Efficient Model Transformations by Combining Pattern Matching Strategies

Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{bergmann, ahorvath, rath, varro}@mit.bme.hu

Abstract. Recent advances in graph pattern matching techniques have demonstrated at various tool contests that graph transformation tools can scale up to handle very large models in model transformation problems. In case of *local-search* based techniques, pattern matching is driven by a search plan, which provides an optimal ordering for traversing and matching nodes and edges of a graph pattern. In case of *incremental pattern matching*, matches of a pattern are explicitly stored and incrementally maintained upon model manipulation, which frequently provides significant speed-up but with increased memory consumption. In the current paper, we present a *hybrid pattern matching* approach, which is able to combine local-search and incremental techniques on a per-pattern basis. Based upon experimental evaluation, we identify scenarios when such combination is highly beneficial, and provide guidelines for transformation designers for optimal selection of pattern matching strategy.

1 Introduction

Model transformations play a crucial role in modern model-driven system engineering, an application domain where transformations need to handle large, industrial models in a short amount of time. *Graph transformation* (GT) [1] based tools have been frequently used for capturing and executing complex transformations. In GT tools, *graph patterns* capture structural conditions and type constraints in a compact visual way. During transformation time, these conditions are evaluated during *graph pattern matching*, which aims to derive one or all matches of a given pattern to execute a transformation rule.

Empirical evidence reported at recent tool contests [2, 3] proved that GT tools scale up for transforming very large models, thanks to highly sophisticated, *local-search based graph pattern matching* algorithms proposed in transformation tools such as GrGEN.NET [4], FUJABA [5], and VIATRA2 [6]. In all these approaches, pattern matching is driven by a search plan, which provides an optimal ordering for traversing and matching nodes and edges of a graph pattern.

As an alternative, incremental pattern matching (INC) approaches [7–11] have recently become a hot topic in the model transformation community. The core idea is to improve the execution time of the time-consuming pattern matching phase by additional memory consumption. Essentially, the (partial) matches of graph patterns are stored explicitly, and these match sets are updated incrementally in accordance with elementary model changes. While model manipulation becomes slightly more complex, all matches of a graph pattern can be retrieved in constant time in exchange by eliminating the need for recomputing existing matches.

Initial benchmarking [12] has shown that in many scenarios, the incremental pattern matching approach (as implemented in the VIATRA2 framework) leads to orders-of-magnitude increases in speed. However, an important implication of caching match sets is increased memory consumption, which needs to be taken into account when scaling up to large models. Unfortunately, in many practical applications of model transformations, available memory is frequently constrained (e.g. when they are executed on average desktop computers and not on high performance servers).

In the current paper, we propose a hybrid pattern matching approach which enables the transformation designer to combine local search-based and incremental pattern matching to adapt to memory constraints. At design-time, transformation engineers may select whether a graph pattern should be matched using the LS or the INC strategy separately for each pattern. Moreover, based upon runtime monitoring, the execution engine may automatically switch from incremental pattern matching to local-search based technique when a certain memory limit has been reached.

Additionally, we present experiments to demonstrate that the incremental strategy is not always the best choice for pattern matching: we highlight scenarios using a performance benchmark of model transformations (object-relational mapping) where a combination of INC with LS significantly outperforms the plain INC-only and LS-only versions. By the analyzing results of our case study, we provide a list of various factors (metrics) which we experienced to have significant factor on performance, and give hints to transformation designers when a graph pattern should be matched using an INC or an LS strategy in each case.

While the direct contributions of the paper are dedicated to graph transformation-based approaches of model transformations, we believe that the conceptual foundations are, in fact, adaptable to other transformation techniques. For instance, similar investigations can be carried out in the future to assess when an OCL constraint should be evaluated incrementally, and when an evaluation should be initiated from scratch.

The rest of the paper is structured as follows. Section 2 briefly introduces graph patterns and graph transformation rules (as available in the VIATRA2 transformation language). It also describes the object-relational mapping used as a performance benchmark throughout the paper. As related work, we highlight main characteristics of the local search-based and incremental pattern matcher implementations. In Section 3, we present scenarios to highlight when a hybrid pattern matching strategy provides significant performance advantage in typical model transformations. We present metrics to optimally select LS or INC strategies for patterns at design time. Moreover, we present an adaptive runtime technique to switch to LS strategy when memory is low (Section 4). Finally, Section 5 concludes the paper.

