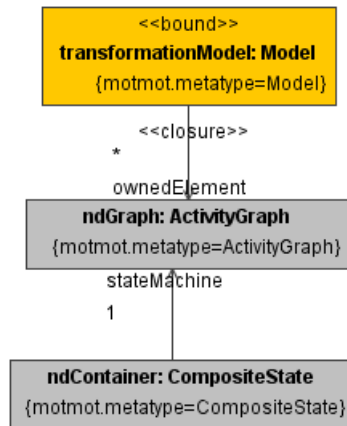


Figure A.24: Find the next Activity Diagram

An Activity Diagram `ndGraph` is looked up, and its state container is bound.

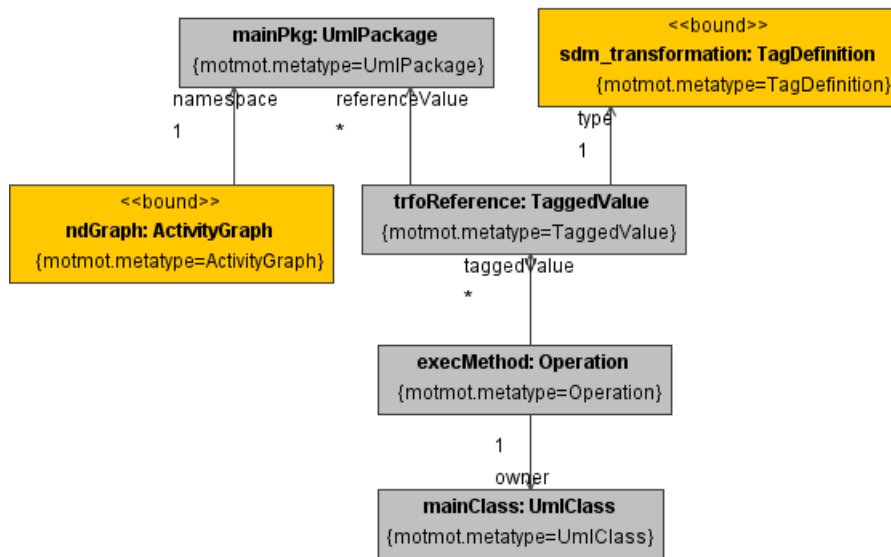


Figure A.25: Check whether the graph is called

It is checked whether `ndGraph` is called. According to MoTMoT, this is done by a method, annotated with a `motmot.transformation` tag with as value the package where the graph resides.

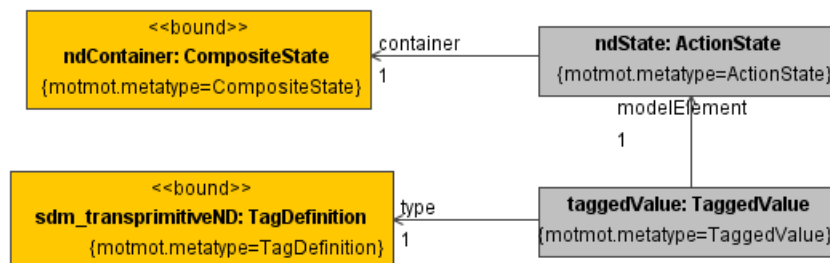


Figure A.26: Find a nondeterministic state

A nondeterministic state is looked up in the state container `ndContainer` of `ndGraph`. A nondeterministic state is simply a state with a `motmot.transprimitiveND` tag.

```
int_id++;
```

Figure A.27: Increment ID for each nondeterministic state

```
boolean found = false;
```

Figure A.28: Initialize a variable denoting if the prototype is found

If the application condition of the prototype is matched, the prototype is found, and this variable will be set to *true*.

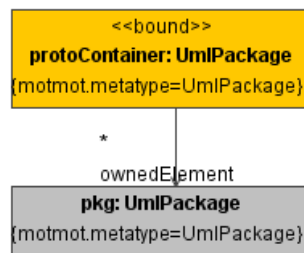


Figure A.29: Find a previously loaded application condition

One of the previously read prototype application conditions is chosen by binding its package *pkg*.

```
boolean nomatch = false;
```

Figure A.30: Initialize a variable denoting if the current prototype is invalid

If the application condition of the currently checked prototype is not matched, this variable will be set to *true*.

