A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation*

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

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar Tudósok krt. 2
bergmann.gabor@gmail.com,
{rath,ahorvath,varro}@mit.bme.hu

Abstract. In graph transformation, the most cost-intensive phase of a transformation execution is pattern matching, where those subgraphs of a model graph are identified and matched which satisfy constraints prescribed by graph patterns. Incremental pattern matching aims to improve the efficiency of this critical step by storing the set of matches of a graph transformation rule and incrementally maintaining it as the model changes, thus eliminating the need of recalculating existing matches of a pattern. In this paper, we propose benchmark examples where incremental pattern matching is expected to have advantageous effect in the application domain of model simulation and model synchronization. Moreover, we compare the incremental graph pattern matching approach of VIATRA2 with advanced non-incremental local-search based graph pattern matching approaches (as available in VIATRA2 and GrGen).

Keywords: incremental graph pattern matching, RETE, benchmarking.

1 Introduction

Incremental graph pattern matching approaches [18,1,2,3] have recently become a hot topic in the graph transformation community. The core guideline is to improve the execution time of the time-consuming pattern matching phase by additional memory consumption. Essentially, the (partial) matches of the left-hand side (LHS) of graph transformation rules 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.

Up to now, the performance evaluation of such incremental graph pattern matching approaches have been limited to dedicated benchmark examples, all of which were characterized by traditional, batch-like execution strategy.

In the current paper, we first propose two benchmark examples where an incremental pattern matching strategy appears to be very beneficial: (i) in the simulation example, enabled transitions of a Petri net are maintained in an incremental way, while (ii) in

* This work was partially supported by the SENSORIA European project (IST-3-016004). The fourth author was also supported by the János Bolyai Scholarship.

the model synchronization example, an object-relational mapping is carried out where changes in the source UML model are propagated incrementally to the corresponding relational database model.

In addition, we evaluate the performance of the incremental graph transformation engine of the VIATRA2[1] framework on various benchmark examples. A full-fledged comparison is provided with respect to the non-incremental version [4] of the VIATRA2 engine, furthermore, an initial comparison is provided with GrGEN.NET [5], which is currently considered to be the fastest graph transformation engine.

The rest of the paper is structured as follows. Section 2 briefly introduces model simulation captured by graph transformation rules, which serves as one of the benchmarks presented in the paper. In Sect. 3, an incremental graph pattern matching approach is overviewed, which was implemented in the VIATRA2 framework. As the main contribution, novel benchmark examples are presented in Sect. 4 for model simulation and model synchronization with performance evaluation discussed in Sect. 5. Finally, Sect. 6 summarizes the related work and Sect. 7 concludes the paper.

2 Foundations of Model Simulation

This section overviews the foundations of modeling language specification and simulation. In order to specify the abstract syntax of most modeling language, the concept of metamodeling is used. For simulating the behaviour of models, the paradigm of graph transformation [6] is applied.

2.1 Running Example: Simulation of Petri Nets

In the current paper, we will use the simulation of Petri nets as one of our performance benchmarks for incremental pattern matching. We will use the same example to demonstrate the technicalities of incremental pattern matching in graph transformation tools.

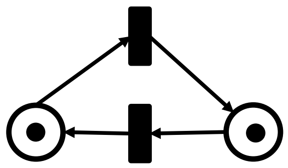


Fig. 1. A sample Petri net

Petri nets (Fig. 1) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. Petri nets are bipartite graphs, with two disjoint sets of nodes: *Places* and *Transitions*. Places may contain an arbitrary number of Tokens. A token distribution (marking) defines the state of the modelled system. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token and no place connected with an inhibitor arc contains a token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by Input Arcs) and add a token to all output places (as defined by Output Arcs).

2.2 Foundations of Metamodeling

A *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have attributes that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra *attributes*. *Associations* define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. The most typical multiplicity constraints are i) the at-most-one (0..1), and (ii) the arbitrary (denoted by *). A simple Petri net metamodel is shown in Fig. 2.

