Pattern Matching in MOLA

Agris Sostaks

IMCS University of Latvia, Latvia
agris.sostaks@lumii.lv

Abstract. This paper addresses the pattern matching problem for model transformation languages. Despite being an NP-complete problem, the pattern matching can be solved efficiently in typical areas of application. A simple pattern matching strategy is proposed for MOLA model transformation language which is efficient for tasks related to the model driven software development. A more advanced solution is also introduced for other domains. It is a local search plan strategy combined with the metamodel annotation mechanism, which allows using the developer's knowledge of model constraints that otherwise could be obtained only by analysis of existing models.

1 Introduction

Model transformation languages are becoming increasingly mature in recent years and range of the areas where transformation languages are being used is widening. The growing popularity of transformation languages puts stricter requirements on their efficiency. Most of the popular transformation languages are using declarative pattern definition constructs. The main implementation problem of such languages is the pattern matching. This problem, in fact, is the subgraph homomorphism problem which is known to be NP-complete. However, in practice typical patterns can be matched efficiently using relatively simple methods. The use of different means of pattern definition results into different implementations of pattern matching for every language. The more sophisticated constructs a language use, the more complicated becomes the implementation of the pattern matching. The pattern matching implementation for the graphical model transformation language MOLA [1] is addressed by this paper.

One of the most popular application domains for model transformations is Model Driven Software Development (MDSD) related tasks. These tasks have been tried to be solved in almost every model transformation language, also in MOLA. In fact, MOLA is designed as a simple and easy readable (therefore graphic!) model transformation language, which would cover the typical transformation applications in MDSD. We refer to transformations used in the European IST 6th framework project ReDSeeDS[1] to illustrate the typical MDSD use cases of MOLA. We present brief overview of MOLA in Section 2.

[1] http://www.redseeds.eu

Section 3 gives a survey of pattern matching strategies used in implementations of model transformation languages. One of the most popular and also the most efficient method to solve the pattern matching problem is the local search plan generation. This method is used by several implementations of transformation languages, e.g. (PROGRES [2], VIATRA [3], GrGen [4] and Fujaba [5]). However, each implementation varies in details depending on pattern definition constructs used in the language. Most sophisticated strategies even use statistical analysis of model in runtime.

The implementation of the pattern matching in MOLA is discussed in Section 4. MOLA also uses the local search plan generation strategies for pattern matching. We propose simple heuristic algorithm that is efficient for most patterns typically used in MDSD-related transformations. This algorithm can be used in fast implementations of pattern matching for transformation languages which use similar constructs to MOLA. However, more advanced algorithms (like in abovementioned implementations of other languages) should be used to cover cases, when the simple algorithm does not succeed. Almost all existing algorithms uses cardinalities of metamodel elements to determine the best search plan, however actual cardinalities are not shown in metamodel. Therefore, we introduce the metamodel annotation mechanism, which allows using developer's knowledge of actual model element cardinalities that otherwise could be obtained only by analysis of existing models.

2 MOLA

MOLA is a graphical transformation language developed at University of Latvia, Institute of Computer Science and Mathematics. It is based on traditional concept among transformation languages: pattern matching. The main distinguishing feature is the use of simple procedural control structures governing the order in which pattern matching rules are applied to the model. The formal description of MOLA and also MOLA tool can be found in MOLA web site[2].

One of the biggest use cases of MOLA is in the European IST 6th framework project ReDSeeDS. One of goals in ReDSeeDS is providing tool support for MDSD. The Software Case Language (SCL) is used to model the system. The main parts of SCL are Requirements Specification Language (RSL)[6] and a subset of Unified Modelling Language (UML). Requirements in RSL are scenarios in a controlled natural language. MOLA is used to specify a transformation from requirements to architecture and further to detailed design models. Transformations are divided into several steps generating a chain of models. We have gained a great experience during this project writing typical MDSD transformations. Short excerpts from these transformations are used in this paper.

2.1 Short Description of MOLA

A MOLA program transforms an instance of a source metamodel (the source model) into an instance of a target metamodel (the target model). Source and

[2] http://mola.mii.lu.lv

target metamodels are jointly defined in the MOLA metamodelling language, which is quite close to the OMG EMOF specification. Actually, the division into the source and target parts of the metamodel is quite semantic, they are not separated syntactically (the complete metamodel may be used in transformation procedures in a uniform way). Typically, additional traceability associations or classes link the corresponding classes from source and target metamodels; they facilitate the building of natural transformation procedures and document the performed transformations.