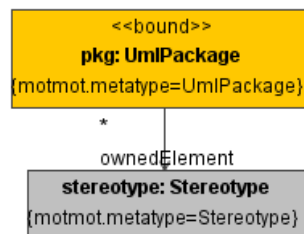


Figure A.31: Iterate over each stereotype of the application condition

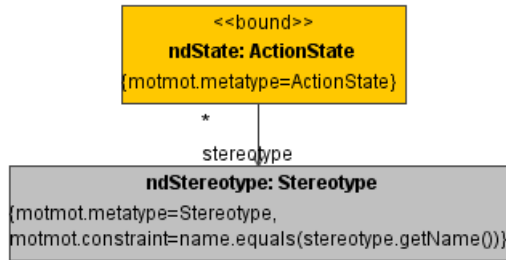


Figure A.32: Match the stereotype of the application condition

`nomatch = true;`

Figure A.33: The current prototype is flagged invalid

`boolean nomatch = false;`

Figure A.34: Re-initialize a variable denoting if the current prototype is invalid

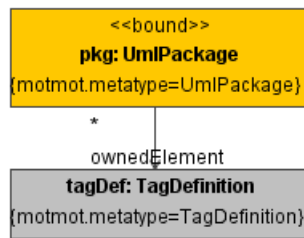


Figure A.35: Iterate over each tag definition of the application condition

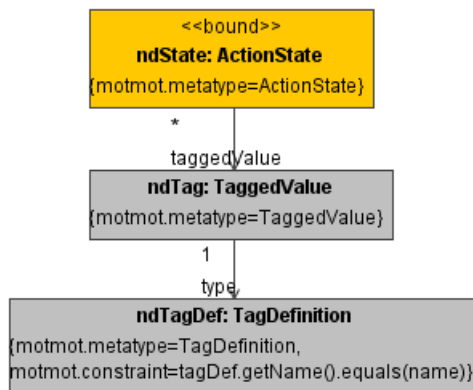


Figure A.36: Match the tag definition of the application condition

`found = true;`

Figure A.37: The current prototype is found

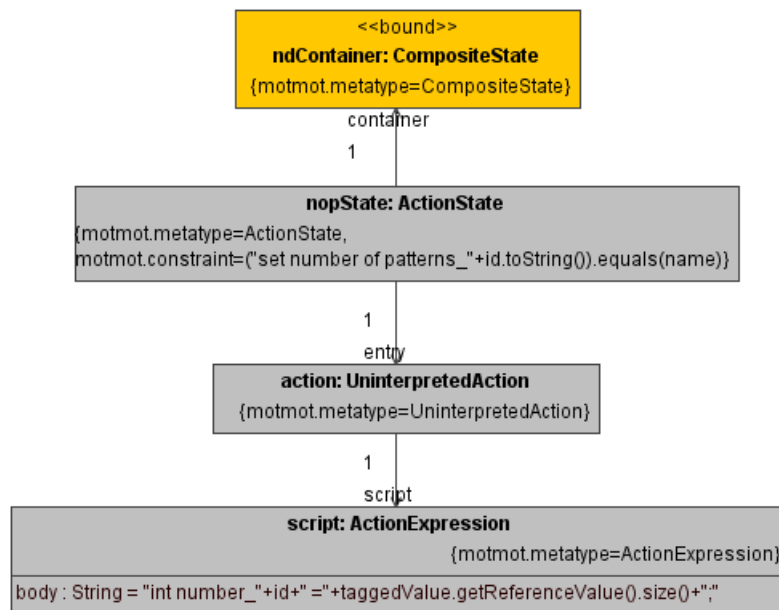


Figure A.38: Set the number of patterns

The actual number of Story Patterns or Story Diagrams is set in the code script of the `set number of patterns` state. This could not be done in the prototype model itself, because the actual number is of course unknown then. Note that the name of `nopState` is a unique name, caused by the ID.

```
int it = 0;
```

Figure A.39: Initialize `it` as a counter that will give a unique ID to each package referenced by the nondeterministic state

