Incremental Pattern Matching in the VIATRA Model Transformation System*

Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, Gergely Varró
Budapest University of Technology and Economics
Department of Measurement and Information Systems
1117 Budapest, Magyar Tudósok krt. 2.
bergmann.gabor@gmail.com, okrosa@gmail.com, {rath, varro}@mit.bme.hu,
gervarro@cs.bme.hu

ABSTRACT

Incremental pattern matching is a key challenge for many tool integration, model synchronization and (discrete-event) model simulation tasks. An incremental pattern matching engine explicitly stores existing matches, while these matches are maintained incrementally with respect to the changes of the underlying model. In the current paper, we present an adaptation of RETE networks [6] in order to provide incremental support for the transformation language of the VIATRA2 framework. We evaluate the performance of the incremental engine on a benchmark problem assessing the speed-up of incremental processing in the case of as-long-as-possible type of rule applications.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets; Object-oriented design methods*

General Terms

Algorithms, Languages, Performance

Keywords

domain-specific languages, incremental graph pattern matching, incremental model transformation

1. INTRODUCTION

Nowadays, in a typical development scenario of safety-critical (e.g. automotive and avionics) systems, tool integration has become a major cost factor due to the large number of development tools. The cost of integration between more than fifty tools from different vendors and roles is frequently comparable to the costs of the tools themselves. Model-driven tool intregration has a growing popularity since it provides a well-founded approach to bridge various modeling languages and tools. These bridges are frequently specified as model transformations between these languages and models. Graph transformation [5] provides a popular and frequently used means to precisely capture such model transformations with a wide range of available tools.

A key problem in model-based tool integration is the incremental synchronization of various models. In this scenario, if a developer changes one model, the effects of these changes should be propagated to other tools, preferably without reexecuting the entire model transformation from scratch. Incremental model synchronization has also been identified by the QVT standard [13] as a key model transformation problem for a successful model-driven engineering process. Unfortunately, existing model transformation tools only provide limited support for incrementality. In the typical case, the models themselves may evolve incrementally, but to achieve such incrementality, complex computations are required. In the case of graph transformation tools, negative application conditions may forbid the application of a rule to source model elements which are already transformed to its target equivalent. However, when a model changes, graph patterns need to be reevaluated from scratch, which includes the expensive evaluation of negative conditions as well. The situation is not very different in case of QVT-based tools, where traceability links are created during a transformation.

In the current paper, we argue that a better support of incremental model transformations are obtained by storing the matches of patterns, and then incrementally updating the existing matches when the underlying models change. As a result, matches of a pattern can be obtained very efficiently in constant time by sacrificing time for the update phase, and space for book-keeping of matches. This is exactly the case for providing efficient support for many model synchronization as well as (discrete-event) model simulation problems.

In order to support incremental graph pattern matching, we implemented and adapted the RETE-approach [6] to support the rich transformation language of the VIATRA2 model transformation system [16]. After a brief conceptual overview (Sec. 3), we demonstrate how a RETE network can be constructed for the graph patterns of the VIATRA2 language (Sec. 4). Then, in Sec. 5, we assess the performance of our RETE-based engine on a benchmark example where we expect that such an incremental solution should outperform traditional pattern matching solutions based on local searches. Since our solution provides full support for the rich language constructs of VIATRA2, we significantly supersede and extend the first (and relatively old) RETE-based graph transformation approach [4].

*This work was partially supported by the Sensoria European IP (IST-3-016004).

GraMoT'08, May 12, 2008, Leipzig, Germany.

2. DEMONSTRATING EXAMPLES

In the paper, we use Petri nets as a demonstrating example to illustrate the technicalities of our approach.

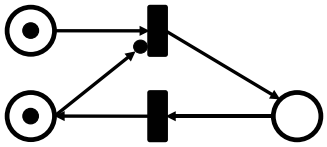


Figure 1: A sample Petri net.

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. From a system modelling point of view, a Petri net model is frequently used for correctness, dependability and performance analysis in early stages of design. 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 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).

2.1 Metamodeling foundations

In order to understand how the concepts of RETE are adapted to the VIATRA2 graph transformation environment, we give a brief overview of the the metamodeling foundations of this framework.

The VIATRA2 framework uses the VPM (Visual and Precise Metamodeling) [17] metamodeling approach, which can support different metamodeling paradigms by supporting multi-level metamodeling with explicit and generalized instance-of relations.