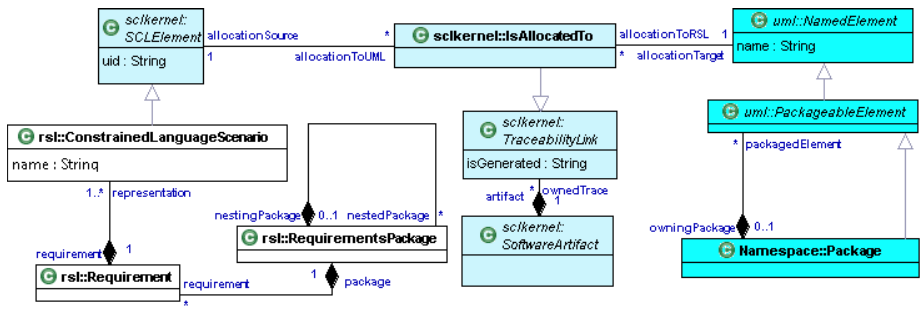

Fig. 1. Fragment of SCL used in transformation example

Figure 1 demonstrates part of the SCL metamodel used in this section as a transformation example. It is a simplified version of full SCL metamodel. As we can see the source and target metamodels are given in the same class diagram and the separation into source (RSL) and target (UML) metamodels is quite semantic. The requirements specification (requirement model) consists of requirements packages which are used to group the requirements in RSL. One of requirement representation forms is the constrained language scenario. Thus a requirement may consist of several scenarios written in the constrained language. The traceability elements play also a significant role in MOLA transformations. The *sclkernel::TraceabilityLink* class (particularly, the *IsAllocatedTo* for mapping from requirements to architecture) is used for this purpose.

MOLA procedures form the executable part of a MOLA transformation. One of these procedures is the main one, which starts the transformation. MOLA procedure is built as a traditional structured program, but in a graphical form. Similarly to UML activity diagrams, control flows determine the order of execution. Call statements are used to invoke sub-procedures. However, the basic language elements of MOLA procedures are specific to the model transformation domain - they are rules and loops based on rules. Rules embody the declarative pattern match paradigm, which is typical to model transformation languages. Each rule in MOLA has the pattern and action part. Both are defined by means of class elements and links. A class element is a metamodel class, prefixed by the element ("role") name (graphically shown in a way similar to UML instance).

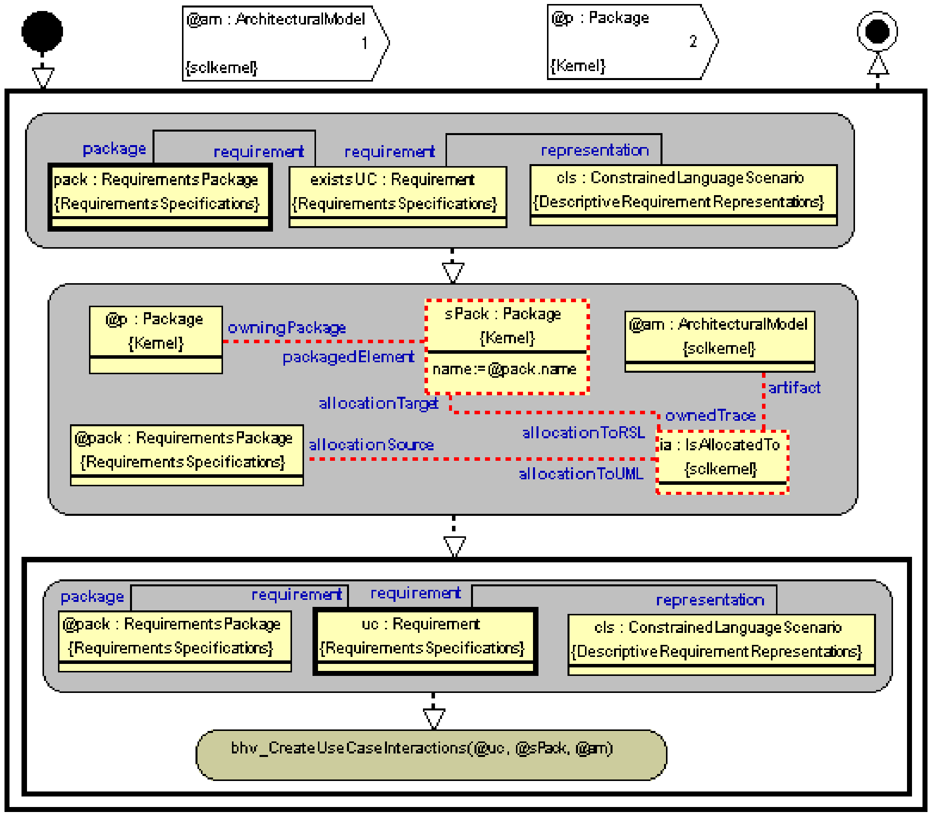

Fig. 2. Transformation example - MOLA procedure building package hierarchy

An association link connecting two class elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class elements and links, which are compatible to the metamodel for this transformation. A pattern may simply be a metamodel fragment, but a more complicated situation is also possible - several class elements may reference the same metamodel class. A class element may be a reference to previously matched instance. This reuse mechanism plays crucial role in the implementation of pattern matching. In addition, a class element may contain also a constraint - a simple Boolean expression. The main semantics of a rule is in its pattern match - an instance set in the model must be found, where an instance of the appropriate class is allocated to each pattern element so that all required links are present in this set and all constraints evaluate to true. If such a match is found, the action part of the rule is executed. The action part also consists of class elements and links, but typically these are creation actions. Instances may also be deleted and modified in the action part. If several instance sets in the model satisfy the rule pattern, the rule is executed only once on an arbitrarily chosen match. Thus, a rule in

MOLA typically is used to locate some fragment in the source model and build a required equivalent construct in the target model.