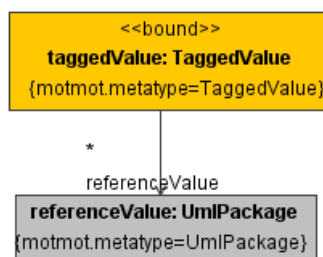


Figure A.40: Look up a package referenced by the nondeterministic state

```
it++ = 0;
```

Figure A.41: Increase `it` in order to have a different number for each referenced package

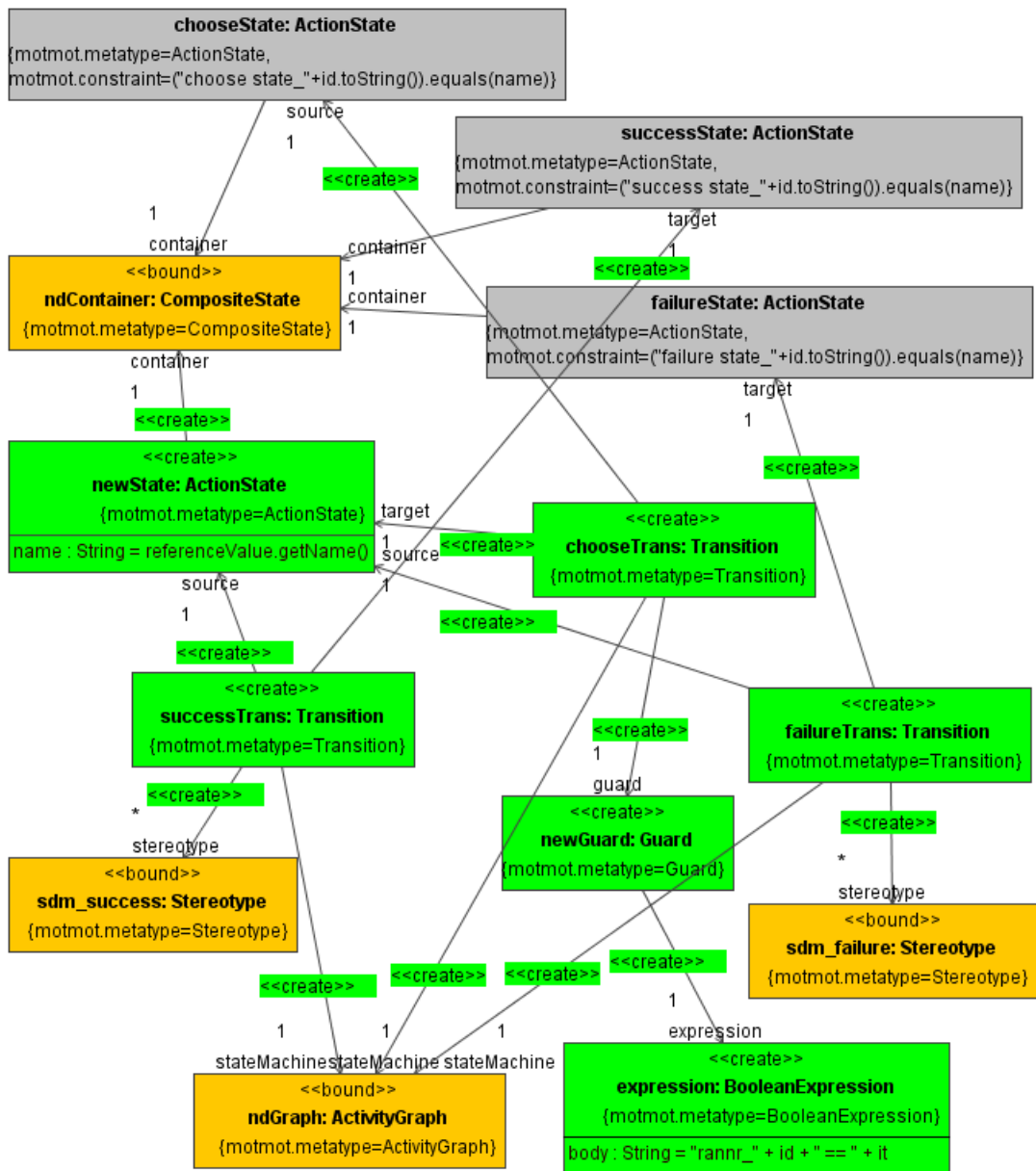


Figure A.42: Create a state and its corresponding transitions

A state `newState` which will reference one of the packages, is created together with its corresponding transitions. The aimed result is similar to one of the *motmot.transprimitive* states in the middle of the diagram of Figure 4.10. The transitions are created with their corresponding stereotypes or guards. `it` is used in the boolean expression of the guard `newGuard` as the number for which this state is chosen by the `choose state`. For now, `newState` only gets a name.