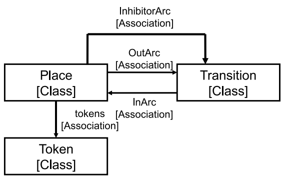


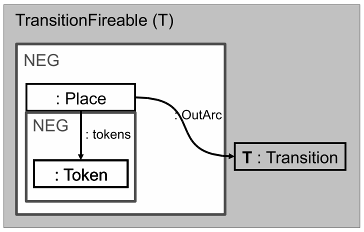
Fig. 2. Petri net metamodel

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required.

2.3 Graph Patterns and Graph Transformation

Graph patterns are frequently considered as the atomic units of model transformations [7]. They represent conditions (or constraints) that have to be fulfilled by a part of the instance model in order to execute some manipulation steps on the model. A basic graph pattern consists of graph elements corresponding to the metamodel. 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. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [8].

As an example, the firing enabledness condition for a Petri net transition may be expressed using a graph pattern as shown in Fig. 3 using the VIATRA2 notation. This pattern uses nested negative application conditions to express that a Transition is



```
pattern isTransitionFireable(Transition) ={
  transition(Transition);
  neg pattern notFireable_fl(Transition) =
  {
   place(Place);
   outArc(OutArc, Place, Transition);
   neg pattern placeToken(Place) =
   {
     token(Token);
     tokens(X, Place, Token);
   }
  }
}
```

Fig. 3. Petri-net firing condition

enabled if every input Place instance connected to the Transition instance has at least one Token instance associated and no inhibitor input Place instance contains tokens. In this example, embedded NACs are used to express universal quantification with double negation of existence.

Graph transformation (GT) [9] 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. For instance, a GT rule may specify how to remove (or add) a token from a place, as shown in Fig. 4.

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Place, in Transition) = {
 precondition find sourcePlaceWithToken
    (Transition, Place, Token);
 postcondition find sourcePlaceWithoutToken
    (Transition, Place, Token);
}
// Adds a token from the place 'Place'.
gtrule addToken(in Place, in Transition) = {
 precondition find targetPlaceWithoutToken
    (Transition, Place, Token);
 postcondition find targetPlaceWithToken
    (Transition, Place, Token);
}
```

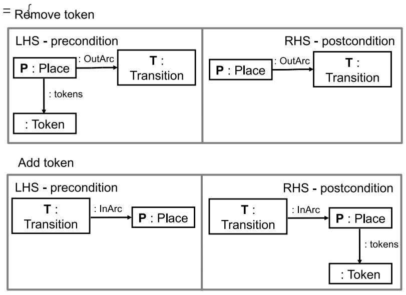


Fig. 4. Graph transformation rules for firing a transition

Complex model transformation can be assembled from elementary graph patterns and graph transformation rules using some kind of control language. In our examples, we use an abstract state machine (ASM) [10] for this purpose as available in the VIATRA2 framework. The following transformation (which will be used as a benchmark example in Sect. 4.1) simulates the firing of a transition, i.e. the removal of tokens from input places and the addition of tokens to output places (see Fig. 5).