Another essential construct in MOLA is the foreach loop. The foreach loop is a rectangular frame, which contains one special rule - the loophead. The loophead is a rule, which contains one specially marked (by a bold border) class element - the loop variable. A foreach loop is an iterator, which iterates through all possible instances of the loop variable class, which satisfy the constraint imposed by the pattern in the loophead. With respect to other elements of the pattern in the loop head, the "existential semantics" is in use - there must be a match for these elements, but it does not matter, whether there is one or several such matches. A loop typically contains the loop body - other rules and nested loops, whose execution order is organised by control flows. The loop body is executed for each loop iteration.

Figure 2 shows a typical MOLA procedure written for ReDSeeDS project. This procedure is a part of transformation which builds sequence diagrams from requirements written using constrained language scenarios. The task for this procedure is to build a UML package and traceability link **for each** requirements package which contains appropriate requirements. Next, **every** requirement in **these** packages should be processed. Note the similarity of bolded linguistic constructs used in the description and MOLA constructs - the foreach loop and reference. This allows describing such algorithms very straightforwardly and easily in MOLA. The outer foreach loop (the bigger black frame) iterates through every requirement package, which has at least one requirement with a scenario. The loophead is the rule containing the loop variable (the uppermost rule in the loop). The loop variable (*pack: RequirementsPackage*) is used to explicitly denote the class to iterate through. A constraint which filters matched instances is given using the additional class elements (*existsUC* and *cls*). It restricts the iteration to requirement packages which contain a requirement represented by the constrained language scenario. This foreach loop contains also a loop body which consists of a rule and another foreach loop. They are executed in every iteration of the loop. The rule, which creates a UML package (*sPack* class element), places this package into the appropriate top-level package (*owningPackage* association link to *@p*), sets the name of the package and finally creates the traceability element (*ia* class element and *allocationSource* and *allocationTarget* association links). The nested loop is a typical approach in MOLA to iterate through contained elements. The inner loop iterates through all requirements (loop variable *uc*) which are in the matched requirements package (the reference *@pack*) and which contain a scenario (the class element *cls*).

3 Related Approaches of Pattern Matching

The closest "relative" to MOLA in the world of model transformation languages is Fujaba Story Diagrams from Fujaba Tool Suite [5]. Fujaba is a graphical model transformation language which uses imperative control structures and declarative patterns. The specification of patterns in Fujaba is almost identical to MOLA. There is a restriction on patterns in Fujaba - the pattern must contain at least one bound (previously matched) element. Graphical syntax, of course, differs for both languages, but that is obvious for independently developed languages. The significant difference between the two is the foreach loop. Fujaba does not specify the loop variable and loop is executed through all of the possible matches of a pattern. In MOLA only the distinct instances that correspond to the loop variable are iterated over. MOLA foreach loop is more readable and easier to use, because of the loop variable.

A different programming paradigm is used in graph transformation language AGG [7], which is a typical example of a declarative transformation language. AGG does not have any imperative control structures, and rules that describe patterns are being executed independently. The only way to affect the execution order is to use layering. Each rule in AGG includes the pattern which is specified by LHS graph and NACs. NACs are used by declarative transformation languages mainly to distinguish already processed model elements. Negative patterns are used differently in MOLA because of the specific loop construct. MOLA also has negative pattern elements, but they are used to express a "logical" negative condition. The graph transformation language PROGRES [2] is a textual graph transformation language where patterns (graph queries) are specified graphically. Patterns allow using similar and even richer options than previously noted transformation languages. The ordering of statements is managed by algebraic structures and PROGRES follows declarative PROLOG-like execution semantics. Graph transformation language VTCL (Viatra Textual Command Language), which is part of the VIATRA 2 framework [3], defines patterns using textual syntax. VIATRA offers broad possibilities for the pattern definition: negative patterns may be at arbitrary deep level; the call of a pattern from another pattern, and even recursive patterns are allowed; the language may work both with model and metamodel. The execution order of rules is managed by ASM (Abstract State Machine) language constructs which are purely imperative. VIATRA has rudimentary graphical syntax of patterns, however it seems that whole expressiveness of language may not be available there. Another textual graph transformation language, which has appeared in recent years, is GrGen [4]. The expressiveness of patterns in this transformation language is close to VIATRA. Transformation rules are combined using similar algebraic constructs to PROGRES (except the PROLOG-like execution semantics).