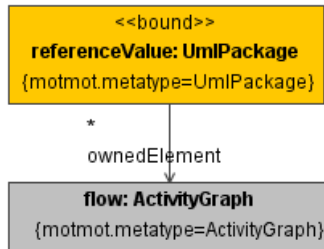


Figure A.43: Check whether the currently bound package contains an Activity Diagram

The package `referenceValue` can contain a Story Diagram or a Story Pattern. In the case of a Story Diagram, a method call must be added to the new state. Otherwise, a `motmot.transprimitive` tag must be added.

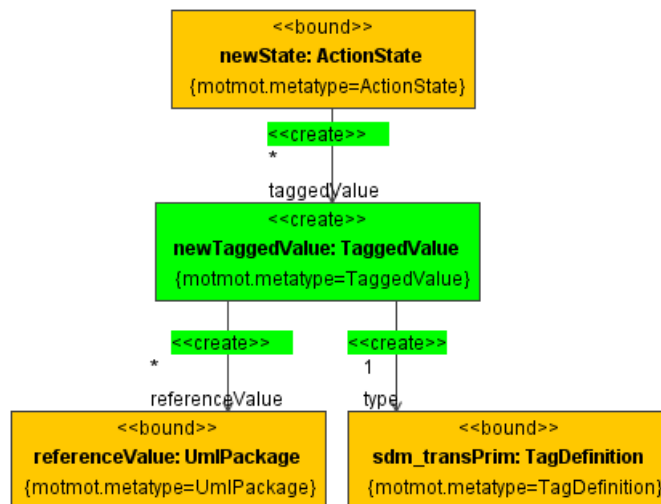


Figure A.44: Create a `motmot.transprimitive` tag on the state

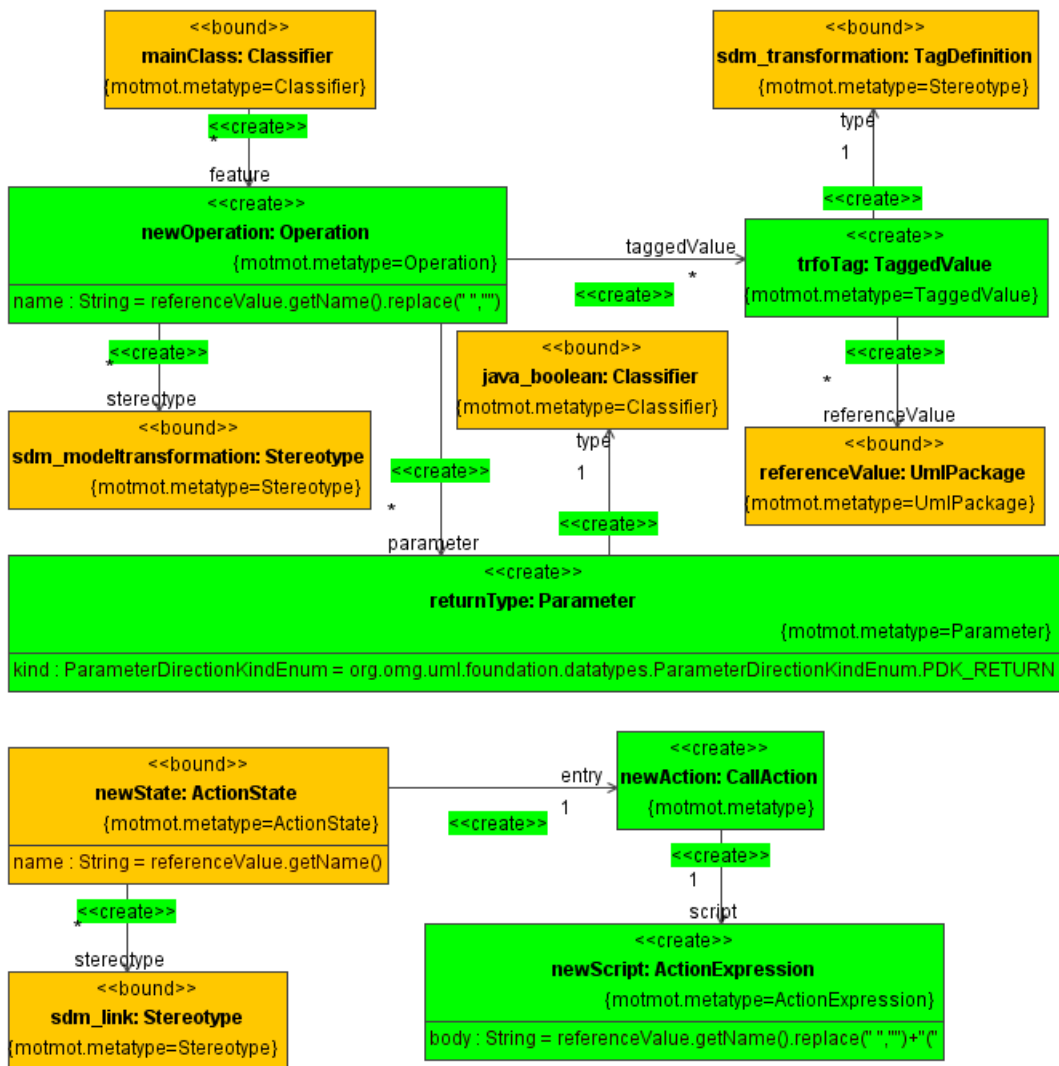


Figure A.45: Create a method and a call

A method `newOperation` is added to the execution class `mainClass`. To mark the method as a method that calls a Story Diagram, it is annotated with a `<<ModelTransformation>>` stereotype and a `motmot.transformation` tag pointing to the package in which the Story Diagram resides. The method will be named after the package. On top of that, a return type is added as a `boolean`. The call to the method is created as Java code on `newState`, which becomes a `<<link>>` state.