```
rule fireTransition(in T) = seq {
 /* perform a check to confirm that the transition is fireable */
 if (find isTransitionFireable(T))
 seq
 {/* remove tokens from all input places */
  forall Place with find inputPlace(T, Place)
   do apply removeToken(T, Place); // GT rule invocation
  /* add tokens to all output places */
  forall Place with find outputPlace(T, Place)
   do apply addToken(T, Place);
 }
}
```

Fig. 5. Transformation program for firing a transition

3 RETE-Based Incremental Graph Pattern Matching

The incremental graph pattern matcher of the VIATRA2 framework [1] adapts the RETE algorithm, which is a well-known technique in the field of rule-based systems.

RETE network for graph pattern matching. RETE-based pattern matching relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates those model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a *view*. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur.

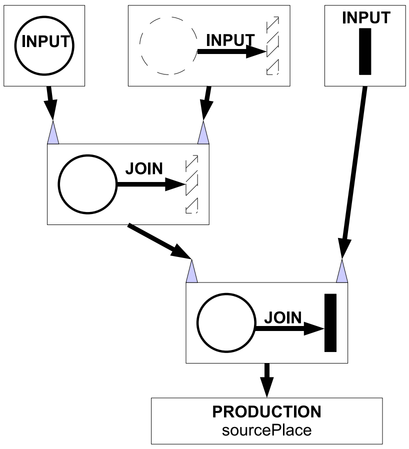


Fig. 6. Simple RETE matcher

Input nodes serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing a view representing all the instances that conform to the type. Similarly, there is an input node for each relation type, containing a view consisting of tuples with source and target in addition to the identifier of the edge instance.

At each *intermediate node*, *set operations* (e.g. filtering, projection, join, etc.) can be executed on the match sets stored at input nodes to compute the match set which is stored at the intermediate node. The match set for the entire pattern can be retrieved from the output *production node*. An intermediate node of a RETE is the join node, which performs a natural join on its input nodes in terms of relational algebra. A *negative node* contains the set of tuples stored at the primary input which do *not* match any tuple from the secondary input (which corresponds to anti-joins in relational databases).

Updates after model changes. Input nodes receive notifications about each elementary model change (i.e. when a new model element is created or deleted) and release an update token on each of their outgoing edges. Such an update token represents changes in the partial matches stored by the RETE node. Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set.

Upon receiving an update token, a RETE node determines how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes. An example RETE network is depicted in Fig. 6.

The match set can be retrieved from the network instantly without re-computation, which makes pattern matching very efficient. As a trade-off, there is increased memory consumption, and update operations become more complex.

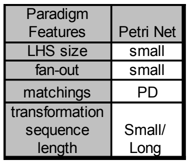
4 Benchmarks for Incremental Graph Transformation

In the paper, we propose two new benchmark problems as an extension to the Varro benchmarks [11]. From a problem-specific viewpoint, they address two important application scenarios, namely, model simulation and model synchronization, which were only partially covered in [11]. Moreover, from a tool-oriented viewpoint, they provide the first test sets for the "as-long-as-possible" optimization strategy, which was not measured up to now. Finally, the change propagation scenario in model synchronization is a highly realistic challenge for model transformation tools.

4.1 Simulation Scenario Based on Petri Net Firing

Description. We selected the Petri net benchmark for the scenario of simulation of visual languages with dynamic operational semantics. This scenario summarizes typical domain specific language simulation with the following characteristics: (i) mostly static graph structure, (ii) relatively small and local model manipulations, and (iii) typical *as-long-as-possible* (ALAP) execution mode. This benchmark focuses on the effective reusability of already matched elements as typical firing of a transition only involves a small part of the net. While an incremental pattern matcher can track the changes of the Petri net and updates only the involved sub-matchings, non-incremental local search based approaches will have to restart the matching from scratch after the net changed.

Test case generation. In the Petri net test set, we selected "regular" Petri nets as *test cases*, which are generated automatically. Here regular means that the number of *places* and *transitions* are approximately equal (where their exact ratio is around 1.1). Furthermore, the net has only a low number of tokens, and thus, there are few fireable transitions in each marking.



Paradigm Features	Petri Net
LHS size	small
fan-out	small
matchings	PD
transformation sequence length	Small/ Long

Fig. 7. Feature matrix of Petri Net benchmark

To generate the elements of the test set we used six reduction operations (in the inverse direction to increase the size of the net) which are described in [12] as means to preserve safety and liveness properties of the net. These operations are combined with a weighted random operation selection. This allows fine parametrization of the number of transitions and places with an average fan-out of 3-5 incoming and outgoing edges. In all test cases, the generation started from the Petri net depicted in Fig. 1 (which is trivially a live net) and the final test graphs are available in PNML [13] format at [14]. As the size of a Petri net cannot be described by only a single parameter we used the number of property preserving we applied to indicate the relative "size" of test cases.

Execution phases. A step in the iterative execution sequence contains two phases: (i) a fireable transition is non-deterministically selected by pattern *isTransitionFireable* (Fig. 3) and then (ii) the GT rules *addToken* and *removeToken* are applied to simulate the token flow (Fig. 4).