2 Background

2.1 Graph Patterns and Transformation

Graph patterns are frequently considered as the atomic units of model transformations [13]. They represent conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of (i) *structural constraints* prescribing the existence of nodes and edges of a given type, and (ii) *containment constraints* specifying containment relation between nodes of graph patterns. A *negative application condition* (NAC), defined by a negative subpattern, prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match.

Graph transformation (GT) [1] provides a high-level rule and pattern-based manipulation language for graph models. Graph transformation rules can be specified by using a left-hand side – LHS (or precondition) pattern determining the applicability of the rule, and a right-hand side – RHS (postcondition) pattern which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged.

In the following, we use the language of the VIATRA2 framework [13] for demonstration purposes, which also provides additional *control structures* for assembling complex transformations defined by abstract state machines (ASM) [14].

Example 1. The graphical representation of an example graph transformation rule, along with the VIATRA2 textual representation of the LHS, is depicted on Figure 1. Informally, the meaning of the *schemaRule* is to map a package *P* contained by the container *C* to a schema *S*, unless the mapping has already been performed. Elements labeled with *new* are the ones created by the rule; elements labeled with *neg* constitute the single NAC of the rule[1]. The NAC is used to exclude the cases when the mapping has been performed; it applies if there is an edge *R* of type schemaRef between the nodes *P* and *SN*.

```
gtrule schemaRule(P,Con) =
{
  package(P) below Con;
  neg pattern mapped(P, SN, RN) = {
    package(P);
    schema(SN);
    package.schemaRef(RN, P, SN);
  }
 new schema(SN);
 new package.schemaRef(RN, P, SN);
}
```

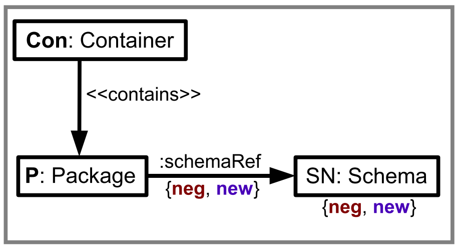

Fig. 1. GT rule for unmapped packages below a container

[1] we do not display rules with multiple NACs or with deletion action in this paper

2.2 Case study

Transformation overview. The (simplified) Object-to-Relational schema mapping (ORM) case study was proposed as a performance benchmark of model synchronization transformations in [12, 15]. The aim of the transformation is to produce corresponding relational database schemas from UML class diagrams according to the following mapping rules:

– First, a relational schema is created for the specified package below a given container by *schemaRule* (Fig. 1). Transitive containment is represented by an edged tagged as "contains".
– Then classes in the package are mapped into tables in the corresponding schema, each with an id column as a primary key (*classRule*).
– Each association in the package is mapped into a table in the corresponding schema with a primary key column (*associationRule*);
– Each association end in the association is mapped into a foreign key of the corresponding table pointing at the table generated from the class that that the association end points to (*assocEndRule*);
– Attributes in the class are mapped into columns of the corresponding table (see *attributeRule* in Fig. 3).

In incremental synchronization, to avoid rebuilding target models in each pass, a *reference model* (also known as trace / correspondence model) is used to establish a mapping between source and corresponding target model elements (Fig. 2) and to trace changes in them. In this context, the reference model is often referenced in NACs to identify elements in the source model that have not yet been mapped into the target model. In the current paper, we restrict our investigations to one-way synchronization.

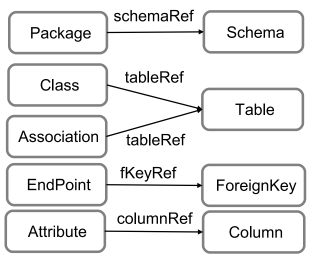

Fig. 2. Reference metamodel

```
gtrule attributeRule(C, A, T) =
{
  class(C);
  class.attribute(A);
  class.cl2attrs(CF1, C, A);
  table(T);
  class.tableRef(R1, C, T);
  neg pattern mapped(A, ColN, RN) = {
    class.attribute(A);
    column(ColN);
    class.attribute.colRef(RN, A, ColN);
  }
 new column(ColN);
 new class.attribute.colRef(RN, A, ColN);
}
```

