Model Patterns for Model Transformations in Model Driven Development

Markus Scheidgen
Institute of Computer Science, Humboldt-Universität zu Berlin
scheidge@informatik.hu-berlin.de

Abstract

Model driven development is a popular approach to master the complexity of computer based systems, but it is still missing well-established technologies for model transformations. A lot of research has been done to address this subject, most of it tends towards highly expressive and highly specialized transformation languages. This paper takes a contra point to this trend, proposing the transformation implementation language Mopa (Model Pattern), which is less expressive but provides more flexibility. Mopa is independent of the chosen modelling techniques, it allows the realization of different transformation approaches, and it is integrated into the Java programming language, hence easy to integrate into existing environments. Mopa is described with formal syntax and semantics, and this paper shows how to use Mopa to implement different existing transformation approaches.

1. Introduction

To master the complexity of computer based systems, engineers use abstractions to describe a system with models that are simpler than the system itself. Model driven development (MDD) is a system engineering approach that uses models as artifacts to develop a system. Different models and modelling languages are used to describe a system on different abstraction levels and from different views. The Model Driven Architecture (MDA) [17] is a popular way of model driven development, promoted by the OMG. It uses platform independent models (PIMs) to abstract from the details of specific platforms and recommends automated transformations from abstract PIMs to platform specific models (PSMs).

The problem with model driven development in general and MDA in particular is the matter of model transformations. While there are well-established technologies to specify and manage models, as well as their modelling languages (meta-models), there is still no widely used and accepted technology to either specify or implement relations between models in general and transformations between PIMs and PSMs in particular.

The subject has been addressed in numerous research papers. OMG's request for proposal *Query Views Transformations* (QVT) [4] led to the recommendation [20]. Gardner et al. [13] give a summary to previous QVT contributions; Czarnecki and Helsen [9] classify different transformation approaches. Other papers that deal with transformations especially in the context of MDA are [14, 5, 15].

Most of this work shows a trend towards special purpose transformation languages that inherently constrain their users to specific model transformation approaches. These are, for example, techniques that only allow transformations within a single model, only allow transformations from one modelling language into another, or only allow code generation from models. As Czarnecki and Helsen show in [9] MDA leads to different applications of model transformations. Each of them requires a different transformation approach, thus a different transformation language. Users of MDD/MDA have to master different transformation technologies, since every single technique is yet expressive, but also too constraining.

This paper proposes the transformation language Mopa that works against this trend. Mopa is less expressive than other technologies, but it provides more flexibility: It works for different transformation approaches; it is independent of the actual modelling technique; and it is integrated in the Java programming language. We use Mopa for our own MDD/MDA tools presented in [12, 11].

The following section 2 gives an exemplary introduction into what Mopa is and how it works; it is followed by a presentation of related work and a classification based on Mopa's model transformation features. Section 3 defines syntax and semantics of Mopa formally. In section 4 it is explained how Mopa can be applied to different approaches to model transformation. This section covers three different application scenarios that typically arise in model driven development: The execution of models, generation of code from models, and finally the transformation from models into models. The paper is closed by the concluding section 5.

2. Basics and Classification

After introducing Mopa, using some basic examples, we discuss where Mopa comes from and how it relates to other model transformation techniques. After that, Mopa will be classified based on its features, using Czarnecki and Helson's classification catalogue [9].

2.1. Introduction to Mopa

Mopa stands for Model Pattern; it is a language to express execution of actions depending on pattern matching. The user defines rules, which use a left hand pattern that whenever it matches a model configuration, executes a right hand block of imperative code. Mopa abstracts from specific model representations; it uses an abstract data type that describes arbitrary models: The models that Mopa handles have to consist of nodes that form a labelled directed graph, with a known spanning tree, and a function that determines the type of each node. Based on this abstract model representation, Mopa defines a pattern language that allows the user to define model configurations. With Mopa's pattern matching algorithm, those configurations can be searched for in given models. To achieve this, the model graph is traversed along its spanning tree. When a specified configuration is discovered in a model (when a pattern matches) the actions defined in the corresponding rule are executed.

The Mopa language is an extension to the Java programming language that allows the user to write special pattern methods that consist of a set of Mopa rules. These methods can be called on models that are represented by Mopa's abstract model representation. Mopa rules consist of a left hand side Mopa pattern and a right hand side Java code block. When a Mopa pattern method is invoked, the patterns in all rules are successively matched against the given model, and every time a pattern matches, its rule is selected