Despite its simple execution semantics, it is easy to derive additional Petri nets as new benchmark scenarios with significantly different run-time characteristics for the different graph transformation tools. For example, a Petri net with an equal number of transitions, places and tokens but with few fireable transitions can be used as a benchmark where type-based optimization strategies of pattern matcher algorithms are neutralized, which forces the pattern matchers to use other heuristics.

Note that the only assumption we made on our Petri net test cases is to use *live* and *bounded* nets to have a potentially unbounded execution sequence. We selected 1000 consecutive transition firings as *Short* execution sequences and 1000000 transition firings as *Long* execution sequences.

For this benchmark, we compared the total execution time of the simulation sequences. As the actual firing transitions are non-deterministically selected by the tools, we allowed the pattern matchers to select their own execution paths, but this turned out to have only insignificant effects on execution times.

Characteristics. In order to give a comparable description of our proposed benchmarks with the ones defined in [11] we also use *feature matrices* to describe the characteristics of the new test sets. The definition of the features are the following:

- *Pattern size*, or the number of nodes and edges in the LHS graph, is a critical factor in the runtime phase of pattern matching.
- The maximum degree of nodes (*fan-out*) in the model is the number of edges that are adjacent to a certain node.
- The third feature is the *number of matches* during the test case execution.
- The *length of the transformation sequence* also affects the overall execution time. For example, with a large number of rule applications, the relative cost of one-time overhead of the pattern matcher is decreased.

Fig. 7 presents the feature matrix describing the Petri net test case. Note that if the characteristics of a feature depends on the concrete parameter settings of the test case, then it is called parameter dependent (marked PD).

4.2 Model Synchronization Scenario by Object-Relational Mapping

Description. The Object-to-Relational schema mapping (ORM) benchmark, as presented in the current paper, is an extension of the original benchmark proposed in Sect.4 of [11]. The original transformation processed UML class diagrams to produce corresponding relational database schemas, according to the known mapping rules. Since a straightforward application of the incremental pattern matching approach is the synchronization between source and target models, we extended the benchmark by two additional sequences: (i) after the initial mappings are created, 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). A local seach-based algorithm has to search for the changes first, while an incremental pattern matcher can track changes in the source model so that the model parts affected are instantly available for the synchronization sequence.

Test case generation. In order to produce sufficiently large model graphs for the measurements, we implemented a simple generator as described in [11]. By this approach, a fully connected graph is created, i.e. for N UML classes, $N(N-1)$ directed associations are defined (with each association represented as three nodes – an association node and two endpoints). Additionally, each UML class can reference K attributes, thus, for a given N and K, $N+3N(N-1)+NK$ nodes and $4N(N-1)+NK$ edges are created (Fig. 8). Although the model produced is not "realistic" in the sense that very few practical UML class diagrams are fully connected, the method is quite efficient in creating large graphs quickly.

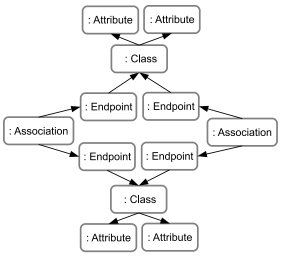


Fig. 8. Generated UML class diagram for N=K=2

Execution phases. The transformation sequence is comprised of four main phases:

1. The *generation* phase creates the model graph.
2. The *build* phase creates the initial mapping of the UML model into the relational schema domain, with reference models connecting mapped model objects.
3. The *modification* phase modifies the UML models programmatically to emulate user editing actions.
4. Finally, the *synchronization* phase locates the affected model elements and makes changes in the schema model accordingly.

Characteristics. For this benchmark, we compare the execution times for the last (synchronization) phase. In order to scale the synchronization sequence as the model size grows, we designed the modification sequence to extend roughly linearly with the model. Thus, in the default case, it is composed of the following operations: (i) first, one third of generated classes, along with their attributes and referenced associations are deleted; (ii) then, one fifth of remaining associations are deleted; (iii) next, every second attribute is renamed; (iv) finally, a new class is added and a new fully connected graph is created (with the remaining UML classes and the newly added class as nodes, ignoring existing associations). The feature matrix based on the notation in Sect. 4.1 is shown in Fig. 9.

Transformation rules for synchronization. In incremental synchronization, to avoid rebuilding target models in each pass, a *reference model* is used to establish a mapping relationship between source and corresponding target model elements (Fig. 10). With correspondence edges, it is possible to track changes in both the source and target models: for instance, the graph pattern on Fig. 11 matches *tables* in the schema model which are no longer referenced by classes or associations in the UML models (*orphan tables*).

Paradigm Features	ORM
LHS size	large
fan-out	medium
matchings	PD
transformation sequence length	PD

Fig. 9. Feature matrix for the ORM benchmark

Similarly, a newly created class may be matched by a negative condition forbidding the existence of a mapped table. Renames (value changes) may be expressed e.g. by

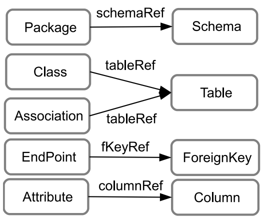


Fig. 10. Reference metamodel