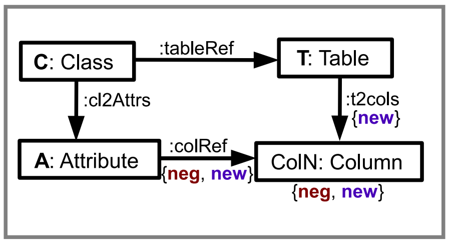

Fig. 3. GT rule for unmapped attributes

Transformation scenario. In the original benchmark example [12], the source model consists of two packages, both containing a (generated) set of classes with attributes. First, (i) the *primary* package will be mapped into a relational schema to create initial mappings. Then, the *source models are modified*, and, in an additional pass, (ii) the system has to *synchronize the changes to the target model* (i.e. find the changes in the source and alter the target accordingly). This scenario is now extended as follows.

Check phase First, we check as a precondition of the transformation that no generalization exists in the source UML model to ensure the applicability of the transformation. It is captured by a corresponding simple graph pattern (*detectGeneralization* in Fig. 4). We intentionally omitted support for inheritance from the model transformation itself to analyze a case where a transformation has to perform a preliminary applicability check; the practical consequences of this choice will be assessed later in Sec. 3.2.

Initial transformation phase When there are no generalizations, the primary package is mapped into a relational schema by the transformation program.

Refactoring phase A refactoring operation modifies the package hierarchy of the source model, namely the secondary package is moved inside the primary package.

Synchronization phase Afterwards, synchronization propagates these changes into the target relational model so that it holds the mapping of the (changed) primary package once again. This involves creating the tables for classes that stem from the secondary package, and creating columns for attributes of these classes.

```
pattern detectGeneralization(Sub,Sup) =
{
  general(Gen);
  class(Sup);
  general.parent(PE,Gen,Sup);
  class(Sub);
  general.child(CE,Gen,Sub);
}
```

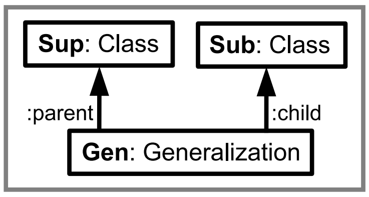

Fig. 4. Graph pattern checking for generalizations

2.3 Pattern matching strategies and related work

Pattern matching plays a key role in the efficient execution of all model transformation engines. In case of graph transformation based approaches, the goal is to find the occurrences of a graph pattern, which contains structural as well as type constraints on model elements. During pattern matching, each variable of a graph pattern is bound to a node in the model such that this matching (binding) is consistent with edge labels, and source and target nodes of the model.

Most model transformation approaches (e.g. [4, 5, 16, 17] and many more) usually rely on a *local search based pattern matching* (LS) that starts the matching process from a single node and extends it step-by-step by neighboring nodes and edges. Informally, a search plan [6, 18] defines an ordering of pattern nodes, in which they are bound to objects of the instance model during pattern matching. With efficient search plans ([4, 19]), LS strategies can produce good runtime performance with a relatively small memory footprint, and low update complexity. Other approaches [20, 21] use constraint satisfaction techniques for matching graph patterns.

As an alternate approach, *incremental pattern matching* (INC) [7, 8, 10, 11] relies on a *cache* in which the matches of a pattern are stored explicitly. The match set is readily available from the cache at any time without searching, and the cache is incrementally updated whenever changes are made to the model. As pattern occurrences are stored, they can be retrieved in constant time – excluding the linear cost induced by the size of the result set itself –, making pattern matching extremely fast. The trade-off is increased memory consumption, and and increased update costs (required to continuously maintain the stored match set caches.

In the current paper, our goal is to investigate if (and when) the combination of pattern matching strategies within a transformation (referred to as *hybrid pattern matching*) can provide better runtime performance, especially, with constraints on available resources (such as memory consumption). For our investigations, we use the VIATRA2 framework, which supports both pattern matching engine strategies and allows to specify the use of INC or LS strategy separately for each graph pattern.

There are cases where the use of either the incremental or the local search based pattern matching approach is significantly more efficient than the other. We argue that many transformations could benefit even more from combining these two approaches, by using different pattern matcher engines for different graph patterns. As a conceptual analogy for our current work, recent research in expert systems [22] demonstrated that an integration between two different incremental strategies can be advantageous.