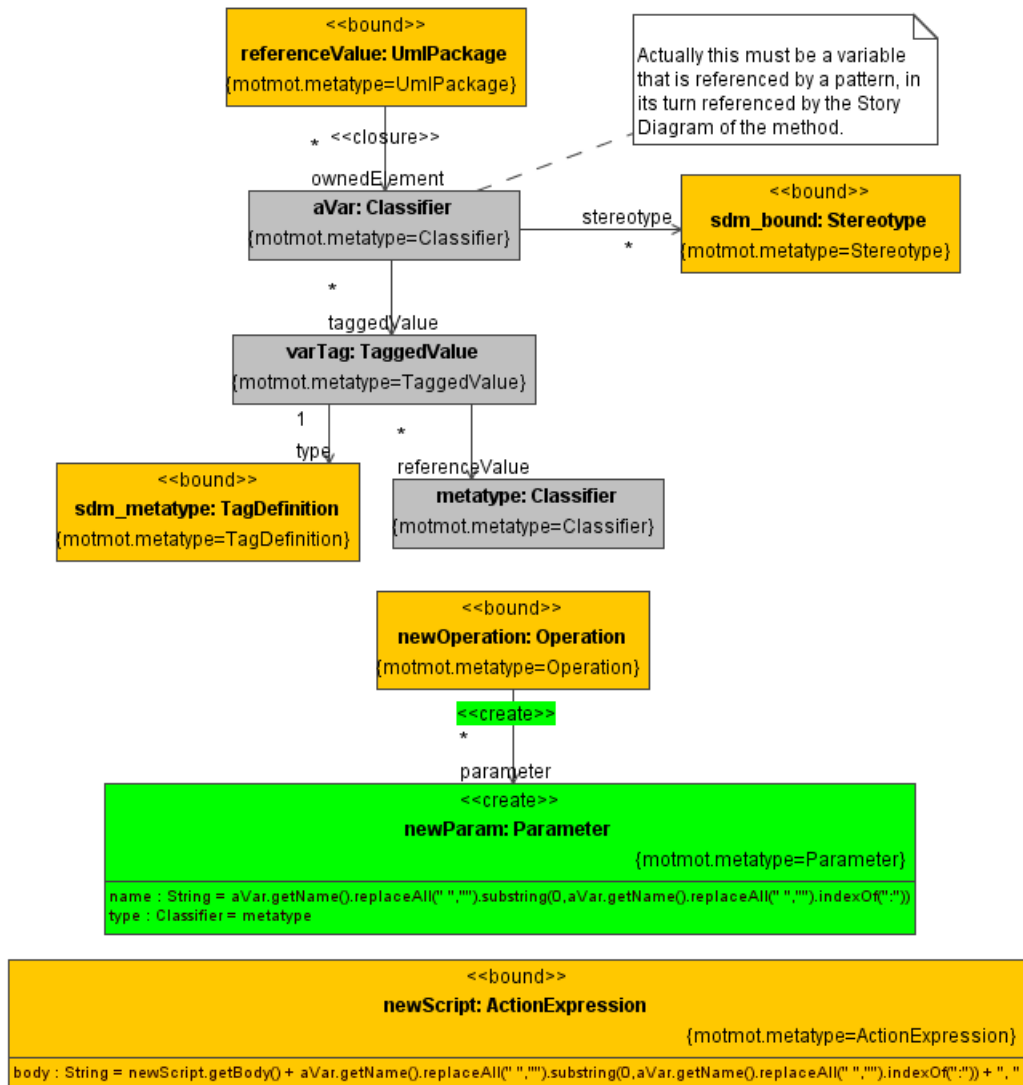


Figure A.46: Add a parameter to the method and its call

A bound variable aVar is looked up, together with its metatype. aVar is then added as a parameter with type metatype to both the method newOperation itself and the call in newScript.

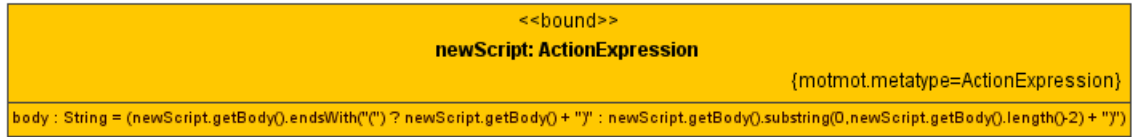


Figure A.47: Close the method call parameter list

The method call Java code string is terminated with a right parenthesis after all the necessary parameters have been added to the call.