The VPM language consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). *Entities* represent basic concepts of a (modeling) domain, while *relations* represent the relationships between other model elements. Furthermore, entities may also have an associated value which is a string that contains application-specific data.

In traditional graph transformation terms, entities can be interpreted as nodes while relations are edges. Entities in a metamodel define node types while entities in models are simply referred to as nodes. In the paper, we use the VIATRA2 terminology for models to avoid the overloading of terms "node" and "edge", also used in the context of RETE networks.

A simple Petri net metamodel, represented in VIATRA2, is shown on Fig. 2.

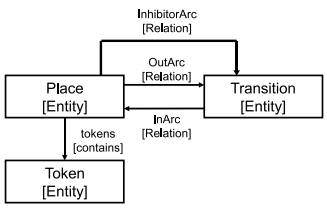


Figure 2: VIATRA Petri net metamodel.

2.2 Model transformations

The transformation language of VIATRA2 (Viatra Textual Command Language – VTCL) consists of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) [5] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [3] rules can be used for the description of control structures.

Graph patterns are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. The basic pattern body contains model element and relationship definitions.

In VTCL, *patterns may call other patterns* using the *find* keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled. A *negative application condition* (NAC, defined by a negative subpattern following the *neg* keyword) 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 [14].As an example, the firing enabledness condition for a Petri net transition may be expressed using a *graph pattern* as shown in Fig. 3. This pattern uses nested negative application conditions to express that a Transition is 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) [5] provides a high-level rule and pattern-based manipulation language for graph models. In VTCL, graph transformation rules may be specified by using a *precondition* (or left-hand side – LHS) pattern determining the applicability of the rule, and a *postcondition* pattern (or right-hand side – RHS) 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. Further actions can be initiated by calling any ASM instructions within the *action* part of a GT rule, e.g. to report debug information or to generate code.

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

Using these constructs, **complex model transformations** can be constructed. In Sec.5, we make use of a simulation sequence to benchmark the performance of the RETE-based pattern matcher against VIATRA2's built-in local search based implementation. This sequence executes transformation rules which simulate the firing of a transition, i.e. the removal of tokens from input places and the addition of tokens to output places. (see Listing 3).

3. INCREMENTAL PATTERN MATCHING

3.1 Core idea

In case of incremental pattern matching, the occurrences of a pattern are readily available at any time, and they are incrementally updated whenever changes are made. As pattern occurrences

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

Listing 1: VIATRA source code for the isTransitionFireable pattern

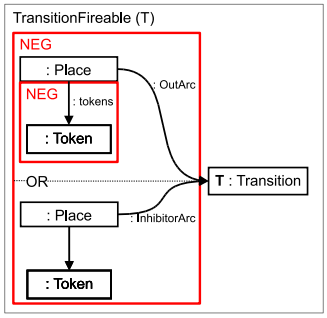


Figure 3: Petri-net firing condition

```
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);
 }
}
```

Listing 3: VIATRA source code for firing a transition

are stored, they can be retrieved in constant time[1], making pattern matching a very efficient process. Besides memory consumption, the drawback is that these stored result sets have to be continuously maintained, imposing an overhead on update operations.

In graph transformation frameworks, pattern matching is required to find the occurrences of left-hand side (LHS) patterns. Since pattern matching can be an important complexity factor in graph transformations, an incremental approach may lead to better performance, especially when transformations are matching-intensive instead of being manipulation-intensive. In this paper, we introduce an incremental pattern matcher component for the VIATRA2 framework; it is based on the RETE algorithm, which is a well-known technique in the field of rule-based systems.