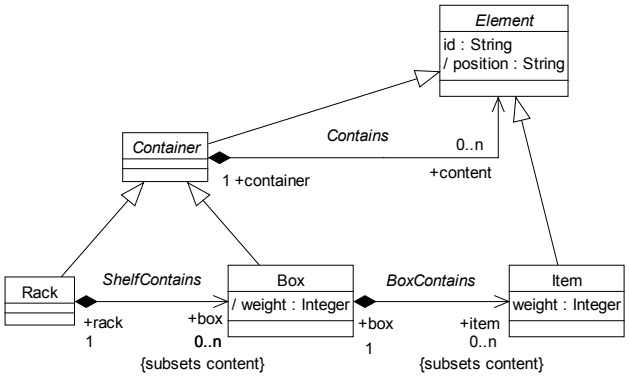

Figure 1. An example meta-model: A warehouse

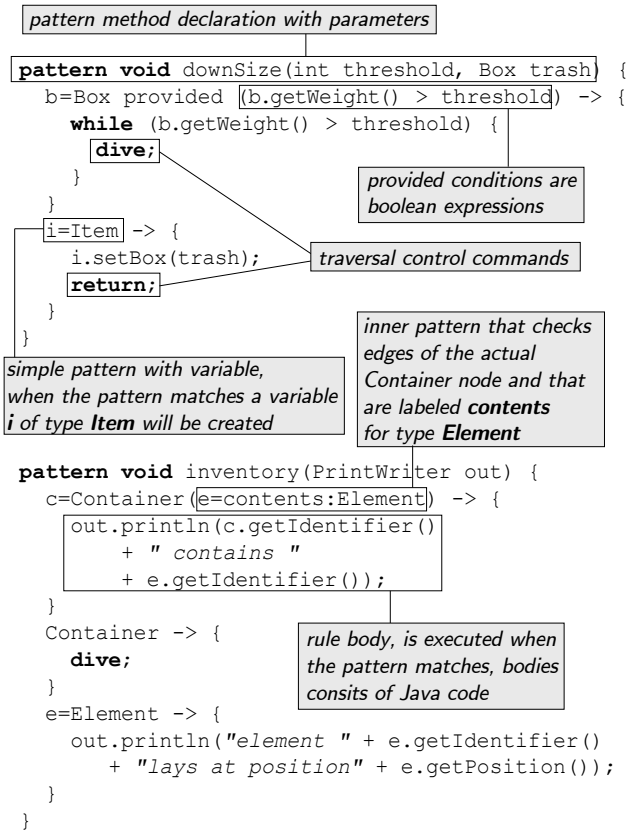

Figure 2. Pattern method examples

and the corresponding right hand side code block is executed.

Figure 2 gives two example pattern methods that work for models based on the *Warehouse* meta-model in figure 1. Consider a software for stock keeping that uses the *Warehouse* meta-model. Based on that model (represented in Java using an JMI-based [16] mapping) the first Mopa method in figure 2 identifies boxes that are heavier than a certain `threshold` and moves items from those boxes to a designated `trashbox`.

Threshold and trash box are given as context parameter to the transformation. The first rule is executed on each instance of `Box` that exceeds the threshold. The threshold condition is given as the rule's *provided condition*. The rule's right hand side uses the `dive` command to apply all rules of the pattern method to the children (according to the spanning tree) of the matched node. This is done again and again until the box's weight has fallen under the threshold weight.

In case of MOF models children are the components of a model element, derived from compositions which form a native spanning tree in MOF based models; in the *Warehouse* model these are the items contained in a box. The corresponding rule that matches these children (items) re-

moves an item and puts it into the trash box. Since all items contained in a box are children of that box, the second rule would match every item the box. To remove only one item at a time, we return the control back to the box rule after one item is put in the trash box; the command `return` is used, to end the currently active dive command and fall back to where the dive command was called, in this case the *while loop* in the first rule.

There is a second example in figure 2 that prints an inventory of the *Warehouse*, using Java IO. The first rule of this method presents a more complex pattern. It describes `Container`-instances that contain an `Element`. This pattern matches for each container-element pair, or more precisely on each for each link of the *Contains*-association. Since the used types *Container* and *Element* are abstract super types of the concrete types *Rack*, *Box* and *Item*, the abstract rules in the `inventory` method work on all kinds of container (*Rack, Box*) and all kinds of elements (*Rack,Box,Item*).

2.2. Related Work

The work on Mopa was inspired by the languages *Kimwitu* [21] and *Kimwitu++* [19]. These languages allow to write rules similar to Mopa rules but for the C and C++ language. Kimwitu(++) has originally been intended to write programs with abstract syntax trees, and therefore has some limitations when it is used for model transformations. Therefore Mopa has added support for data organized in labelled graphs rather then simple trees or terms, Mopa allows a node to have several types to mimic type specialisation and polymorph types, Mopa uses an abstract data type to work on arbitrary data structures, rather then only on data defined in the proprietary Kimiwtu(++) type system.

This way Mopa copes better with model transformation requirements, especially for object-oriented (meta-)models, where model elements are typed by classes that can have several super classes, and where models are organized like directed labelled graphs, meta-modelled with navigable named association ends and attributes. Mopa's syntax is still influenced by Kimwitu(++), but its pattern matching semantics are more oriented on graph pattern matching as it is used in graph transforming approaches [5]. Another addition to Kimwitu(++) are commands that can be used to control model traversal.

Most model transformation languages and technologies are highly specialized tools. They are tailored towards a concrete transformation approach or modelling technique. Often model transformations have to fulfil a certain degree of formality to allow unambiguous specifications or mathematical sound transformation results. In either case it is hard to automatically derive implementations from transformation specifications and it is even harder to integrate such implementations into tools. OMG's transformation recommendation MOF 2.0 QVT (in its pre-final version [20]) for example suggests three different kinds of transformations to broaden the scope of possible QVT applications: (1) a purely declarative approach to define bi-directional relations between languages as transformation specification, (2) a version of this relational approach augmented by means to user control rule scheduling and application strategy, necessary to actually let a machine use this rules to check a relation between existing source and target models, and (3) a more imperative right hand side that is seen necessary to allow the execution (creation of a target model from a source) of an unidirectional defined transformation.

Mopa intentionally is a less specific and less expressive language but is therefore more flexible. It incorporates general purpose programming languages in its transformations to allow the flexibility needed to implement model transformations. The drawbacks are: (1) even with Mopa being formally defined, the combination Java/Mopa is not mathematically predictable, (2) in contrast to typical transformation techniques Mopa is less expressive, hence there is more implementation work for the transformation engineer. The advantages are: (1) one language for multiple model representations (e.g. MOF, JMI, XML, abstract syntax trees, etc.), (2) the possibility to implement aspects that were not originally intended, achieved through arbitrary Java code, (3) a flexible language that can be used for many different transformation approaches.

2.3. Classification

In [9] Czarnecki and Helsen identify a set of features that can be used to classify transformation techniques. We will use this feature-based classification for a more detailed Mopa characterization.