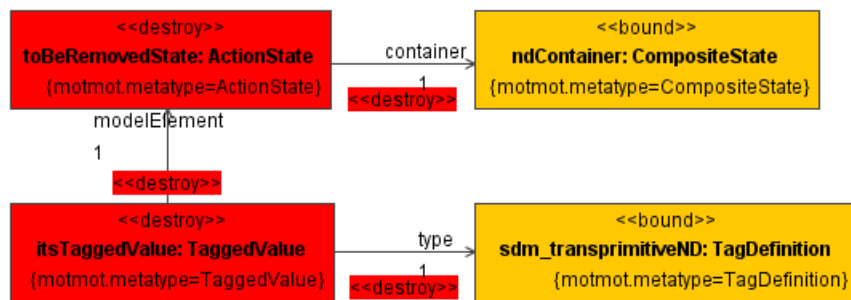


Figure A.48: Remove a nondeterministic state

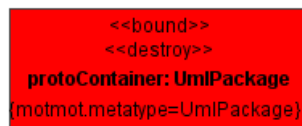


Figure A.49: Remove the package containing the prototype application conditions

Bibliography

- [Amelunxen06] Carsten Amelunxen, Alexander Königs, Tobias Röttschke, and Andy Schürr, *MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations*, Model Driven Architecture - Foundations and Applications: Second European Conference (Heidelberg) (Arend Rensink and Jos Warmer, eds.), Lecture Notes in Computer Science (LNCS), vol. 4066, Springer Verlag, 2006, pp. 361–375.
- [Amelunxen09] Carsten Amelunxen, Alexander Königs, Tobias Röttschke, and Andy Schürr, *MOFLON home*, <http://www.moflon.org>, 2009.
- [Balasubramanian02] Daniel Balasubramanian, Anantha Narayanan, Chris van Buskirk, and Gabor Karsai, *GReAT - graph rewriting and transformation*, <http://www.isis.vanderbilt.edu/tools/GReAT>, 2002.
- [Balasubramanian06] Daniel Balasubramanian, Anantha Narayanan, Chris van Buskirk, and Gabor Karsai, *The graph rewriting and transformation language: GReAT*, Electronic Communications of the EASST **1** (2006).
- [Bernard00] Zeigler Bernard, Herbert Praehofer, and Tag Gon Kim, *Theory of modeling and simulation*, Academic Press, January 2000.
- [Bézivin04] Jean Bézivin, *In search of a basic principle for model driven engineering*, CEPIS, UPGRADE, The European Journal for the Informatics Professional **V** (2004), no. 2, 21–24.
- [Bézivin05] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt, *Model transformations in practice workshop*, Satellite Events at the MoDELS 2005 Conference (Jean-Michel Bruel, ed.), LNCS, vol. 3844, 2005, pp. 120–127.
- [Blostein96] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec, *Issues in the practical use of graph rewriting*, 5th Workshop on Graph Grammars and Their Application To Computer Science, Lecture Notes in Computer Science, Springer, 1996, pp. 38–55.
- [Csertán02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap Zsigmond, András Pataricza, and Dániel Varró, *VIATRA - visual automated transformations for formal verification and validation of UML*

- models*, Proceedings of the 17th IEEE international conference on Automated software engineering (Washington, DC, USA), IEEE Computer Society, 2002, pp. 267–270.