3 Motivating scenarios for hybrid pattern matching

Recent benchmarks evaluations [12] and tool contests [3] in the graph transformation community have shown that INC can easily be orders of magnitude faster than (most) LS approaches for certain problem classes. This section identifies three scenarios where, on the other hand, LS has a clear advantage, as demonstrated by our experiments[2]. For each scenario, we identify a hybrid pattern matching approach where some patterns and transformations should use LS, while the rest of the transformation relies upon INC to obtain a better performance than the two extremes (LS-only or INC-only).

3.1 Scenario: match set cache does not fit into memory limit

This scenario demonstrates that the high memory consumption of incrementally maintained caches can be a bottleneck of INC. By choosing LS for patterns that are memory-intensive (i.e. with many matches) but not time-critical, the high memory consumption can be greatly reduced, while still retaining the short execution time comparable to INC.

[2] Measurement environment: Intel Core Duo t2400@1,83 GHz processor, 3 GB RAM, Windows XP SP3, Sun HotSpot Java 1.6.0_02 and VIATRA2 Release 3 build 2009.02.03.

Our experiments were performed on the *Transform phase* of the ORM case study (see section 2.2), by measuring the heap commit size of the Java VM. In the followings, we model a frequently occurring development scenario. As the transformation designer is typically working with small toy models, scaling up to large model sizes might lead to unexpected results. For instance, while a toy model with 10 classes and 250 attributes and the corresponding INC cache easily fits in a few megabytes, a memory usage of 128MBs can be reached by increasing the model to 575 classes and 14375 attributes, as shown on Table 1. With a memory limit of 128M, as the match set cache expands rapidly, the JVM begins to trash due to memory starvation shortly after the transformation is started. This leads to significant slowdown (to 21 seconds), and may even result in a failed execution because of heap exhaustion. If the amount of memory is suitably large (i.e. 1GB in our case), execution is very fast (4.6 seconds). LS is not an alternative here: while the memory consumption of the caches is spared, the execution time for this model size is very long (avg. 184 seconds).

Closely observing this transformation, we may identify LHS pattern *attributeRule* (see Figure 3) and its embedded negative application condition as patterns with high number of occurrences. By sacrificing execution time (runs in 5.0s with a 1G heap), we marked this pattern to be matched by the LS engine, despite using INC for the rest of the transformation. This reduced memory consumption to 105M, and allowed the transformation to run with approximately the same execution time (5.6s) even with a memory limit of 128M. Therefore the hybrid approach has the potential to efficiently scale up to higher model sizes given the same memory constraints.

PM Strategy	Memory limit [MB]	Used heap [MB]	Transform phase time [ms]
LS	128	99	183729
INC	128	128	21057
INC	1024	128	4639
Hybrid	128	105	5573

Table 1. Match Set Memory and Performance

3.2 Scenario: construction time penalty

This scenario emphasises that the time required to initialize the incrementally maintained caches might itself be too expensive. The construction time of the caches is not less than the time required to find all occurrences of the pattern, since the match set is directly available from this cache. If the transformation needs to find only one (or few) of many pattern occurrences altogether, there is no need for LS to continue the search and retrieve the entire match set, therefore it can be significantly faster than INC. This phenomenon only applies if the pattern is efficiently matchable by LS, unlike large patterns with high combinatorial complexity.

This behaviour was observed in the *Check phase* of the ORM case study (see section 2.2). We measured the time it takes to find an arbitrary generalization edge if all 2500 generated classes inherit a common superclass, which is a single-occurrence query of a very simple graph pattern (see Figure 4) consisting of a single edge. The measurements show (in Table 2) that constructing the cache took 0.14s on average, while INC would gain only about 16 ms time (too small to be more accurately measured) compared to LS with each further query (if there were any). To complement these results, we also took the measurement on a source model without generalization, for which the transformation could be performed; we found that, in accordance with expectations, LS has no significant advantage in this case: constructing the cache took 16 ms, while LS needed 36 ms to complete query (both of them too small to be accurately measured).

PM Strategy	Used heap [MB]	Cache construction time [ms]
with generalization edges		
LS	152	10
INC	159	143
without generalization edges		
LS	147	36
INC	149	16

Table 2. Construction Time Performance

3.3 Scenario: expensive model updates

This scenario happens if there is heavy model manipulation between infrequent pattern queries. In this case, the time overhead imposed on model manipulation by INC may outweigh its benefits. The cost of incrementally maintaining the match set caches for a long period of time with frequent model updates may be larger than the cost of applying LS and calculating the match set from scratch at each pattern query. In other terms, it may be superfluous to continuously maintain the match sets if they are not frequently used for model queries.