A mandatory feature for model transformation are *transformation rules* that consist of a *left hand* and a *right hand side*. Mopa rules fulfil this requirement. The left hand sides of Mopa rules are *patterns*. Mopa patterns are *syntactically type*; they are based on types and labels, which can be checked against the models that they are used upon. Mopa defines an *abstract* and a *textual concrete syntax* for its patterns. Patterns are *graph patterns* that identify sub-graphs in the source model. The right hand side of Mopa rules are *executable* and have *imperative* character; right hands sides are blocks of Java code. Mopa rules incorporate the optional feature of *syntactically typed variables*. These variables are either defined by patterns with values assigned by pattern matches, or manually defined local, member, or class variables.

Mopa transformation rules *syntactically separate* left hand from right hand sides; they are strictly *unidirectional*. Sets of Mopa rules (pattern methods) can be *parameterised*

to act depending on a given context. A context is formed by additional parameter to pattern methods, or member variables and class variables. Czarnecki and Helsen identified the possibility to use *intermediate model structures* in some model transformation approaches. Mopa does not support such structures itself, but is open to create any kind of Java structure and you can define and schedule different sets of Mopa rules to implement more complex, multi-stage transformations scenarios.

Some approaches use *rule application scoping* on source and target models to restrain transformations to specific parts of models. Mopa has a deterministic behaviour and gives the user explicit control over rule scheduling and model traversal. Therefore the user has the possibility to control which model parts are affected or not.

Some transformation approaches involve the modification of the source model, others completely separate the target model from the source. Both *source-target relationships* are possible when using Mopa. Mopa allows the source to be a valid transformation target and does not constrain the possible updates on the source model. But from own experience this approach has to be done with care, because it is very difficult to predict the behaviour of a transformation when transformation rules are applied to a changing source model.

Transformation systems are rule based, and there has to be a *rule application strategy* that determines on what node a rules is to be applied when the rule matches multiple nodes. Mopa uses a interactive application strategy; in fact Mopa would a apply the rule to all matches, but provides a mechanism—explicit tree traversal control—to let the user determine skip matches or discard rules.

Rule scheduling determines the order in which individual rules are applied. Mopa uses *implicit* rule *iteration* over the given source model nodes. The scheduling is determined by rule order. Rules can disabled by *explicit conditions*. Once a rule is executed, the user can influence further rule scheduling, using the right hand side. The possibilities to influence rule scheduling are to continue with the next rule (continue), to skip all rules for the actual source node and continue with the next node (break pattern), to explicitly apply rules to the node's children before continuation (dive), or to end the transformation process on the actual node set (return).

Mopa rules are *organized* in Java methods. This *modularisation* allows only the *reuse* of hole rule sets. There are no mechanisms for the *logical composition* of rules other then those implemented by the user himself. Rules are organized *source-oriented*; that means the actions that a rule executes (RHS) are attached to source model patterns (LHS).

Mopa has no *dedicated support* for *tracing* of executed model transformations.

3. Foundations

Mopa uses an abstract data type for models, which we define first. Then we introduce the realisation of this abstract data type in Java. Afterwards the actual concrete syntax of the Mopa language is given. Then we present MOPA's abstract syntax and its semantics. MOPA's semantics are explained as a sequence of set-theoretic constructs that lead from its abstract syntax and the abstract data type signature to the behaviour of Mopa methods.

3.1. An abstract data type for models

An abstract data type for models is needed as interface to concrete model representations. The used abstract data type Σ provides: (1) the notion of trees to allow easy model traversal, (2) named properties to define relations between nodes in order to describe attributes and associations, and (3) a type notion to classify model nodes.

The abstract data type Σ is defined with sorts $\mathcal{U}$, $Properties$, and $Types$, functions signatures

$$\begin{aligned} children &: \mathcal{U} \to \wp(\mathcal{U}) \\ link &: \mathcal{U} \times Properties \to \wp(\mathcal{U}) \\ type &: \mathcal{U} \times Types \to \mathbb{B} \end{aligned}$$

and the axiom

$$\neg\exists u_1 \ldots u_n \in \mathcal{U} : [u_{i+1} \in children(u_i)]_{i=1..n-1} \wedge u_1 = u_n$$

Where $\mathcal{U}$ is a sort of arbitrary model nodes, $Properties$ describes labels for relations (e.g. associations) that a node can have with other nodes, and $Types$ is a sort of names for types (e.g. classifier) of nodes. The function $children$ assigns a sequence of child nodes to each node. This function is considered to impose a tree structure, that might have multiple roots, but does not have any circles (see the axiom). The function $link$ assigns a sequence of nodes based on a property. This function describes the edges of directed, labelled graphs. The last function $type$ determines the types of a node. It defines a correlation $\mathrm{R}_{type} \subseteq \mathcal{U} \times type$. Certain modelling environments only allow one type per node: $u_1 \,\mathrm{R}_{type}\, t \wedge u_2 \,\mathrm{R}_{type} \mathrm{t} \Rightarrow u_1 = u_2$. Other type system may allow multiple types per node to mimic polymorph types.

The operator $\wp(\mathcal{U})$ denotes the set of all possible sequences of nodes $N \in \wp(\mathcal{U})$. A sequence N is indexed by natural numbers starting with 1. For all $N \in \wp(\mathcal{U})$ and $x > 0$ is $N(x) \in \mathcal{U} \cup \{undef\}$. There are no holes of undefined elements in a sequence: $N(x) = undef \Rightarrow N(x+1) = undef$. The length of a sequence is defined as $|N| = x$ with $N(x) \neq undef \wedge N(x+1) = undef$. We use sequences instead of sets to reflect that ordering and non uniqueness in structured data might have valuable semantics, and it is easier to masquerade the behaviour of a set with a sequence then vice versa.