3.2 Workflow

Initialising an incremental pattern matching engine involves the

[1]excluding the linear cost induced by the size of the result set itself

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Place, in Transition) = {
 precondition find sourcePlaceWithToken
    (Transition, Place, Token); // pattern call
 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);
}
```

Listing 2: VIATRA source code for graph transformation rules

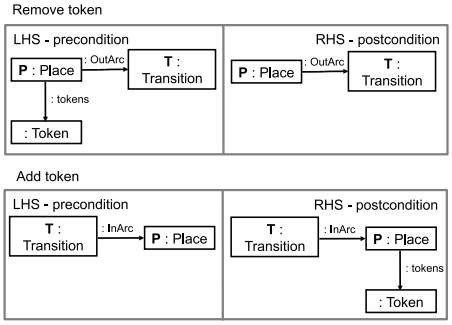


Figure 4: Graph Transformation rules for firing a transition

following conceptional steps:

1. The transformation designer defines various patterns and transformation rules.
2. An incremental pattern matcher (in our case, a RETE network) is constructed based on the pattern definitions.
3. The underlying model is loaded into the incremental pattern matcher as the the initial set of matches.

Typically Step 2 and 3 are carried out in RETE networks a single, interleaving process (as to be discussed in Sec. 4.8). Furthermore, the initialization need not be complete; the pattern matcher RETE network can be freely extended (on demand) with additional patterns at a later phase. It is worth pointing out that a RETE-based incremental pattern matcher can be integrated with any a graph transformation engine or any other underlying model manipulation library. For instance, a GT engine with a RETE-based incremental pattern matcher necessitates the the repeated execution of the following steps (see Fig. 5 for illustration):

1. Match LHS and other patterns in *constant time*;
2. Calculate the difference of the RHS and LHS (and potentially perform more actions);
3. Update the underlying model and notify the incremental pattern matcher of the changes;
4. Propagate the updates within the RETE network to refresh the set of matches.

3.3 Architecture

Since the VIATRA2 model transformation framework is designed in a way such that it is extensible with alternative pattern matcher modules, our prototype implementation of a RETE-based matcher is based on this as illustrated on Fig. 6. The incremental pattern matcher offers (implements) the standard pattern matcher interface, and while the RETE network is being constructed, it loads the contents of the initial model. The key architectural difference from the standard local search-based pattern matcher is that the incremental

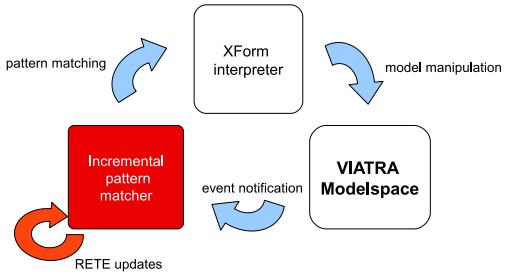


Figure 5: Incremental pattern matching information flow

pattern matcher subscribes for change notifications from the model management framework (called model space in VIATRA2 [17]); this allows RETE to update the results sets automatically whenever changes are made to the model.

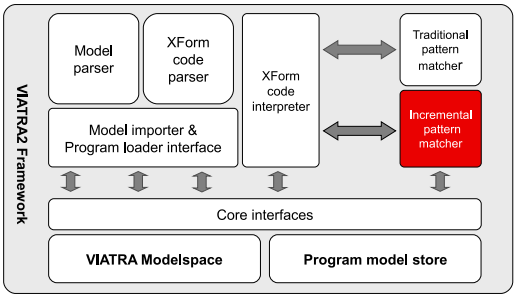


Figure 6: Incremental pattern matching in the VIATRA2 architecture

4. ADAPTING THE RETE ALGORITHM FOR VIATRA TRANSFORMATIONS

The RETE algorithm, introduced in [6], has a wide range of interpretations and implementations. This section describes how we adapted the concepts of RETE networks to implement the rich language features the VIATRA2 graph transformation framework.

In this section, we will gradually construct a RETE-based pattern matcher capable of matching the pattern isTransitionFireable, which is the LHS of the Petri net firing rule depicted in Fig. 3.

4.1 Tuples and Nodes

The main ideas behind the incremental pattern matcher are conceptually similar to relational algebra. Information is represented by a tuple consisting of model elements. Each node in the RETE net is associated with a (partial) pattern and stores the set of tuples that conform to the pattern. This set of tuples is in analogy with the relation concept of relational algebra.

The *input nodes* are a special class of nodes that serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing unary tuples representing the instances that conform to the type. Similarly, there is an input node for each relation type, containing ternary tuples with source and target in addition to the identifier of the edge instance. Miscellaneous input nodes represent containment, generic type information, and other relationship between model elements.

Intermediate nodes store partial matches of patterns, or in other terms, matches of partial patterns. Finally, *production nodes* represent the complete pattern itself. Production nodes also perform supplementary tasks such as filtering those elements of the tuples that do not correspond to symbolic parameters of the pattern (in analogy with the projection operation of relational algebra) in order to provide a more efficient storage of models.

4.2 Joining

The key component of a RETE is the join node, created as the child of two parent nodes, that each have an outgoing RETE edge leading to the join node.

The role of the join node can be best explained with the relational algebra analogy: it performs a natural join on the relations represented by its parent nodes.