Almost all model and graph transformation languages that use similar pattern concepts as MOLA are forced to deal with pattern matching task. There are four most popular algorithms that are used by transformation language implementations to solve the pattern matching problem:

1. **Local search plan generation**. The search of the instances corresponding to pattern is started from a single instance, which is a potential match. Next, the adjacent instances are examined according to the given pattern. If examined instances do not match or another valid match is needed, backtracking is required. Search plan is the order in which the potential matches are examined. In other words, search plan is the sequence in which pattern elements are traversed

in order to find a valid match. The search plan generation algorithm must examine various search plans and evaluate them in order to choose one that is least expensive. The algorithm uses a cost model of basic search operations. Then the search graph based on pattern is built and weights are attached according to the cost model. At last, the appropriate ordering is determined from the graph.

2. **Reducing to CSP** (Constraint Satisfaction Problem[8]). The pattern matching problem is reduced to equivalent CSP. Pattern elements are mapped to the variables and constraints. This enables to use all the techniques known in the area of CSP. The main techniques are related to the appropriate ordering of variables and efficient use of backtracking. Thus, in general both methods, local search plan generation and reduction to CSP, are quite similar, but CSP puts more emphasis on intelligent backtracking.

3. **Using relational database**. Using relational database reduces the pattern matching problem to taking advantage of the power of query optimization in relational databases management systems. The task is to choose an appropriate database schema to store the model and to generate the SQL query which returns the valid matches.

4. **Incremental pattern matching**. The core idea of incremental pattern matching is to make the occurrences of a pattern available at any time. It requires caching all occurrences of a pattern and incremental updating whenever changes are made. If this requirement is met then the pattern matching is made in almost constant time (linear to the size of result set itself). However, the drawbacks are memory consumption and overhead on update operations.

PROGRES was the first transformation language addressing the pattern matching problem[9]. It uses the local search plan generation method. PROGRES builds a pattern graph, where a node is built for each pattern element. Next, the operation graph is built, adding information about all basic operations that may be used by pattern matching. The cost of each search operation is derived from heuristic assumptions and knowledge on multiplicities of pattern elements. The best-first method is used to determine the search plan from the operation graph.

VIATRA has been implementing most of pattern matching algorithms. Relational database algorithm for VIATRA uses a separate table for each class [10]. An appropriate SQL query is used for finding the pattern matches.

The generation of local search plans also has been used by VIATRA[11]. The search graph is built for a pattern. An additional starting node is added to the graph. Directed edges connect the starting node to every other search graph node. Each edge of the pattern is mapped to a pair of edges in the search graph, expressing the bidirectional navigability. The cost model is obtained by analyzing the existing models of the domain, e.g. typical UML class diagram, if the UML class diagram is being transformed. The collected statistics illustrate an average branching factor of a possible search space tree, built when pattern matching engine selects the given pattern edge for navigation. Costs are added to the search graph and the minimum spanning tree (MST) is found with the starting node taken as the root node. The search plan is determined from MST.

An incremental pattern matcher[12] (RETE network) is constructed based on the pattern definitions. Before transformation the underlying model is loaded into incremental pattern matcher as the initial set of matches. The pattern matching is performed efficiently; however the changes should be propagated within the RETE network to refresh the set of matches.

The authors of VIATRA have introduced a hybrid pattern matching approach[13], which is able to combine local search and incremental techniques on a per-pattern basis. Two scenarios are proposed: design-time selection of strategy by developer and runtime optimization based on monitoring of statistics (available memory or model space statistics).

GrGen uses very similar local search plan generation method[14] to VIATRA. The plan graph (search graph by VIATRA) is built in a similar way, but lookup operation for pattern edges is added. The cost model is built based on statistics collected from the host graph (model) just before the execution of the transformation. The costs are added, the MST calculated, and search plan is determined in a way similar to VIATRA.

Fujaba uses less advanced local search plan strategy[15]. The pattern matching in Fujaba is started from the bounded element (the requirement in Fujaba is to have at least one per pattern). If there is more than one bounded element, one is chosen arbitrary. Links to follow are chosen by priorities using the first-fail principle. Regardless of simplicity of this algorithm, the benchmark tests[16] show that this strategy works almost as good as more advanced algorithms.

AGG uses an algorithm, equivalent to current pattern CSP[17]. This approach introduces variables for each pattern node and queries for each pattern edge forming the constraint graph. This graph is quite similar to the search graph in local search graph generation technique. Variable ordering used in the area of CSP is essentially the same as the concept of the search planning.

The popular model transformation languages ATL[18] and MOF QVT[19] are not addressed here, because to our knowledge no pattern matching implementation details are available for them.