```
pattern orphanTable(T) =
{
  table(T);
  neg pattern mapped(T) =
  {
    class(C);
    table(T);
    class.tableRef(_REFN, C, T);
  } or {
    association(A);
    table(T);
    association.tableRef(_REFN, A, T);
  }
}
```

Fig. 11. Graph pattern for orphan tables

matching for both the attribute and the mapped column, and looking for pairs where the name (attribute value) is different. The modification sequence results in the following synchronization sequence: (i) all orphan tables belonging to the deleted classes and their associations are deleted; (ii) all orphan tables belonging to the deleted associations are deleted; (iii) column names mapped to the renamed attributes are changed; (iv) new tables are added for the newly created class and the new associations.

5 Measurement Results

The measurements reported in this paper have been carried out on a standard desktop computer with a 2 GHz Intel Core2 processor with 2 gigabytes of system RAM available, running version 1.6.0_05 of the 32-bit Sun Java SE Runtime (for VIATRA2) and version 3.0 of the .NET Framework on Windows Vista (for GrGEN.NET). In general, ten test runs were executed, and the results were calculated by averaging the values excluding the highest and lowest number. The transformation sequences were coded so that little or no output was generated; in the case of VIATRA2, we refrained from disabling the GUI. Execution times were measured with millisecond precision as allowed by the operating system calls.

5.1 Distributed Mutual Exclusion Algorithm

In order to compare the pattern matcher algorithms used in this paper with an already available benchmark, we evaluated the performance of the VIATRA2 local search based (VIATRA/LS) and incremental (VIATRA/RETE) pattern matchers along with the GrGEN.NET with the *distributed mutual exclusion algorithm* test set defined in [11] which is not a primary application filed for incremental pattern matching.

The results are shown in Fig. 12 with logarithmically scaled axes, where the *size of the process ring* represents the number of processes in the run, which is, in turn, the runtime parameter for the test case. We can make the following observations: (i) the scaling complexity is a high order polynomial for VIATRA/LS and close to linear for VIATRA/RETE and linear for GrGEN.NET; (ii) this test set seems to fit better for optimized local search based approaches as incremental caching of non-reuseable model

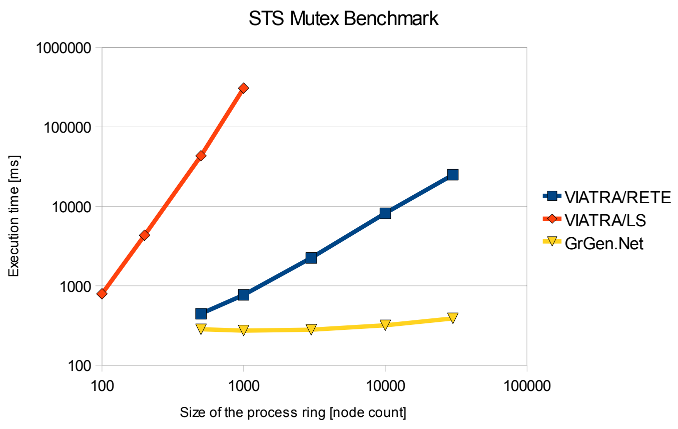


Fig. 12. Results for the Short Transformation Sequence mutex benchmark

elements produced in the second phase increases the overhead of the cache synchronization. Additionally, by looking at memory consumption figures, it can be seen that the static graph structure limits the memory overhead to the same order of magnitude for VIATRA/RETE and GrGEN.NET.

5.2 Simulation of Petri Nets

The Petri net synchronization benchmark was executed with *short* (1000) and *long* (1000000) execution sequences.

The size parameters of the nets used as test cases are depicted in Fig. 13. *Net size* represents the number of randomly applied inverse property preserving operations used during their generation, while *Places*, *Transitions* and *Tokens* represent their actual number. The results are shown in Fig. 14 with logarithmically scaled axes, where model size indicates the *net size* of the test case.

Net Size	Places	Transitions	Tokens
10000	7497	7450	10
20000	14987	14870	10
50000	37581	37593	10
75000	56331	56053	10
100000	74924	75124	10

Fig. 13. Size of test cases

As it can be seen from the graph, VIATRA/RETE has a predictable linear scaling up to model size of 10^5 with a speed of at least two orders of magnitude faster than VIATRA/LS. As expected, the incremental approach works well for large model sizes as long as there is enough memory (the spike in case of long transformation sequences occured because of garbage collection as the heap was exceeded).

VIATRA/RETE matches and outperforms the GrGEN.NET tool for very large models in case of both short and long execution sequences. Moreover, with additional memory provided, the characteristics of VIATRA2 are expected to be better for even larger models with predictable execution time.