Many kinds of data can be described with models of the abstract data type Σ. We want to give a few examples: Metamodelling architectures for object-oriented modelling, like MOF 1.x [18], MOF 2.0[2], EMF[7], or JMI[16], use *meta-classifier* for *objects* in order to model *objects* with common characteristics. Those characteristics are expressed as *structural features*, which usually come in the forms *association ends* or *attributes*. Structural features have names, e.g. role names or attribute names. Structural features can have composite semantics, that means that some features impose circle free graphs, since containment must not be cyclic.

The objects of a model can be mapped to $\mathcal{U}$. Each object has types. These types are the corresponding classifier, as well as generalisations of that classifier. The classifier of an object are used to define $types$. Associations and Attributes define properties that are instantiated as links between objects or properties of objects. These can be used to define $link$. The features that have composite semantics can be used to define $children$.

Another popular example of Σ models are XML documents. The nodes of a document as defined in the Document Object Model [1] define the set of all nodes. Each node has a type, that is either an element definition or an attribute definition. The nodes in a document form a tree structure. Child elements and attributes can also be seen as properties, distinguished by the element's or attribute's name.

Besides these examples there are a lot more, like abstract syntax trees, decision trees, ontology, etc.

To use the abstract date type Σ in the programming language, we define the Java interface in figure 3. Instances of an implementing class for this interface describe nodes in a model. If the *this* parameter for each method is taken into account, then the three interface methods directly conform to the functions signatures defined in Σ. All instances of all implementing classes of this interface form the set $\mathcal{U}$. Properties are described by strings, e.g. property names. The Java type for types is not closer described in order to be more open for all kinds of concrete data structures. This node interface can be easily implemented using the delegator-pattern.

For MOF based models, for instance, we implemented the `Node` interface with JMI. The delegator that implements the *Node* interface uses JMI-reflection facilities to gain container and contents of a model element, to access its properties by feature names, and to inspect its meta-type, either by retrieving an object that actually represents the meta-object or by using the programming language type that reflects the actual meta-type, including its generalisations. An implementation for XML could simply use the node objects of a DOM implementation. Similar implementations are possible for the other examples.

```
interface Node {
    Collection<Node> children();
    Collection<Node> links(String property);
    boolean hasType(Node node, Object type);
}
```

Figure 3. The Java interface for the abstract data type Σ

3.2. Syntax

The MOPA language is designed as an extension to Java. A Mopa program is processed by the Mopa compiler, which generates Java code. A Mopa file, like a Java file, consists of a *public class* definition. Mopa and Java classes can be mixed to form a program. Mopa classes or files use the same syntax as Java except for a special method kind: *pattern*.

These pattern methods are declared, using `pattern` as method specifier. Mopa methods use the syntax in figure 4 as method body. Where `java_pattern_block` is a block of Java statements with three additional possible statements `dive`, `break pattern`, and `continue pattern`. The symbol `type` must correspond to a valid type of the models that the pattern method is indented for. `property` must be a valid property name in that model. The symbol `var_name` must denote a legal java variable name, the symbol `java_expression` denotes a java expression.

3.3. Semantics

The definition of a more abstract syntax is needed in order to conveniently talk about the semantics of Mopa. First will define the syntax of the patterns, the left hand side of Mopa rules:[1]

$$P \in Pattern ::= t | t(p_i P_i)_k.$$

[1] We specify the abstract syntax by a context free grammar production rule, where '::=' denotes *is composed of*, while '|' separates alternatives and all P_i are also produced with this rule.

```
pattern_method_block ::= (rule)*

rule ::= [var_name "="] pattern [provided_clause]
    "->" java_method_block

pattern ::= type [ "(" sub_pattern ("," 
    sub_pattern)* ")" ]

sub_pattern ::= [var_name "="] [property ":"]
    pattern

provided_clause ::= "provided" "("
    java_expression ")"
```

Figure 4. Mopa concrete syntax

Where $t \in \mathit{Types}$ and $p \in \mathit{Properties}$. We use $(a_i b_i)_k$ to note $(a_1 b_1, \ldots, a_k b_k)$. We will also need a helper construct: *sub-graph*, which is defined by:

$$S \in \mathit{Subgraph} ::= (n, x) | (n, x)(p_i S_i)_k$$

Where $n \in \mathcal{U}$ and x a natural number. A sub-graph conforms to a sequence of nodes and is defined as $\mathrm{cf} : \wp(\mathcal{U}) \times \mathit{Subgraph} \to \mathbb{B}$ with:

$$\mathrm{cf}(N, (n, x)) := \begin{cases} \mathit{true} & \text{if } N(x) = n \\ \mathit{false} & \text{, else} \end{cases}$$

$$\mathrm{cf}(N, (n, x)(p_i S_i)_k) := \mathrm{cf}(N, (n, x)) \wedge \bigwedge_{i \in \{1 \ldots k\}} \exists n_i \in \mathit{link}(n, p_i) : \mathrm{cf}(n_i, S_i)$$

For the right hand side of Mopa rules we define commands $\mathcal{C} = \{j, c, b, r, d\}$. Where j denotes an arbitrary Java step of execution, c continue, b break pattern, r return, and d dive. The execution of a Java code block will result in a sequence of such commands, depending on the current environment the block is executed in. This environment consists of the state of all Java variables, objects, static member, etc.; we denote the set of all possible Java states with $\mathcal{E}$. Of course we cannot describe the Java semantics; therefore we will abstract from it. We will later on introduce a function that simulates the Java semantics and gives us the next command in $\mathcal{C}$ for an environment in $\mathcal{E}$.

A pattern P matches a sub-graph S, $\mathrm{match}(S, P)$, only if (1) $S = (n, x)$ and $P = t$ with $\mathit{type}(n, p) = \mathit{true}$ or (2) $S = (n, x)(p_i S_i)_k$ and $P = t(p'_i P_i)_l$ with $k = l$, $\mathrm{match}((n, x), t)$, for all $i \in \{1, \ldots, k\}$ $p_i = p'_i$ and $\mathrm{match}(S_i, P_i)$. In all other cases P does not match S.