Expensive update overhead is observable in the *Refactoring phase* of the ORM case study (see section 2.2). We measured the time it takes to move a package in the source model to a different package, while the INC maintains the caches of patterns that represent the location of classes in the namespace hierarchy of packages, classes and attributes (Table 3). The transitive containment is a model feature of high combinatorial complexity, and moving a high-level element will cause drastic changes in this relationship, thereby forcing the INC to perform intensive cache updates. The measurements have shown that the cost of the single move operation can be as high as 2.1 seconds with INC. Using pure LS was not a feasible solution either, as the *Synchronization phase* did not terminate within half an hour. A hybrid pattern matcher assignment solved these problems: patterns using the transitive containment (see Figure 1) were matched by LS and the rest by INC, resulting in a fast move operation and an execution time of 14.5 ms for the entire *Refactoring phase*. These measurements were taken with both the primary and secondary packages consisting of 1000 classes and 25000 attributes.

PM Strategy	Used heap [MB]	Refactoring phase time [ms]	Synchronization phase time [ms]
LS	-	0	$>$2000000
INC	493	2109	13386
Hybrid	298	0	13570

Table 3. Model Update Performance

3.4 Overall performance on the entire case study

Finally, we compare the overall performance of the three approaches on all three steps of the case study combined. Measurements were taken for various source model sizes, scaling up until the transformation became too slow (LS) or did not fit into memory (INC, hybrid). Figure 5 indicates the total execution time versus the number of classes in the primary source package. For these measurements, the number of classes in the secondary package (N/4) was always one quarter of the number of classes initially in the primary package (N), and each class still had 25 attributes (25N, 25N/4); thus the largest case (N=2400) consisted of 2400 classes and 60000 attributes in the primary package, 600 classes and 15000 attributes in the secondary package, i.e. more than 150 000 source model elements altogether including edges. As the figure shows, INC scales up higher than LS, but the hybrid approach is even more efficient.

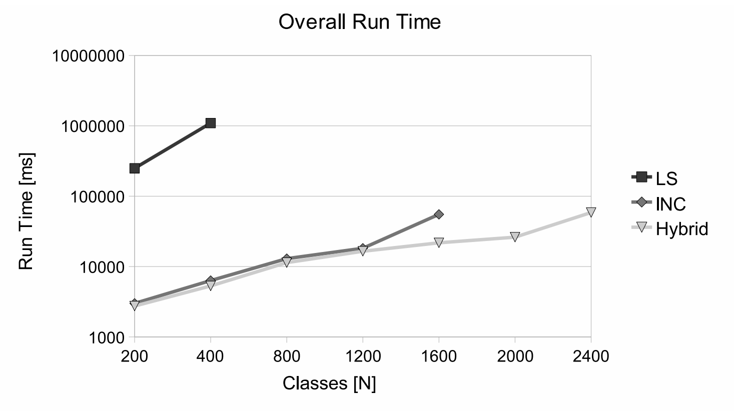

Fig. 5. Overall Execution Time

4 Towards intelligent selection of matching strategies

In this section, we first *identify various factors* (qualitative metrics) which help transformation designers decide when a certain pattern matching strategy (LS or INC) would be beneficial (Sec. 4.1). Then, in Sec. 4.2, we discuss how an *adaptive run-time behaviour*

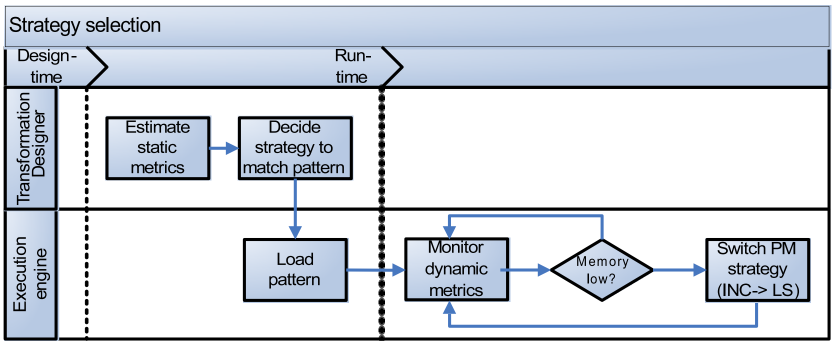

Fig. 6. Selecting pattern matching strategies at design-time and runtime

can be obtained by monitoring relevant metrics, and switching from one strategy to the other at runtime. Compared to existing adaptive pattern matching solutions [4, 19], the main novelty of this approach lies in the fact that we are able to *automatically switch between two entirely different pattern matching strategies* to increase performance. The high-level workflow of these techniques is illustrated in Fig. 6.