```
pattern sourcePlace(T, P) = {
  transition(T);
  place(P);
  outArc(A, P, T);
}
```

Listing 4: VIATRA source code for the `sourcePlace` pattern

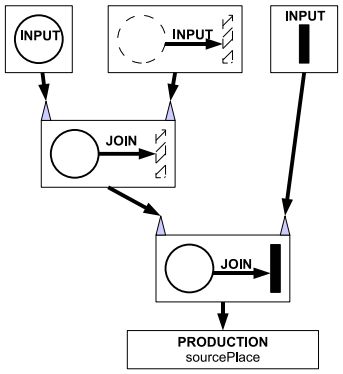


Figure 7: RETE matcher for the sourcePlace pattern

Figure 7 shows a simple pattern matcher built for the *sourcePlace* pattern illustrating the use of join nodes. By joining three input nodes (the top-most nodes on Fig. 7), this sample RETE net enforces two entity type constraints ('Place' and 'Transition' entity types on the left and right input nodes) and an edge (connectivity) constraint (corresponding to the relation connecting the 'Place' and 'Transition' entity types), to find pairs of Places and Transitions connected by an out-arc. A RETE node (depicted as white rectangles) represents a collection of partial pattern matches (the set of elements satisfying a constraint); the output node (on the bottom) stands for the matching set of the entire pattern for which the matcher network was built.

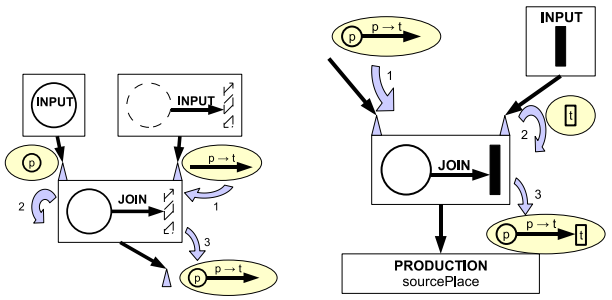
4.3 Updates after model changes

The primary goal of the RETE net is to provide incremental pattern matching. To achieve this, input nodes receive notifications about changes on the model, regardless whether the model was changed programmatically (i.e. by executing a transformation) or by user interface events – hence, our RETE-based concept can be adapted to any kind of model management and transformation framework as long as the model container supports elementary change notification.

Whenever a new entity or relation is created or deleted, the input node of the appropriate type will release an update token on each of its outgoing edges. Such an update token represents changes in the partial matchings stored by the RETE node. To reflect type hierarchy, input nodes also notify the input nodes corresponding to

the supertype(s). Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set.

Each RETE node is prepared to receive updates on incoming edges, assess the new situation, determine whether and 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.



(a) Phase I. (b) Phase II.

Figure 8: Update propagation

Figure 8(a) shows how the network in Fig. 7 reacts on a newly inserted out-arc. The input node for the relation type representing the arc releases an update token (depicted by a yellow ellipse). The join node receives this token, and uses an effective index structure to check whether matching tuples (in this case: places) from the other parent node exist. If they do then a new token is propagated on the outgoing edge for each of them, representing a new instance of the partial pattern "place with outgoing arc". Fig. 8(b) shows the update reaching the second update node, which matches the new tuple against those contained by the other parent (in this case: transitions). If matches are found, they are propagated further to the production node.

4.4 Pattern Call

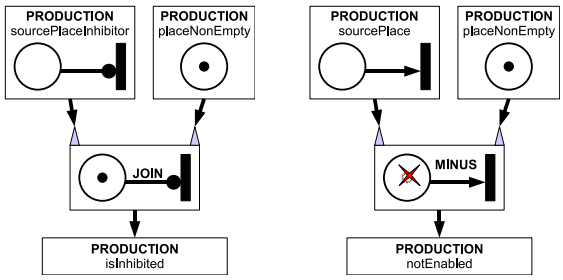
An important feature of the RETE algorithm is that network parts can be shared between patterns, thus reducing space and time complexity. The currently implemented pattern matcher is not yet capable of identifying common subpatterns for this optimization, but the transformation designer can help by decomposing patterns into smaller, reusable parts calling each other (also called pattern composition).

When a pattern calls another pattern, it can simply use the appropriate production node to obtain the set of tuples conforming to the other pattern. Naturally, the production node may have children attached like any other nodes. It is even possible to define recursive patterns that call themselves; in such cases, the production node of the pattern will have an edge leading back to one of the previous nodes. It is the designer's responsibility to ensure that the recursion is well-founded and that there is always exactly one fixpoint as result.

Figure 9(a) shows the matcher for pattern isInhibited provided that the simple patterns placeNonEmpty and sourcePlaceInhibitor already have their respective matchers constructed. The matcher selects tuples where the corresponding transition is inhibited by the place for whom the place inhibits the transition, and the place has at least one token.