4 Pattern Matching in MOLA

The implementation of MOLA uses a lower level model transformation language L3 [20] as the target language for compilation. It provides the basic operations on models and its implementation is oriented to the maximum execution efficiency and performance. Of course, the performance depends also on model repository the language uses. Currently high efficiency repositories MIIREP [21] and JGraLab [22] are supported as well as Eclipse Modelling Framework (EMF). L3 language allows creating and deleting instances and links, updating attribute values, iterating through instances, and checking the values of attributes. L3 also includes the classical constructs like loops and branching. It is a purely imperative language where patterns are also expressed in an imperative way. At the same time it is powerful enough to serve as a target language for MOLA compiler.

The basic model lookup operation (*first*) in L3 allows binding a pointer to an instance of the appropriate class. This operation creates an iterator which allows going through all instances of this class. What is more important, L3 allows creating an iterator from already bound instance using navigation by some association. Additionally, constraints can be specified that narrows the set of instances to be iterated over, e.g. constraint on attribute value (*attr*) or existence of link (*link*) to the already bound instance. In fact, there is the hidden backtracking step. The composition of basic lookup operations is allowed. Let us explain the following L3 code:

```
first z:Z suchthat
   first x:X from z by R suchthat attr x.a<10;
```

It means - bind an instance of class "Z" to the pointer "z" if there is an instance of class "X" connected to "z" by link of type "R" and it has the value of attribute "a" less than 10. The *first* command iterates through all instances of the given type until an acceptable instance is found. The nested *first* commands are used to form advanced constraints. Failure of a nested *first-suchthat* command causes a backtracking step. In such a classical way all potential matches are examined until one which fits is found.

Pattern matching implementations of other transformation languages also use similar basic lookup operations. For example, the *first-suchthat* command corresponds to the *GetInstance* operation used by PROGRES[9] or the *lkp* operation used by GrGen[14].

The most obvious way to compile a MOLA pattern to these operations is to start from one (chosen by some algorithm) class element and traverse the pattern graph. The result of such compilation is a first command created for the initial class element and nested first commands for other class elements. It is obvious that the same pattern can be matched in different ways using the basic lookup operations. Finding the most efficient way (the optimal search plan) is the main task for pattern matching.

4.1 The Simple Pattern Matching Algorithm for MOLA

As it was mentioned in the previous section, implementation of pattern matching for MOLA uses the local search planning strategy. This is one of the most popular strategies, however typically it requires sophisticated analysis of pattern or even underlying model to choose the best search plan. We propose a simple algorithm (in the sense of how complex is the implementation) which is efficient for the typical MOLA patterns used in MDSD-related tasks (it is efficient also for others if appropriate constructs are used). The simple algorithm uses the following principles:

– if the pattern contains a reference class element, then the pattern matching starts from the reference (if there are more than one, then an arbitrary is chosen).

- otherwise the pattern matching starts from the loop variable in the loophead or from arbitrary chosen element in the normal rule.
- the pattern matching is continued with class elements accessible from already traversed class elements by association links.

Pattern matching in a regular rule is started from the reference class element, if such class element exists in the pattern. Though MOLA does not require the presence of a reference class element in the pattern, the practical usage of MOLA has shown that most of the regular rules contain it. It is because the usage of imperative control structures causes reuse of the previously matched instances, which are represented by the reference class elements in MOLA. This is one of the main reasons why such simple optimization technique works almost as well as more sophisticated approaches.

Use of reference class elements is natural also in loopheads. It is common to have a loop over, for example, all properties of a given class. This task can be easily described, using a single MOLA loop, where the pattern in the loophead is given using the reference class element and the loop variable. See the loophead of the inner loop in Figure 2 for the typical case. In this case the pattern matching is started from the referenced element (*@pack*) reducing the search space dramatically. Of course, the path from the reference class element to the loop variable may be longer. The only restriction is that cardinalities of associations along the path (except one directly before the loop variable) should be "1" or "0..1".

For foreach loop statements without reference in the loophead, pattern matching is started from the loop variable in the loophead. Practical usage of MOLA has shown that typical tasks are naturally programmed using patterns, where cardinalities of association links leading from loop variable are "1" or "0..1". This causes the execution of the loop to work in a linear time dependant on the number of the instances corresponding to the loop variable. Of course, this does not apply for every example, but if an appropriate metamodelling (UML-like, using composition hierarchy) and imperative algorithms are used, then this condition holds for most cases.

Note the loophead of the outer loop in Figure 2. Though cardinalities of association links leading from the loop variable are "0..*", the pattern matching started from loop variable is still efficient. Since class elements other than loop variable provide the "existence semantics" (find first valid match), in practice this loop works also in linear time because almost all requirements are described using scenarios. In fact, this additional constraint is used to filter out those few cases where requirements are described using different means.