When we try to match a pattern P to a sequence of nodes $n \in \wp(\mathcal{U})$ the set of matching sub-graphs is defined as:

$$\mathrm{M}(N, P) := \{S \in \mathit{Subgraph} | \, \mathrm{cf}(N, S) \wedge \mathrm{match}(S, P)\}$$

Finally we define a equivalence relation $=_{N,P}$ and ordering relation $\leq_{N,P}$ on those matches. Since all $S \in \mathrm{M}(N, P)$ do $\mathrm{match}(S, P)$ they all have the same structure and differ only in positions x and actual nodes n. Two sub-graphs S^1 and S^2 have children $(p_i^1 S_i^1)_k^1$ and $(p_i^2 S_i^2)_k^2$ have always the same number of children $k^1 = k^2$ and the same properties $p_i^1 = p_i^2$. The same can be said about each pair (S_i^1, S_i^2) of children and of their children and so forth. We say that all sub-graphs in $\mathrm{M}(N, P)$ have the same *structure*.

We define $=_{N,P}$ recursively on the structure of sub-graphs:

$$\begin{aligned} & (n^1, x^1) =_{N,M} (n^2, x^2) \\ \Leftrightarrow \quad & x^1 = x^2 \end{aligned}$$

and

$$\begin{aligned} & (n^1, x^1)(p_i S_i^1)_k =_{N,P} (n^2, x^2)(p_i S_i^2)_k \\ \Leftrightarrow \quad & (n^1, x^1) =_{N,P} (n^2, x^2) \wedge \bigwedge_{i \in \{1 \ldots k\}} S_i^1 =_{N,P} S_i^2 \end{aligned}$$

Because all sub-graphs in $\mathrm{M}(N, P)$ have the same structure, $=_{N,P}$ is total in $\mathrm{M}(N, P)$.

Lemma: Two $S^1, S^2 \in \mathrm{M}(N, P)$ yield $S^1 =_{N,P} S^2 \Rightarrow S^1 = S^2$. *Proof:* S^1, S^2 have the same structure, since both are in $\mathrm{M}(N, P)$; both have everywhere the same node positions x, because of the definition of $=_{N,P}$. Thus assuming there would be some $S^1 \neq S^2$ with $S^1 =_{N,P} S^2$ they would only differ in nodes n. But that would mean that for some node $n_1 = n_2$ and property p and position x $\mathit{link}(n_1, p)(x) \neq \mathit{link}(n_2, p)(x)$, what is not possible. □

From now on we use $=$ for $=_{N,P}$. We define $\leq_{N,P}$ recursively on the structure of sub-graphs.

$$\begin{aligned} & (n^1, x^1) \leq_{N,M} (n^2, x^2) \\ \Leftrightarrow \quad & x^1 \leq x^2 \end{aligned}$$

and

$$\begin{aligned} & (n^1, x^1)(p_i S_i^1)_k \leq_{N,P} (n^2, x^2)(p_i S_i^2)_k \\ \Leftrightarrow \quad & (x^1 \leq x^2) \\ & \vee \quad \left[x^1 = x^2 \wedge S_i^1 = S_i^2\right]_{i=1..l} \Rightarrow S_l^1 \leq_{N,P} S_l^2 \end{aligned}$$

Because all sub-graphs have the same structure in $\mathrm{M}(N, P)$, $\leq_{N,P}$ is a total order in $\mathrm{M}(N, P)$.

Lemma: The total order $\leq_{N,P}$ is a *well order* in $\mathrm{M}(N, P)$. *Proof:* We will show the minimum S_m in each subset $\mathcal{S} \subseteq \mathrm{M}(N, P)$ with $\forall S \in \mathcal{S} : S \leq_{N,P} S_m \Rightarrow S = S_m$. We use recursion over the structure of sub-graphs (all have the same in all $\mathcal{S}$). If (1) $\mathcal{S} = \{(n_i, x_i)\}_k$ then $\mathcal{S}$ has a minimum because $\leq$ is a well order in $\mathbb{N}$ and every $S \in \mathcal{S}$ is unique; if (2) $\mathcal{S} = \{(n_i, x_i)(p_j S_{ij})_l\}_k$, the definition is:

$$\begin{aligned} \mathcal{S}_0 \; = \; & \{S = (n_i, x_i)(p_j S_{ij})_l | S \in \mathcal{S} \wedge \\ & \qquad (n_i, x_i) \text{ is minimum in } \{(n_i, x_i)\}_k\} \\ \mathcal{S}_q \; = \; & \{S = (n_i, x_i)(p_j S_{ij})_l | S \in \mathcal{S}_{q-1} \wedge \\ & \qquad S_{iq} \text{ is minimum in } \{S_{1q} \ldots S_{kq}\}\} \end{aligned}$$

The last set $\mathcal{S}_k$ contains only one S, this is the minimum S_m of $\mathcal{S}$. □

We use the well order $\leq_{N,P}$ on $\mathrm{M}(N, P)$ to define the sequence $\mathrm{M}_{\leq}(N, P)$.

When we apply a pattern P to a sequence of nodes N, we retrieve a sequence of sub-graphs $\mathrm{M}_{\leq}(N, P)$ that conforms to the nodes in N and contains matches of P. The order

of matches in $\mathrm{M}_{\leq}(N, P)$ is completely determined by the order of nodes in N and order of nodes in $link(n, p)$. In other words when we apply a pattern to nodes of structured data, we derive an ordered set of matches that reflects the order of elements given by the structure of these nodes. If the order in the given data has a defined semantics, it is preserved. In the other case, if there is no specific order, and sequences are only arbitrary, Mopa will still use this order, but the user can ignore it.