- [De Lara00] Juan De Lara, Hans Vangheluwe, and Manuel Alfonseca, *Meta-modelling and graph grammars for multi-paradigm modelling in atom3*, Software and Systems Modeling (SoSyM) **3(3)** (2000), 194 – 209.
- [De Lara02] Juan De Lara and Hans Vangheluwe, *Atom3 - a tool for multi-formalism meta-modelling*, <http://atom3.cs.mcgill.ca/>, 2002.
- [Fujaba07] Fujaba, *The fujaba home page*, <http://www.fujaba.de/>, 2007.
- [Gerber03] Anna Gerber and Kerry Raymond, *Mof to emf: there and back again*, Eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange (New York, NY, USA), ACM, 2003, pp. 60–64.
- [Habel95] Annegret Habel, Reiko Heckel, and Gabriele Täntzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae **26** (1995), 287–313.
- [Holt00] Richard C. Holt, Andreas Winter, and Andy Schürr, *GXL: Toward a standard exchange format*, 7th Working Conference on Reverse Engineering (WCRE 2000), 2000, pp. 162–171.
- [Kalnins08] Audris Kalnins, Janis Barzdins, and Karlis Podnieks, *Mola - model transformation language*, <http://mola.mii.lu.lv/>, 2008.
- [Kaplan89] Simon M. Kaplan and Steven K. Goering, *Priority controlled incremental attribute evaluation in attributed graph grammars*, TAPSOFT, Vol.1, 1989, pp. 306–336.
- [Kleppe03] Anneke Kleppe, Jos Warmer, and Wim Bast, *MDA explained: The model driven architecture—practice and promise*, Addison-Wesley Professional, April 2003.
- [Lambers08] Leen Lambers, Hartmut Ehrig, and Fernando Orejas, *Efficient conflict detection in graph transformation systems by essential critical pairs*, Electron. Notes Theor. Comput. Sci. **211** (2008), 17–26.
- [Levendovszky07] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Tamás Mészáros, *Vmts - visual modeling and transformation system*, <http://www.vmts.aut.bme.hu/>, 2007.
- [Mens06] Tom Mens and Pieter Van Gorp, *A taxonomy of model transformation*, Electronic Notes in Theoretical Computer Science **152** (2006), 125–142.
- [Meyers08] Bart Meyers and Pieter Van Gorp, *Towards a hybrid transformation language: Implicit and explicit rule scheduling in story diagrams*, Sixth International Fujaba Days, September 18–19 2008, pp. 15–18.
- [Mohagheghi07] Parastoo Mohagheghi and Vegard Dehlen, *An overview of quality frameworks in model-driven engineering and observations on transformation quality*, Proceedings of the 2nd Workshop on Quality in