Note that this strategy does not even require analysis of the cardinalities of metamodel elements at the same time remaining efficient in the practical usage. A similar pattern matching strategy is used also by Fujaba. The bound variable (reference class element in terms of MOLA), is even required by the pattern in Fujaba. However, the benchmark tests [16] have shown that this strategy performs as well as more sophisticated strategies. The same tests also have shown that an appropriate usage of the language constructs (improvement of Fujaba transformation) causes significant positive impact on the performance. The same holds also for MOLA, however the feature which distinguishes both languages is the loop variable in the MOLA foreach loop. First of all, the transformation becomes more readable for human reader, secondly, it gives slight advantage in the performance of the pattern matching. It allows iterating through the instances corresponding to the loop variable only, while other patterns elements are checked just for the existence. On the contrary, Fujaba is forced to examine corresponding instances to all pattern elements in the foreach loop.

4.2 Empirical Study of Pattern Matching Cases in ReDSeeDS Project

In this section the analysis of typical patterns in the ReDSeeDS project is done. As it was mentioned before, one of the goals of the project is model driven software development using RSL and UML languages. The main idea is to obtain a part of the software system automatically from requirement specification using model transformations. During the ReDSeeDS project two model-based methodologies have been proposed and corresponding sets of transformations in MOLA developed.

To approximately estimate the volume of the transformations written during the ReDSeeDS project we are giving some statistics. The model-based methodologies used in the project cover quite a large subset of UML being generated - UML class, activity, component and sequence diagrams are being generated. Both methodologies include several transformation steps. The first step for both methodologies is the transformation of requirements. The next steps are generating new UML models adding more specific details. There are ∼350 MOLA procedures developed during the ReDSeeDS project. They include ∼200 loops and ∼800 rules that gives ∼1000 pattern specifications. We have investigated the structure of patterns used in the project and most of them are fit to the simple pattern matching strategy used by MOLA.

Figure 2 refers to the typical usage of loops in ReDSeeDS project - the model driven software development tasks are *compilation*-like jobs where every element of the source model is processed and corresponding elements in the target model are created. Since RSL and UML model elements form tree-based hierarchy, the transformation algorithms traverse model elements in the top-down style starting from the top elements of the hierarchy. Therefore, the most natural way to describe such traversing is by using nested foreach loops referencing the previous loop variables. The pattern may contain additional class elements for collection of all necessary neighbourhood instances or specifying additional constraints on the existence of appropriate nearby instances.

Another typical pattern used in the ReDSeeDS project is depicted in Figure 3. This pattern finds the name of an actor (names are coded as noun phrases in RSL). Note, that all associations leading from the Actor class to the Noun have cardinality "1" or "0..1" - each actor has exactly one name (represented by noun phrase), there is only one noun link for each noun phrase and every noun link is connected to exactly one noun. Therefore this pattern is matched in constant

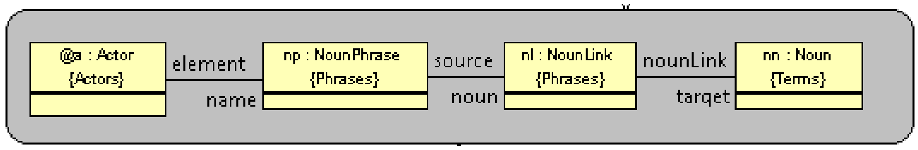

Fig. 3. Pattern example - collecting nearby instances

time when the simple pattern matching strategy is applied. This is a typical case where MOLA rule is used to collect the nearby instances.

A variation of the previous pattern is shown in Figure 4. This pattern describes the collecting of nearby elements of a UML interaction. The owning classifier and component corresponding to the lifeline named "UIComponent" should be matched. Unlike in the previous example there is an association with cardinality "*" leading from the referenced element (to Lifeline). However, as we see in practice, typically there is only one model element in the model satisfying the given constraint and the "suspicious" association has low cardinality in practice. In this case there are no more than 5-10 lifelines per interaction. Thus this pattern matches in linear time with regard to the number of lifelines in the given interaction, which is relatively low.

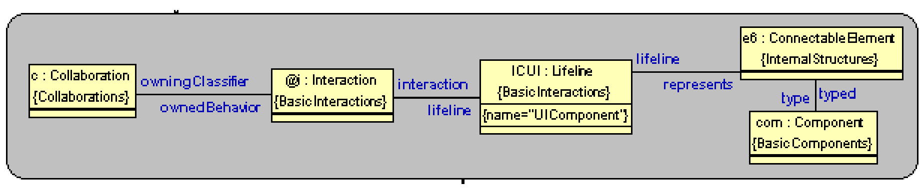

Fig. 4. Pattern example - collecting nearby instances using additional constraints

We have tested the transformations on several sufficiently large software cases developed within the ReDSeeDS project. The total time of transformations execution turns out to be almost linear with regard to the total number of constrained language sentences in the requirement scenarios specified in the RSL for the case. The patterns described above are the most typical patterns used in MOLA transformations for the ReDSeeDS project. The total amount of such patterns is about 95% of all patterns. Some specific sub-tasks require non-typical patterns which theoretically may cause insufficient pattern matching performance, however in practice they are performed on elements which are relatively low in number compared to the number of constrained language sentences. Thus, they do not affect the overall performance of pattern matching.