As identified in Sec. 3, several factors may influence the behaviour of the pattern matching algorithms. *Static factors* like (i) *static attributes of graph patterns* (e.g. pattern size, fan-out, structural complexity) and (ii) *control structures* of model transformations (e.g. forall, iterate) determine operative characteristics which, in combination with the characteristics with the different pattern matcher strategies, greatly influence the cost of pattern matching.

In contrast, *dynamic factors* change in-between transformation runs on the same system, and also with different target execution platforms: (iii) *model-specific graph characteristics* like qualitative attributes related to structure (e.g. average fan-out) and quantitative parameters related to model size (e.g. total number of model elements) may change as the transformation is changing the underlying model. Moreover, (iv) *memory limitations* impose external constraints which are related to the execution environment.

4.1 Factors for design-time selection of matching strategies

In the VIATRA2 framework, transformation designer can fine-tune the performance of graph pattern matching by prefixing a graph pattern with `@incremental` or `@localsearch` annotations to select the designated pattern matching strategy.

Based on our previous experience with performance benchmark transformations [12] and practical model transformations of large complexity [23], we identified the following factors to be important for transformation designers to choose between LS and INC strategies:

(i) *Graph pattern static attributes*
- *number of graph patterns* in a transformation program has a huge impact on the memory consumption – especially in resource constrained environments like embedded systems. The cache size of the pattern increases memory consumption when matched by INC strategy.
- *pattern size*, in practical applications, we experienced that the number of matches gradually decrease as the pattern to be matched becomes more and more complex (contradicting the theoretical complexity, which predicts that large patterns will have more matches. As a result, large patterns should be preferably matched by INC.
- *containment hierarchy constraints*, especially transitive containment, may significantly increase the memory consumption of incremental pattern matching due to fact that all containment relation between model elements have to be *cached* and incrementally updated. A good compromise could be to decompose the pattern and match only the containment constrained part with the LS engine while leaving INC strategy for the rest. Another solution would be to refactor patterns (and possibly the model also) so that they use explicit graph edges instead of relying on the implicit containment hierarchy.

(ii) *Control structures*
- *parameter passing* is using the result of rules or patterns as an input of other rules or patterns. This technique increases efficiency in LS as search operations are much more efficient if one or more pattern variables are bound, i.e. their values are known at time of the query. INC performance is not affected.
- *usage frequency* of patterns is relevant, since the more often a pattern is used, the more advantage INC has. Frequently used patterns can be identified by static analysis of the transformation code, e.g. by marking patterns that are used from within a loop. Trace analysis can yield more valuable estimates, if typical example inputs are available, by executing the transformation on these inputs and counting the times each pattern are accessed.
- *model update cost*: if program code analysis can reveal that model element types belonging to a certain pattern are rarely (or never) manipulated, the model manipulation costs imposed by INC can be neglected.

(iii) *Model dependent pattern characteristics*
- *node type complexity*, a rough upper bound on the number of potential matches can be obtained as the product of the cardinalities (number of model instances) of the types of each node in the graph pattern. This estimate is, of course, accurate as there are also edges in the pattern to constrain the possible combinations of nodes. However, high complexity may result in high memory consumption for INC, and long search operations for LS.
- *model statistics* generally extend graph pattern static attributes to the entire instance model the transformation is working on. A well-known practical statistics on pattern complexity is the *search space tree cost*, that has already been used to adaptively select the search plan for LS-based matchers [19]. It uses model statistics to assess the branching factors (node type complexity) during the search process. Other important factors like fan-out, hierarchy depth and model symmetries can also effectively make the estimation of match set sizes and time complexity of the pattern matching more precise.

By evaluating these (qualitative) metrics on the ORM case study described in Sec. 2.2, the observed behaviour in Scenarios 3.1–3.3 can be explained in more detail.