Now that the semantics of patterns are clear, we will define the application of rules. A sequence of rules $(P_i, B_i)_k$, an environment E_0, and a starting sequence of nodes N_0 are given. The set of *traversal points* TP is recursively defined: (1) the end point e is a traversal point, (2) (N, i_P, i_S, t) is a traversal point, where N is the sequence of actual nodes, i_P denotes the index of the actual pattern with $1 \leq i_P \leq k$, i_S denotes the actual match with $1 \leq i_S \leq |\mathrm{M}_{\leq}(N, P_{i_P})|$, and $t \in TP$ is the parent traversal point.

A traversal on N_0 and $(P_i, B_i)_k$ starts with $(N_0, 1, 1, e)$. The function

$$java : TP \times \mathcal{E} \to \mathcal{C} \times \mathcal{E}$$

reflects the execution of a statement in a corresponding code block with an environment $\in \mathcal{E}$. For (N, i_P, i_S, t) is (P_{i_P}, B_{i_P}) the corresponding rule and B_{i_P} the corresponding code block. The function $java$ is an abstraction of Java semantics that, of course, cannot be given here. The function $java$ is assumed to be total and it is defined by the Java programming language, except for $java((N, i_P, 0, t, E) = (c, E)$; this is needed to reflect that a pattern might have no match. We use this trick to express that when a pattern does not match any node, nothing is done but continued.

The actual state of a traversal is an element of $TP \times \mathcal{E}$. We will describe the traversal semantics of Mopa as a function

$$mopa : TP \times \mathcal{E} \to TP \times \mathcal{E}$$

that gives the successor state to a given state and the successor traversal point to a given traversal point.

$$mopa(t, E) := \begin{cases} (t, E') & \text{if } java(t, E) = (j, E') \\ (cont(t), E') & \text{if } java(t, E) = (c, E') \\ (break(t), E') & \text{if } java(t, E) = (b, E') \\ (dive(t), E') & \text{if } java(t, E) = (d, E') \\ (return(t), E') & \text{if } java(t, E) = (r, E') \end{cases}$$

The execution of $(P_i, B_i)_k$, on nodes N_0 and environment E_0 starts with $((N_0, 1, 1, e), E_0)$. The functions for the traversal commands are defined as follows.

Continue, if another match exists then continue executes the code block on the next match, if not it goes to the next pattern , if that doesn't exist either, it continues at the parent position. The state is not changed if we continue on the end point e:

$$cont(e) := e$$

$$cont(t) := \begin{cases} cont(t) & \text{if } i_S = m_S \wedge i_P = k \\ (N, i_P + 1, 0, t) & \text{if } i_S = m_S \wedge i_P < k \\ (N, i_P, i_S + 1, t) & \text{if } i_S < m_S \end{cases}$$

where the actual traversal point $t = (N, i_P, i_S, t)$ and $m_S = |\mathrm{M}_{\leq}(N, P_{i_P})|$ the number of matches for the actual Nodes N and the i_p-th pattern P_{i_P}.

Break pattern continues the execution with the next pattern, all other matches of the actual pattern are discarded:

$$break((N, i_P, i_S, t)) := \begin{cases} cont(t) & \text{if } i_P = k \\ (N, i_P + 1, 0, t) & \text{if } i_P < k \end{cases}$$

Return continues the execution on the parent node:

$$return(t) := cont(t)$$

Dive traverses down the tree; the execution of the current block is paused an the rules are applied to the children of the actual node in the actual match:

$$dive((N, i_P, i_S, t)) := (children(n_{i_S}), 1, 0, (N, i_P, i_S, t))$$

with $(n_{i_S}, x)(\dots) = \mathrm{M}_{\leq}(N, P_{i_P})(i_S)$.

Finally we can give the behaviour (a sequence of states in $TP \times \mathcal{E}$) as a result of applying $(B_i, P_i)_k$ to N_0 and start state E_0:

$$(t_0, E_0), (t_1, E_1), \dots, (t_n, E_n)$$

where $mopa(t_i, E_i) = (t_i + 1, E_i + 1)$, $t_0 = (N_0, 1, 0, e)$, and $t_n = e$.

4. Application

This section is about applying Mopa, how to it is used and how it performs for the different model transformation approaches. Czarnecki and Helsen [9] categorize model transformation into two major approaches *model-to-code* and *model-to-model*. Where *model-to-code* describes the less generative—but more used—task of creating program text in a specific implementation language, in contrast to *model-to-model* transformation that allows transformation into the domain of well defined (meta-modelled) modelling languages. We introduce another approach—*Mode-to-execution*—that describes the transformation from a model to executed actions. The use case is the implementation of execution semantics for languages in the context of language driven development; simulation is one example (compare to Executable Metamodelling in [8]).

4.1. Model-To-Execution

Model-To-Execution—the execution of a model—is a simple method to directly realize the semantics of languages. It is realized through rules that define what actions to be execute for each instance of a language concept, or as Clark and Evans put it in [8]: ”This [executable metamodelling] is achieved by augmenting the modelling language with an action language or other executable formalism”.

Where Clark and Evans, as well as other (JetBrains [10]), use an own language (that is consequently bootstrapped with their own metamodelling language) to define the action semantics of their metamodelling technique. We, in contrast, map models to the execution of Java statements.

One Mopa feature is especially helpful to implement such semantics: Mopa’s determined order of pattern matches and ability to control the traversal of a model. When implementing the execution of a model it is often vital to control the order of actions taken in order to achieve the desired language semantics. Mopa preserves any orders existing in the model; this is a necessary feature, common among metamodelling and model transformation tools. But Mopa also allows ordered execution within multiple rules. The pattern in two distinct rules might match the same subgraph (part of a model) in part or as a whole and execute different actions on them. Imagine a general, abstract rule that describes a distinct aspect of a language and that implies actions for all matches of a very common model pattern. In addition, imagine another rule that this time describes the very specific semantics of a specific concept that only requires execution of actions for the rare matches of a special pattern. Mopa allows to express order between those rules and concluding the order between the actions they imply.