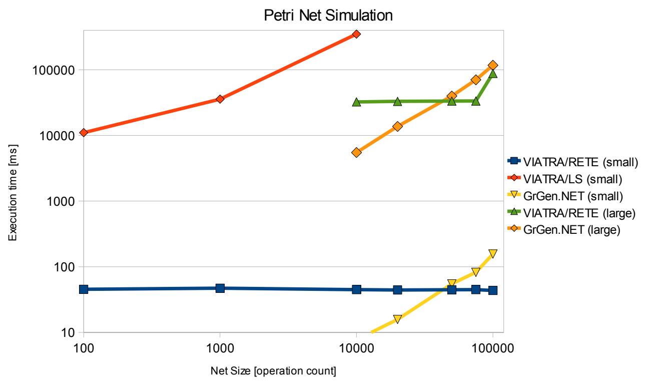


Fig. 14. Results for the Petri net firing benchmark

This result is a significant achievement considering the architectural and run-time differences between VIATRA2 and GrGEN.NET. Most notably, GrGEN.NET uses compile-time optimizations and an entirely different model persistence approach based on compile-time generated type information, whereas VIATRA2 uses a generic model storage supporting dynamic typing and support for interactive applications such as a notification and transaction management mechanism (note that the VIATRA2 GUI was not disabled for the measurement, while GrGEN.NET was used without GUI through GrShell). However, for fairness, it should be pointed out that (unlike the mutual exclusion case) this benchmark was prepared by ourselves (i.e. by GrGEN non-experts), thus additional language or tool-specific optimizations might be available.

5.3 Object-Relational Mapping Synchronization

The ORM synchronization benchmark was executed with the VIATRA2 tool (due to time constraints, measurements with GrGEN.NET and others are left as future work). Models up to 67800 nodes (with edges, the total model size is 157800 model elements) were generated (Fig. 15) and the execution time for the *build* and *synchronization* phases was measured.

Total node count	Generated classes	Generated associations	Deleted orphan tables	Renamed attributes	Newly created nodes
85	5	20	16	6	39
705	15	210	159	20	333
1925	25	600	453	32	819
7600	50	2450	1765	66	3369
17025	75	5550	4015	100	7653
30200	100	9900	7178	132	13269
67800	150	22350	16080	200	30303

Fig. 15. Model and synchronization sequence sizes for the ORM benchmark

The results are shown in Fig. 16 (model size is the total number of nodes). It is again revealed that the scaling characteristic of both phases is exponential for VIATRA/LS and linear for VIATRA/RETE. With respect to synchronization, the constant

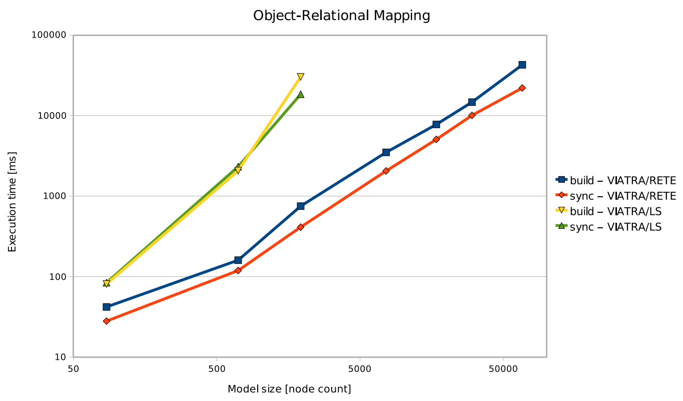


Fig. 16. Results for the ORM synchronization benchmark

difference between the *build* and *sync* phases for VIATRA/RETE means a constant multiplier; thus, since the model elements affected by the modification sequence are a linear fraction of the whole model, it can be concluded that the execution time for the synchronization process is a linear function of the model elements affected (as expected), and independent of the size of the rest of the model. VIATRA/LS, on the other hand, exhibits an ever increasing time difference between *build* and *sync*, thus, the time taken for the synchronization process increases exponentially with the number of affected model elements (again, as expected, since in case of local search, the system has to locate the changed elements first which is an additional graph traversal). It is important to note that for "practical" model sizes (e.g. below the 5000 node count range), VIATRA/RETE can perform a synchronization affecting a considerable portion of the model in the 10-500 msec range which makes the approach very suitable for interactive applications.

In addition to execution times, the memory consumed by the Java Virtual Machine was also recorded. The sequence for the RETE matcher (75, 100, 114, 245, 490, 750, 1000 megabytes respectively for model sizes from 85 to 67800 nodes) shows a linearly expanding RETE network as the node count grows, which is in-line with our expectations based on the nature of the RETE building algorithm (note that the above figures include the whole user interface with a complete Eclipse instance).

5.4 Summary

Analyzing the results obtained in our test cases, the following conclusions can be drawn:

(i) A major concern of any incremental pattern matching implementation is the increased memory consumption. While our implementation does indeed consume more memory than the standard local search-based VIATRA engine, this overhead, even for the extreme model sizes in the benchmark problems, is still within the bounds of RAM available in modern desktop computers making the approach feasible for a wide range of applications.