```
pattern isInhibited(T) = {
  find sourcePlaceInhibitor(T,P);
  find placeNonEmpty(P);
}
pattern notEnabled(T) = {
  find sourcePlace(T,P);
  neg find placeNonEmpty(P);
}
```

Listing 5: VIATRA source code for the `isInhibited` and `notEnabled` patterns



(a) isInhibited (b) notEnabled

Figure 9: Positive and negative pattern calls

4.5 Negative Application Conditions

A powerful feature of VIATRA2 is to embed patterns into each other as negative application conditions, thus allowing negation at arbitrary depth. To support such negative pattern calls, the existing mechanism for pattern calls can be used, but the production node has to be connected to a negative node instead of a join node. A *negative node* (in the RETE network) has two distinct parents: primary and secondary inputs, respectively. The negative node contains the set of tuples that are also contained by the primary input, but do *not* match any tuple from the secondary input (which corresponds to antijoins in relational databases, see a similar idea with left outer joins e.g. in [18]).

Figure 9(b) shows the matcher for pattern notEnabled, provided that the simple patterns placeNonEmpty and sourcePlace already have their respective matchers constructed. The matcher selects the transitions with source places that do not have any tokens.

4.6 Disjunction

OR-Patterns (containing the 'or' keyword) are treated as a disjunction of independent pattern bodies. A separate matcher can be constructed for each body, sharing the production node, which will perform a true union operation on the sets of tuples conforming to each pattern body.

Figure 10 shows the matcher for pattern isTransitionFireable (see Listing 6), containing an inline negative pattern with two bodies. In this case, each body is a simple reference to a previously constructed pattern, connected to a single production node for the inline pattern.

4.7 Term Evaluation

In addition to simple graph-based structural constraints, the VIATRA2 framework supports the use of attribute conditions to restrict the names and values of model elements. Various arithmetical and logical functions, or even user-provided arbitrary Java code can be applied to model elements to check the validity of a pattern.

The term evaluator node propagates only those tuples that pass a given test. Furthermore, it registers the affected elements of incom-