– In Sec. 3.1, we have identified the cause of the performance bottleneck to the *attributeRule* graph pattern with large match set. Since this pattern is used to filter for Attributes which have not yet been mapped to a Table column, it can be expected to have an initially large match set for class models with a large number of attributes. The match set size can be estimated a-priori by looking at *instance count* numbers for the Attribute type, or, by simply considering the general type composition characteristics of models the transformation is to be executed on.
– Sec. 3.2 demonstrated the usage simple pattern for structural checking (i.e. executing only once). This case corresponds to low pattern complexity and low usage count which, especially when combined with a potentially high match count, indicates a good candidate for switching to LS.
– Finally, Sec. 3.3 uses a pattern with a transitive containment constraint which, when used for synchronization after a model move high in the containment hierarchy, caused a drastic overhead for the incremental pattern matcher. As the resolution suggests, such patterns should generally be matched with LS.

4.2 Adaptive runtime optimization

Dynamic factors like memory consumption can quite easily change in-between transformation runs (even on the same system), especially using INC pattern matching, leading to performance degradation or insufficient memory. The current section focuses on an adaptive approach that can intervene in the predefined matching strategy in order to adapt to the altered environment.

In accordance with the general strategy described in Sec. 3, the adaptive engine generally prefers using the incremental pattern matcher for all graph patterns. When shortage of available memory is detected, pattern match set cache structures are gradually abandoned. For constructing such an adaptive approach monitoring, the following parameters are actually considered:

– During the execution of a VIATRA2 transformation the memory consumption is directly observable through the Java Virtual Machine (JVM), which provides a straightforward way for *monitoring available memory*.
– Simple *model space statistics* (e.g. the total number of model elements) are automatically registered by the VIATRA2 engine, along with sizes of match sets available from the incremental pattern matcherthat can also be used as a model-specific indicator for actual memory consumption and to dynamically detect situations where run-time adaptive matching selection strategy switching is needed.

For the actual strategy the priority order for the cache removal is determined by the *largest-first* principle, where the pattern match cache structure with the largest overall memory footprint is selected for removal resulting, that the forthcoming pattern match operation requested for the corresponding pattern will always be executed by the LS-based pattern matcher leading to a smaller memory consumption. In our case, memory shortage is detected when the available heap memory is less than 15%, which initiates dropping PM caches and switching to LS strategy.

In order to evaluate the efficiency and impact of this approach, we ran the benchmark experiment described in Sec. 3.1 with the adaptive implementation. The results for this measurement were obtained in a different software environment: we used the 64-bit version of IcedTea 1.3.1 as a JVM (hence the larger memory consumption figures). Execution times can be observed in Table 4.

PM strategy	Used heap [MB]	Transform phase time [ms]
LS	201	77054
INC	353	13693
Static hybrid	220*	10958
Adaptive hybrid	235*	35716

Table 4. Match Set Memory and Performance of the Adaptive Hybrid Strategy

Unsurprisingly, the execution time of the hybrid adaptive approach is between the fastest INC, the static hybrid approaches and a pure LS run. Note that memory was constrained for hybrid runs, marked with *; with memory constraints, INC would not run successfully in this case.

Overall, this technique prevents the transformation engine from trashing due to memory starvation, but is theoretically sub-optimal since the largest match set caches may not be the best choice for abandonment when optimizing for the shortest possible execution time. A straightforward approach for future optimization is adjusting the priority order based on static analysis of the transformation program.

5 Conclusion and future work

Practical experience has shown that optimizing model transformations is an important part of building powerful model transformations in a model-driven development process. First, as models are increasing in size and complexity, transformations need to be able to transform them efficiently. Secondly, as transformations are becoming *hidden* (e.g. embedded in a design tool), they should execute seamlessly - quickly and using as little resources as possible.

In this paper, we presented a *hybrid pattern matching* approach, which provides smart selection from two entirely different matching strategies (namely, the local search-based and incremental pattern matching) to improve overall performance.

Based on experience with complex applications of model transformations (e.g. [23]), we selected three scenarios for the investigation. Based on our experimental analysis, we argue that many practical transformations may significantly benefit from a hybrid pattern matching approach with properly selected matching strategies for the patterns. We gave conceptual guidelines on manual optimization based upon various metrics in Sec. 4. Additionally, as an initial contribution towards automatic optimization, we presented an adaptive approach switches pattern matching strategies when memory is running low.