(ii) Within the memory boundaries, our new RETE-based pattern matcher provides a *predictable, linear scaling* up to the 10^5 model size range in all three scenarios. While even generic transformations experience a speed-up, the real potential of the implementation is revealed in the scenarios especially suited for incremental pattern matching where the execution speed matches, or even surpasses the speed of the fastest conventional graph transformation tool employing compile-time optimization.

(iii) By comparing the run-time characteristics of our multiple test cases, it seems evident that the best results could be achieved by employing different pattern matching strategies for different execution phases, or, even for different patterns in a model transformation program.

6 Related Work

Incremental pattern matching. Incremental updating techniques have been widely used in different fields of computer science. Now we give a brief overview on incremental techniques that are used in the context of graph transformation. The transformation engine of TefKat [16] performs an SLD resolution based interpretation during which a search space tree is constructed to represent the trace of transformation execution. This tree is maintained incrementally in consecutive steps of transformations as described in [17]. The uniform, incremental handling of model elements and patterns can be considered a unique, advanced feature of the approach. [18] proposes a graph pattern matching technique, which constructs and stores a tree for partial matchings of a pattern, and incrementally updates it, when the model changes. The main advantage of this solution is that only matchings, which appear as leaves of the tree, have to be physically stored, which possibly saves a significant amount of memory. The memory saving technique of [18] is orthogonal to the structure of the underlying RETE network, and, thus, it can expectedly be used for our approach as well, but the exact integration requires further research and implementation tasks.

RETE networks. RETE networks [19], which stem from rule-based expert systems, have already been used as an incremental graph pattern matching technique in several application scenarios including the recognition of structures in images [20], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [21]. Our contribution extends this approach by supporting a more expressive and complex pattern language.

Graph transformation benchmarking. Some of the measurements in the current paper are conceptual continuations of the comprehensive graph transformation benchmark proposed in [11], which gave an overview on typical application scenarios of graph transformation together with their characteristic features. [15] suggested some improvements to the benchmarks described in [11] and reported measurement results for many graph transformation tools including AGG [22], PROGRES [23], Fujaba [24], and GrGEN.NET [5]. A similar approach to graph transformation benchmarking was used for the AGTIVE Tool Contest [25], including a simulation problem for the Ludo table game. Our Petri net firing test case is better suited for benchmarking performance since it can be parameterized to scale up to large model sizes and long transformation sequences.

7 Conclusion and Future Work

In the current paper, we have have proposed two new test cases as performance benchmarks of graph transformation which are suitable for assessing incremental graph transformation strategies. For this purpose, we focused on two scenarios: (i) The Petri net *model simulation* benchmark was designed to provide a parameterizable and scalable test case for analyzing the impact of incremental pattern matching on a typical simulation scenario; (ii) the Object-Relational Mapping scenario was adapted to *model synchronization* which is a prime target for an event-driven application of graph transformation, where models have to be mapped on-the-fly as the user is editing the model.

We carried out various measurements to assess the performance of the incremental pattern matcher of the VIATRA2framework [1], which clearly demonstrate the viability of the approach: very fast execution with predictable, linear scaling up to memory limitations.