- Modeling (Nashville, TN, USA) (Ludwik Kuzniarz, Jean Louis Sourrouille, and Mirosław Staron, eds.), 2007, pp. 3–17.
- [Muliawan08] Olaf Muliawan, *Extending a model transformation language using higher order transformations*, WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering (Washington, DC, USA), IEEE Computer Society, 2008, pp. 315–318.
- [Nickel00] Ulrich Nickel, Jörg Niere, and Albert Zündorf, *Tool demonstration: The fujaba environment*, 22nd International Conference on Software Engineering (ICSE), 2000, pp. 742–745.
- [No Magic09] No Magic, *Magicdraw uml*, 2009.
- [OMG00] Object Management Group OMG and R. Soley, *Model-driven architecture*, 2000.
- [OMG01] OMG, *OMG unified modeling language specification, Version 1.4*, June 2001.
- [OMG05] Object Management Group OMG and R. Soley, *XML metadata interchange (XMI) specification, Version 2.0*, 5 2005.
- [OMG06] OMG, *OMG object constraint language, Version 2.0*, May 2006.
- [Peterson77] James L. Peterson, *Petri nets*, ACM Computing Surveys **9** (1977), no. 3, 223–252.
- [Rekers97] Jan Rekers and Andy Schürr, *Defining and Parsing Visual Languages with Layered Graph Grammars*, Journal of Visual Languages and Computing **8** (1997), no. 1, 27–55.
- [Rozenberg97] Grzegorz Rozenberg (ed.), *Handbook of graph grammars and computing by graph transformations, volume 1: Foundations*, World Scientific, 1997.
- [Schippers04] Hans Schippers, Pieter Van Gorp, and Dirk Janssens, *Leveraging UML profiles to generate plugins from visual model transformations*, Electronic Notes in Theoretical Computer Science **127** (2004), no. 3, 5–16, Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.
- [Schürr95] Andy Schürr, *Specification of graph translators with triple graph grammars*, 20th International Workshop on Graph-Theoretic Concepts in Computer Science (London, UK), Springer-Verlag, 1995, pp. 151–163.
- [Schürr09a] Andy Schürr, *Progres - programmed graph rewriting systems*, <http://www.se.rwth-aachen.de/research/projects/progres>, 2009.
- [Schürr09b] Andy Schürr, *The PROGRES language manual version 9.x*, 2009.
- [Sendall03] Shane Sendall and Wojtek Kozaczynski, *Model transformation - the heart and soul of model-driven software development*, IEEE Software, Special Issue on Model Driven Software Development (2003), 42–45.

- [Syriani07] Eugene Syriani and Hans Vangheluwe, *Programmed graph rewriting with DEVS*, International Conference on Applications of Graph Transformations with Industrial Relevance (Kassel) (Manfred Nagl and Andy Schürr, eds.), Lecture Notes in Computer Science, Springer, 2007.
- [Tántzer97] Gabriele Tántzer and Olga Runge, *Agg - the attributed graph grammar system*, <http://tfs.cs.tu-berlin.de/agg/>, 1997.
- [Tántzer06] Gabriele Tántzer and Olga Runge, *The agg 1.4.0 development environment - the user manual*, 2006.
- [Van Gorp07] Pieter Van Gorp, Hans Schippers, and Olaf Muliawan, *Motmot*, <http://motmot.sourceforge.net/>, 2007.
- [Van Gorp08a] Pieter Van Gorp, *Model-driven development of model transformations*, Ph.D. thesis, University of Antwerp, 2008.
- [Van Gorp08b] Pieter Van Gorp, Anne Keller, and Dirk Janssens, *Transformation language integration based on profiles and higher order transformations*, 2008.
- [Varró04] Dániel Varró and András Pataricza, *Generic and meta-transformations for model transformation engineering*, Proc. UML 2004: 7th International Conference on the Unified Modeling Language (Lisbon, Portugal) (T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, eds.), LNCS, vol. 3273, Springer, October 10–15 2004, pp. 290–304.
- [Wikipedia09] Wikipedia, *Poker*, <http://en.wikipedia.org/wiki/Poker>, 2009.
- [Zündorf02] Albert Zündorf, *Rigorous object oriented software development*, Ph.D. thesis, University of Paderborn, 2002.