However, we also recognize that the ultimate goal for optimizing model transformation performance is to enable the user to concentrate only on functionality and the software tool should select the optimal pattern matching strategy. In order to provide semi-automatic aids to the transformation designer for code optimization, and to develop a more optimal method for adaptive strategy switching, several well-known approaches can be adapted in the future.

- *Pattern analysis* may be used to classify graph patterns according to complexity, size, and complex cost metrics statically. While such techniques are currently used internally in our LS implementation, direct user interface feedback is needed to expose relevant data to the transformation designer.
- *Program analysis* aims to identify patterns and model manipulation steps that are frequently used, rarely used, or unused for a period of time by analyzing the transformation program, without actually running it.
- *Trace analysis* improves this knowledge of transformation behaviour by actually running the program on one or more provided typical models and gathering statistics on the type and amount of executed pattern queries and model manipulations.
- *Quantitative model analysis* is a highly promising approach to estimate the match set cardinality of graph patterns based on statistics of the model (without actually running the pattern matching algorithm).

As a main direction for future work, we plan to implement a framework with high-level support for these static analysis techniques. Additionally, we plan to investigate ways to achieve tighter integration between the two pattern matching engines. This will allow different strategies to be responsible for matching different subpatterns within the same pattern.

References

1. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
2. The AGTIVE Tool Contest: official website (2007) http://www.informatik.uni-marburg.de/˜swt/agtive-contest.
3. GraBaTs - Graph-Based Tools: The Contest: official website (2008) http://www.fots.ua.ac.be/events/grabats2008/.
4. Geiss, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Graph Transformations - ICGT 2006. Lecture Notes in Computer Science, Springer (2006) 383 – 397 Natal, Brasil.
5. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, ACM Press (2000)
6. Varró, G., Horváth, Á., Varró, D.: Recursive Graph Pattern Matching With Magic Sets and Global Search Plans. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE ’07). Volume 5088 of LNCS., Springer (2008)

7. Varró, G., Varró, D., Schürr, A.: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In Karsai, G., Taentzer, G., eds.: Graph and Model Transformation (GraMoT 2006). Volume 4 of Electronic Communications of the EASST., EASST (2006)
8. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT'08, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)
9. Matzner, A., Minas, M., Schulte, A.: Efficient Graph Matching with Application to Cognitive Automation. In Schürr, A., Nagl, M., Zündorf, A., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007), Springer Verlag (2007)
10. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 321–335
11. Mészáros, T., Madari, I., Mezei, G.: VMTS AntWorld submission. GraBaTs - 4th International Workshop on Graph-Based Tools: The Contest (September 2008)
12. Bergmann, G., Horvath, A., Ráth, I., Varró, D.: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: ICGT2008, The 4th International Conference on Graph Transformation. (2008)
13. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
14. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag (2003)
15. Varró, G., Schürr, A., Varró, D.: Benchmarking for Graph Transformation. Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics (March 2005) `http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf`.
16. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: In [1]. World Scientific (1999) 487–550
17. ATLAS Group: The ATLAS Transformation Language. Available from `http://www.eclipse.org/gmt`.
18. Zündorf, A.: Graph Pattern Matching in PROGRES. In: Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science, London, UK, Springer-Verlag (1996) 454–468
19. Varró, G., Varró, D., Friedl, K.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Karsai, G., Taentzer, G., eds.: Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of ENTCS., Tallinn, Estonia, Elsevier (September 2005) 191–205
20. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: 6th International Workshop on Theory an Application of Graph Transformations (TAGT, Springer (1998) 238–251
21. El-Boussaidi, G., Mili, H.: Detecting patterns of poor design solutions using constraint propagation. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008. Proceedings. Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 189–203
22. Wright, I., Marshall, J.: The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. International Journal of Intelligent Games and Simulation **2**(1) (February 2003) 36–48
23. Kovács, M., Lollini, P., Majzik, I., Bondavalli, A.: An Integrated Framework for the Dependability Evaluation of Distributed Mobile Applications. In: Proc. Int. Workshop on Software Engineering for Resilient Systems (SERENE 2008), Newcastle upon Tyne, UK, November 17-19. (2008) 29–38