One important aspect of model execution is the changeability of a model. Where languages with semantics that is based on transformations into other languages, have static models, execution semantics of models could also allow dynamic changes to the model. You can compare such dynamic models to interpreted languages (e.g. perl, phython, tcl), or functional languages based on term-rewriting (e.g. lambda calculi, etc.): Interpreted languages often allow to create parts of a program dynamically using some kind of *eval* statement; functional languages can be executed by modifying the program (e.g. term) itself by mapping function applications to term-rewriting.

Mopa does not limit the right hand side of its rules to anything, since they are just lines of Java code; hence it is possible to manipulate the source model itself. Using the lambda calculus example: Imagine a rule that matches a first order, single parameter lambda function and its argument, the rule can first replace all free occurrences of the function parameter identifier with the argument and can then trigger the execution of the body by applying patterns to the modified body using the *dive* command.

4.2. Model-To-Code

Model-To-Code describes the generation of program code from a model. In the context of MDA this is probably the most used approach in existing MDA tools; Czarnecki and Helsen say: ”Most existing MDA tools provide only model-to-code transformations, which they use for generating PSMs (in this case being just the implementation code) for PIMs.” This is also probably the best researched approach, because code generation is an applied technique since long before the ages of MDA and UML.

And here lays the main advantage of Mopa, using arbitrary Java code to execute rules also allows the usage of programming language libraries. Therefore it is possible to reuse existing code generation frameworks with Mopa. You can use Mopa’s deterministic pattern matching and explicit traversal mechanism to control and schedule the code generation using either existing code generation frameworks or ad-hoc techniques like print writing.

Model-To-Code approaches can be divided into *visitor-based approaches* and *template-based approaches*. Both approaches can be realized with Mopa. Visitor-based simply means that the model is traversed and code is generated for each element visited. This is very natural for Mopa and can be implemented easily: Rules for each meta-concept describe what code to generate and which elements to visit next.

Template-based approaches use target text that includes placeholders that are to be replaced with concrete data in concrete template instances. Rules in such template-based approaches usually access information in the source model using the left hand side of a rule and give a template as right hand side that incorporates this data in a piece of target code. In Mopa, patterns are used to locate data in the source model and bind it to Java variables, which then can be used to instantiate a template in the right hand side of a rule. You can either instantiate own code templates or use existing template libraries.

Compared to visitor-based approaches template-based approaches have the advantage that they reuse pieces of code and thus are less error prone then most print writer based visitor-based tools. But still the model-to-code approach in general leaks safety. The used program code, whether as templates or not, is not syntactically checked, type-safe or otherwise semantically proofed. But despite this weakness model-to-code is the most used approach, probably because of the simpler development process, and the achievement of (at least at first sight) faster results.

4.3. Model-To-Model

Model-to-model means transformation between source and targets models, where both are instances of well defined metamodels; the elements and the relationships of elements of both source and target models are instances of defined classifiers and associations. Such a transformation environment is more safe and less error prone than the former approaches. That is one reason why such an approach is desperately searched for in order to formally define the relations between languages. Beside the need to simply express the mappings and transformations between languages, a technology is needed that allows execution of transformations with reasonable performance.

The problem of adequate model-to-model transformations is an often discussed and open issue of MDA. In Gerber et. al.'s *Transformation: The Missing Link of MDA* [14]: "[well established foundations exist for meta-modelling], no such well-established foundations exists for transforming PIMs to PSMs." Many approaches to model-to-model transformation existed at that point, but yet non manifested as *the* MDA technique to describe language mappings and transformations. The OMG issued a request for proposal in 2002 called *Query View Transformations* [4] to create a recommendation for model-to-model transformations based on UML/MOF 2.0 [3, 2, 13, 20]. We relate Mopa to QVT and others classified by Czarnecki and Helsen [9].

Relational and Graph-Transformation Approaches Relational approaches use mathematical relations to declarative express relations between models. Because of its declarative character those approaches are often only applicable to check whether two models relate in a certain way to each other or not, but you can not generally derive an executable transformation from it. In [20] however it is distinguished between bi-directional, non-executable relations and unidirectional and executable mappings, which are used to actually implement transformations.

Graph-transformation approaches use graph-transformation as described in [5]. Graph-transformation are described by LHS/RHS rules, where the left hand side consist of a sub-graph which is searched for in the source model and which is replaced by the graph given in the right hand side, when the rule is applied. The LHS-graph is cut out using an interface graph that describes the points where the sub-graph is torn out and a glue graph on the RHS describes how the replacing RHS graph has to be inserted.

Both approaches relational and graph-transformations are real formalisms backed by mathematical foundations and formality. Most approaches of that kind, as for example [20] and [5] present closed formalisms that cannot easily be extended or interact with other techniques. And that is usually the main weakness of such approaches: They describe languages for a very specific use with small boundaries in which transformation can be expressed. Approaches of both kinds are usually added heavy machinery to remain flexible; they usually define means to constrain rules, combine or specialize existing rules. Given the trade-off between formality and applicability, approaches in this category often operate on the far mathematical side. There are no major applications of such techniques that are used to actual implement model transformations; quite the opposite, it is more realistic to assume that model transformations will sure be formally specified using such well defined approaches, but are implemented using a different technique.