By analyzing the test runs with a Java code profiler, we have identified some key areas where the performance of the VIATRA/RETE tool could be further improved in the future, such as (i) optimizing VIATRA model persistence, especially with regard to type information storage and attribute handling; (ii) employing more efficient search plan generation for the construction of the RETE network; (iii) reducing code interpretation overhead by precompiling model manipulation sequences into native Java calls.

In additon to improving performance, we plan to provide support for mixing different pattern matching strategies to allow the transformation designer to specify which pattern matcher implementation should be used on a per-pattern basis. Additionally, we plan to investigate the possibilities of adaptive pattern matching strategy change based on automatic profiling.

References

1. Bergmann, G., et al.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT 2008, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (accepted, 2008)
2. Matzner, A., Minas, M., Schulte, A.: Efficient graph matching with application to cognitive automation. In: Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Springer, Heidelberg (2007)
3. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
4. Varró, G., Horváth, Á., Varró, D.: Recursive graph pattern matching with magic sets and global search plans. In: Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Springer, Heidelberg (2007)
5. Geiss, R., et al.: GrGEN: A fast spo-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
6. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations: Foundations. World Scientific, Singapore (1997)
7. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. 68(3), 214–234 (2007)

8. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)
9. Ehrig, H., et al.: Handbook on Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
10. Börger, E., Särk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
11. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, Texas, USA, pp. 79–88. IEEE Press, Los Alamitos (2005)
12. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE, April 1989, published as Proceedings of the IEEE, vol. 77(4), pp. 541–580 (1989)
13. Jungel, M., Kindler, E., Weber, M.: The Petri Net Markup Language. In: Algorithmen und Werkzeuge fur Petrinetze (AWPN), Koblenz (June 2002)
14. The VIATRA2 Framework: official website (2008), `http://viatra.inf.mit.bme.hu`
15. Geiss, R., Kroll, M.: On improvements of the Varro benchmark for graph transformation tools. Technical Report 2007-7, Universität Karlsruhe, IPD Goos 12 (2007)
16. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Proc. International Workshop on Model Transformation in Practice (MTiP 2005) (October 2005)
17. Hearnden, D., et al.: Incremental model transformation for the evolution of model-driven systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
18. Varró, G., Varró, D., Schürr, A.: Incremental graph pattern matching: Data structures and initial experiments. In: Proc. Graph and Model Transformation (GraMoT 2006). Electronic Communications of the EASST, vol. 4 (2006)
19. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (1982)
20. Bunke, H., Glauser, T., Tran, T.H.: An efficient implementation of graph grammar based on the RETE-matching algorithm. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 174–189. Springer, Heidelberg (1991)
21. Matzner, A., Minas, M., Schulte, A.: Efficient graph matching with application to cognitive automation. In: Proc. 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance, Kassel, Germany, October 2007, pp. 293–308 (2007)
22. Ermel, C., Rudolf, M., Taentzer, G.: The AGG-Approach: Language and Tool Environment. In: [9], pp. 551–603. World Scientific, Singapore (1999)
23. Schürr, A.: Introduction to PROGRES, an attributed graph grammar based specification language. In: Nagl, M. (ed.) WG 1989. LNCS, vol. 411, pp. 151–165. Springer, Heidelberg (1990)
24. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland. ACM Press, New York (2000)
25. The AGTIVE Tool Contest: official website (2007), `http://www.informatik.uni-marburg.de/~swt/agtive-contest`