There was made a conjecture that a transformation program in MOLA written in an appropriate style becomes efficient at the same time [23]. Our empirical analysis of typical patterns in the ReDSeeDS project confirms that this holds also in praxis and MOLA is a suitable model transformation language for MDSD-related tasks. In this case the simple pattern matching algorithm gives very efficient results.

4.3 Local Search Planning Using Annotated Metamodels

MOLA language can be used not only in the MDSD-like domains, where patterns are similar to those described in the previous section, but also in others. A more advanced pattern matching technology should be used to support efficient matching of these patterns. The classical local search planning approach is used in MOLA for these cases. We are also building a local search graph and calculating minimal spanning tree. The analysis of corresponding metamodel fragment is made for each pattern to get weights for edges of the local search graph. In general, our approach is similar to the one used by PROGRES and it is currently implemented partially.

The search algorithm described above optimizes the search plan selection using only data from the metamodel and pattern specification. Other approaches that are based on the statistical analysis of the model collect actual cardinalities for classes and associations (the number of instances of the given class in the model). These approaches give very efficient results, however there are situations where such analysis can not be made (e.g. runtime repository does not support the required statistics for runtime analysis or there are no models created yet in the case of offline analysis). Therefore we propose an approach which allows using developer's knowledge of model constraints that otherwise could be obtained only by analysis of existing models. A part of actual cardinalities can be already predicted at the design time of a transformation. Development of a transformation requires a good knowledge of the corresponding domain. Therefore, the transformation developer should be able to define prospective cardinalities. Of course, precise number of the instances cannot be predicted, except for singleton classes. However, the proportion of instances for different classes is frequently known. For example, the number of properties in UML model is several times greater than the number of classes. Since neither metamodelling standard MOF, nor UML class diagrams provide convenient means for the specification of the prospective cardinalities, we propose to annotate the metamodel and patterns in MOLA. Our goal is to have a simple, handy annotation mechanism that helps to select an efficient search plan for the pattern matching.

We allow annotating classes and association ends in the metamodel and class elements and association link ends in patterns. An annotation predicts the number of instances for classes and the number of instances reachable by links for association ends. Pattern matching algorithm takes into account the annotations, and edge weights in the search graph are adjusted accordingly. In fact, an annotation sets the priority on the pattern element. The lower the predicted number of instances is for the pattern element, the higher priority it gets for the pattern matching. Annotations made in the metamodel affect the pattern matching algorithm in every rule where pattern elements of the corresponding type are used. Annotations made in the pattern affect the pattern matching algorithm only in the scope of the rule. The developer annotates metamodel elements during the development process of the metamodel. Since metamodelling requires the knowledge of the modelled domain, typically there are no problems to resolve actual cardinalities. It should be noted that the annotations are optional - they are additional means to improve the efficiency of transformations. The following annotations can be used:

SINGLE - denotes that the class (or navigation result) has at most one instance. Such instances and links as well as references are preferred for the pattern matching.

FEW - denotes that the class (or navigation result) has nearly constant number of instances, or it is relatively low compared to the total number of instances in the model. For example, we can expect that in a UML class diagram a typical class will have about 5-10 properties, and this number is independent of the model size. Such links will be preferred over links that are not annotated for the pattern matching.

MANY - denotes that the class (or navigation result) has a relatively large number of instances, and this number grows together with the size of the model. For example, in a UML class diagram the number of typed elements for every type grows as the size of the class diagram increases. Links that are not annotated will be preferred over links with the "MANY" annotation for the pattern matching.

Figure 5 shows a pattern in a loophead where annotations help to find the best search plan. This loop iterates trough every property (p) of the given class ($@c$) having the given type ($@t$). The problem is that associations *ownedAttribute* and *typed* both have cardinality "*" and without additional information both are treated equally (un)efficient for pattern matching. However, in practice the average number of owned attributes for a class is by magnitude less than typed properties for a type. Therefore, adding annotations FEW and MANY to *ownedAttribute* and *typed* association ends accordingly gives the desired result - the pattern matching is started from the reference $@c$.

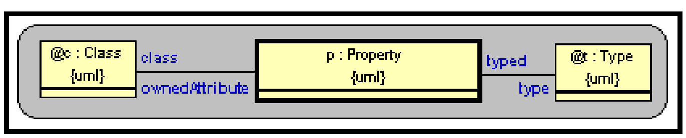

Fig. 5. Pattern example - annotation use case

5 Conclusions

We have made a review of pattern matching mechanisms for the most popular model transformation languages in this paper. There are several pattern matching approaches, but the most popular is the local search planning. In fact, it is the most universal strategy - it gives efficient results for different types of patterns. However, implementations of more advanced approaches are rather complex, although simpler strategies (like in case of Fujaba) frequently give similar results. Of course, that holds not for every use case, but mostly for **the domain** the transformation language is designed for. For example, MOLA is efficient for MDSD-related tasks, as the empirical analysis of typical MOLA patterns in the ReDSeeDS project has shown. Other languages are efficient in other domains, e.g. VIATRA in the simulation of complex systems or Fujaba in the program refactoring domain.