The Structure-Driven Approach The basic metaphor for this approach is to create target elements for each source element and to adapt the target elements to achieve the desired transformation result. The assumption that reasons this approach is that target models often have similar structure than their source models. Methods using this approach (like Interactive Object's QVT proposal [13, 15]) often navigate the containment hierarchy of the source model to create a top-down configuration of the target model. This top-down approach has the problem, that not all model information lays out in a top-down manner; when a source element is transformed it often refers to elements not yet transformed and thus there exists no target to refer to. According to [9] this approach is often divided into two phases: A first phase that creates the hierarchal structure of the target model, and second phase that sets attributes and references in the target.

This in not intuitive, because it separates actions from each other that logically belong together. When using Mopa for this approach we recommend another strategy: Mopa patterns are used to select elements in the source model, whereby the pattern can constrain elements to distinct contexts; Mopa transversal control commands can be used for a determined source model navigation; target elements are created within the rules' right hands sides. To cope with the problem of non existing referred target elements, we suggest the notion of a *flux box*. This concept was used and introduced for MOF-based model transformations in [6]. A flux box uses a model element factory to create the target elements; the reason to use the flux box instead of the factory itself is that the flux box can even give you elements that will be *created* later in time. When you need to refer to an element not yet existent, the flux box will create that element in advance and hand it to you to reference it; when now the time has come to actually create the element, the flux box will hand the same former created element to you.

Hybrids – The QVT Approach Hybrid approaches combine declarative techniques, as used in relational or graph transformation approaches, with imperative techniques. OMG's MOF 2.0 QVT [20] recommends a sorely declara-

tive relational approach to specify transformations and concurrently suggests imperative mapping rules for implementation. The thereby used mapping rules, use a declarative left hand patterns augmented by keywords that give the user some rule scheduling control. The right hand side of mapping rules consist of sequence of executed commands.

Gardner et. al. [13] suggest the following hybrid approach: "A combination of declarative and imperative constructs to define transformations. Typically a declarative approach is used to select rules for application and an imperative approach is used to implement the details of rules that are not completely expressed declaratively." Mopa patterns are such a declarative left hand side that select rules for execution, using a pattern matching algorithm, and Mopa right hand sides are imperative Java code block that can be used to create target model elements.

5. Conclusions

We introduced the Mopa language and showed how to use it for existing applications to model transformation. Mopa performs well in applications of model-to-execution and model-to-code. Here it takes advantage out of the fact that Mopa is basically Java code that as such can be transparently integrated into any Java application and of course use any existing Java library, e.g. libraries for code generation or libraries for execution environments, like simulation frameworks. For model-to-model application Mopa needs assisting modules to perform well. Rule selection and scheduling can adequately expressed using Mopa, but the actual model creation or manipulation has to be given in plain Java code. This again is flexible and open but on the other hand leak expressiveness; all the flexibility marks the major drawback of Mopa: Since it is not very specific, it is also not very expressive.

A strong point is that Mopa abstracts from concrete modelling platforms and allows easy adaptation to all possible kinds of models. This does not only include object-oriented meta-modelling frameworks like MOF, JMI, or EMF. For instance, Mopa was successfully used on XML based data to handle models in XMI and on abstract syntax trees (ASTs) to implement Mopa itself.

On a scale between highly specialized transformation language to multi purpose programming language Mopa can be spotted more on the right side. But this scale also poses a serious trade-off between specialization and flexibility; a trade-off that most proposed transformation languages trade unbalanced.

References

[1] *Document Object Model (DOM)*. World Wide Web Consortium (W3C).

[2] *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group, Oct. 2003. pct/03-10-04.

[3] *UML 2.0 Infrastructure Specification*. Object Management Group, Sept. 2003. pct/03-09-15.

[4] *MOF 2.0 Query/Views/Transformations RFP*. Object Management Group, Apr. 2004. ad/2002-04-10, Request for Proposal.

[5] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schuerr, and G. Taentzer. Graph Transformation for Specification and Programming. 1996. Technical Report 7/96, Universitaet Bremen.

[6] H. Boehme, G. Schuetze, and K. Voigt. Component Development: MDA Based Transformation from eODL to CIDL. In *Twelfth SDL Forum*, June 2005.

[7] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, Aug. 2003.

[8] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling, A Foundation for Language Driven Development*. 2004.

[9] K. Czarnecki and S. Helson. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[10] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. JetBrains, 2004.

[11] J. Fischer, E. Holz, A. Prinz, and M. Scheidgen. Tool-based Language Development. In *Workshop on Integrated-reliability with Telecommunications and UML Languages*, Nov. 2004.

[12] J. Fischer, A. Kunert, M. Piefel, and M. Scheidgen. ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages. In *Twelfth SDL Forum*, June 2005.

[13] T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review on OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. In *MetaModelling for MDA, Workshop*, 2003.

[14] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. In A. Corradini et al, editor, *ICGT 2002m LNCS 2505, pp. 90-105*, 2002.

[15] Interactive Objects and Project Technology. MOF Query/Views/Transformations, Revised Submission. Object Management Group, 2003. ad/03-08-11, ad/03-08-12, ad/03-08-13.

[16] JMI. *The Java Metadata Interface(JMI) Specification(Final Release)*. Java Community Process, June 2002. JSR-000040.

[17] MDA. *Model Driven Architecture Guide, Version 1.0.1*. Object Management Group, June 2003. omg/03-06-01.

[18] MOF. *Meta Object Facility, Version 1.4*. Object Management Group, Mar. 2003. formal/2002-04-03.

[19] T. Neumann and M. Piefel. *Kimwitu++– A Term Processor*. Humboldt-Universität zu Berlin, 2002.

[20] QVT-Merge Group. *MOF 2.0 Query/Views/Transformations*. Object Management Group, Apr. 2004. ad/04-04-01, Revised Submission.

[21] P. van Eijk and A. Belinfante. *The Term Processor Kimwitu – Manual and Cookbook*. Apr. 2000.