A great role for efficient pattern matching is plaid also by the constructs of the pattern used in the language. MOLA offers very natural means for describing MDSD-related tasks, the foreach loops combined with explicit reference mechanism. At the same time even the simple pattern matching algorithm which has been implemented for MOLA works efficiently in these cases. Thus, for the *compiler*-like tasks, where every element of a structured model (like UML) should be processed, MOLA can be used in a natural way with a high efficiency with very simple implementation of pattern matching.

MOLA is used not only for MDSD-related tasks(though it is designed for that). Therefore we have considered also more universal pattern matching strategy based on analysis of the pattern and underlying metamodel. The local plan search algorithm is partially implemented in the latest version. We have introduced the metamodel annotation mechanism, which captures the domain knowledge of actual cardinalities in the metamodel. It permits to make pattern matching more efficient, that could be achieved only by runtime analysis of models which may itself be costly at runtime or not available at design time.

The future work is to identify model transformation domains - the areas where typical patterns are used. The most appropriate pattern matching approaches should be addressed for each domain. That would make the choice of appropriate model transformation language easier for a concrete task.

References

1. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. In Aß mann, U., Aksit, M., Rensink, A., eds.: Proceedings of MDAFA. Volume 3599 of LNCS., Springer (2004) 62–76
2. Schürr, A., Winter, A.J., Zündorf, A. In: The PROGRES approach: language and environment. Volume 2. World Scientific Publishing Co. (1999) 487–550
3. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: Proceedings of 17th IEEE International Conference on Automated Software Engineering, IEEE Comput. Soc (2002) 267–270
4. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: Grgen: A fast SPO-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Proceedings of ICGT. Volume 4178 of LNCS., Springer (2006) 383–397
5. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Ehrig,

H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Proceedings of TAGT. Volume 1764 of LNCS., Springer (1998) 296–309
6. Śmiaek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T.: Complementary use case scenario representations based on domain vocabularies. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Proceedings of MoDELS. Volume 4735 of LNCS., Berlin, Heidelberg, Springer (2007) 544–558
7. Taentzer, G.: AGG: A Tool Environment for Algebraic Graph Transformation. In Nagl, M., Schürr, A., Münch, M., eds.: Proceedings of AGTIVE. Volume 1779 of LNCS., Springer (1999) 481–488
8. Dechter, R., van Beek, P.: Local and global relational consistency. Theoretical Computer Science **173**(1) (1997) 283–308
9. Zündorf, A.: Graph Pattern Matching in PROGRES. In Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G., eds.: Proceedings of ICGT. Volume 1073 of LNCS., Springer (1994) 454–468
10. Varró, G., Friedl, K., Varró, D.: Implementing a Graph Transformation Engine in Relational Databases. Software & Systems Modeling **5**(3) (2006) 313–341
11. Varro, G., Friedl, K., Varro, D.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. Electronic Notes in Theoretical Computer Science **152** (2006) 191–205
12. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: Proceedings of GraMoT, ACM (2008) 25–32
13. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: Efficient Model Transformations by Combining Pattern Matching Strategies. In Paige, R.F., ed.: Proceedings of ICMT. Volume 5563 of LNCS., Springer (2009) 20–34
14. Batz, G.V., Kroll, M., Geiß, R.: A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proceedings of AGTIVE. Volume 5088 of LNCS., Springer (2008) 471–486
15. Fischer, T., Niere, J., Torunski, L.: Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML. Master thesis, University of Padeborn (1998)
16. Geiß, R., Kroll, M.: On Improvements of the Varro Benchmark for Graph Transformation Tools (2007)
17. Rudolf, M.: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Proceedings of TAGT. Volume 1764 of LNCS., Springer (1998) 238–251
18. Jouault, F., Kurtev, I.: Transforming Models with ATL. In Bruel, J.M., ed.: Proceedings of MoDELS. Volume 3844 of LNCS., Springer (2006) 128–138
19. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.0 (2008)
20. Barzdins, J., Kalnins, A., Rencis, E., Rikacovs, S.: Model Transformation Languages and Their Implementation by Bootstrapping Method. In Avron, A., Dershowitz, N., Rabinovich, A., eds.: Pillars of Computer Science. Volume 4800 of LNCS., Springer (2008) 130–145
21. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards Semantic Latvia. In Vasilecas, O., ed.: Proceedings of DB&IS, Vilnius, Technika (2006) 203–218
22. Kahle, S.: JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Master thesis, University of Koblenz-Landau, Institute for Software Technology (2006)
23. Kalnins, A., Barzdins, J., Celms, E.: Efficiency Problems in MOLA Implementation. In: Proceedings of ooPSLA. (2004)