```
pattern isTransitionFireable(T) = {
  transition(T);
  neg pattern notFireable(T) = {
    find notEnabled(T);
  } or {
    find isInhibited(T);
  }
}
```

Listing 6: Source code for isTransitionFireable pattern

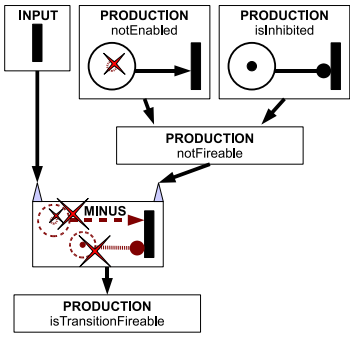


Figure 10: RETE matcher for the isTransitionFireable pattern

ing tuples (regardless whether they had passed the filter or not), so that whenever one of these elements experience change, the tuples containing it can be re-evaluated. If the result changes, the appropriate update tokens will be propagated. The node will monitor changes influencing a tuple until that tuple is finally removed by a negative update received from the parent node.

4.8 Construction

Given the definition of a pattern, the method to construct a RETE net for finding the matches of a pattern with good efficiency is a non-trivial task. The heuristics employed by VIATRA2 is a straightforward, but not necessarily optimal approach.

The key is perceiving a pattern as a collection of constraints imposed on subsets of the group of pattern variables. The construction algorithm processes these constraints one by one, and continues a connected sequence of nodes ("the line") to match larger and larger partial patterns, eventually using up all constraints and connecting the last node to the production node.

For simple entity and type constraints, pattern calls and miscellaneous cases (e.g. containment), (1) the appropriate input node or production node is accessed; (2) a *join node* will be attached as a child to it and also to the end of the line; (3) the join node will be prepared to match against variables that are involved in the constraint and are already introduced in the line. For negative application conditions, a *negative node* is used instead of the join node in an otherwise similar setup. A different setup is required for check conditions (and some miscellaneous cases including injectivity constraints), where a single *filtering node* (in this case, a *term evaluator node*) is attached at the end of the line.

When a child node is connected, it automatically receives all the tuples stored by the parent node as positive update tokens (and becomes subscribed for further updates); this way the construction and loading of the RETE net happens simultaneously, even though they are conceptionally separate. Input nodes and production nodes of called patterns are created upon first access; for production nodes, the matcher of the called pattern is also built at this time. This on-demand behaviour ensures that no unnecessary network parts are built and no unnecessary update notifications are delivered. The systems also supports extending an already built and used RETE network with new matchers if the need for new patterns arises.

5. PERFORMANCE

In this section, we analyze the runtime performance of our incremental pattern matcher implementation, comparing it to the built-in pattern matcher of the VIATRA2 framework [1]. As a benchmark, we make use of the Petri net firing example used in this paper.

5.1 Benchmarking considerations

Benchmarking a particular graph pattern matching algorithm can be performed by selecting a suitable set of test graphs and patterns and measuring the execution time. In this case, however, the target is to emphasize the effect of a conceptionally different pattern matching strategy (i.e. incremental matching) on the execution of a complete model transformation, in various usecases such as incremental synchronization and batch-like processing. Our benchmark transformation (Petri net simulation) suits the incremental pattern matching concept well.

As laid out in [19], benchmarking the execution of a complete transformation is difficult due to a number of reasons: (i) pratical transformations, such as the Petri net firing example, involve *non-determinism*, and (ii) external overhead which arises from the underlying platform (such as plugin management processing in the case of Eclipse-based tools like VIATRA2).

Non-determinism was eliminated by temporarily modifying the VIATRA2 execution environment so that non-deterministic constructs always return a pre-determined pick of possible matchings, thereby guaranteeing identical execution paths for all test runs.

To account for overhead, and to gain some insight into how the processing time is split up between the various phases of the execution (match computation, model manipulation, and code interpretation), we have employed a sophisticated Java profiler [23] which yielded precise (function-level) analysis results.

From a practical viewpoint, we were interested in two use cases: (i) the cost of firing the *first/next* simulation step, which is the most important use-case for incremental model transformations; (ii) a sequence of consecutive steps in a *batch execution* scheme, which shows how the RETE matcher performs in a more traditional model transformation application.

Thus, benchmark transformation was executed in two ways: (a) *As-Long-As-Possible* (ALAP) style execution where at each iteration, a new match is calculated (a fireable transition is picked) and the model manipulation (firing) is performed - matching the characteristics of the first use case; (b) *forAll* style execution where at each iteration, all matches (all fireable transitions) are calculated, and manipulation is performed in a pseudo-parallel fashion[2] - matching the characteristics of the second use case.

5.2 Results

The benchmark was performed on a live and bounded Petri net graph, consisting of 63 places, 69 transitions, and 302 arcs (including inhibitor arcs). The test runs consisted of firing 1000 transitions.

The aggregate results are shown on Fig. 11. Since our goal was to reveal the potential of the RETE-based approach, absolute execution times have been omitted; a more thorough benchmark evaluation is planned for future work. Relative execution times (the right-

[2]Conflicting transitions were accounted for by using an additional check before the actual execution of the firing rule.

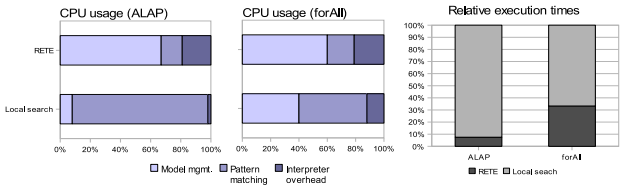


Figure 11: Benchmark results

most column of Fig. 11) confirmed that the RETE-based pattern matching approach is approximately an order of magnitude faster for the ALAP scheme, than VIATRA2's default, local search-based pattern matcher. (The 100% relative execution time corresponds to the mean time measured for normal execution - RETE runs scale at approximately 9%.)

This is supported by the CPU usage distribution data, which reveals that incremental updates can drastically reduce the overall execution time for transformations which require iterative pattern matching – in fact, the pattern matching phase is reduced so that it becomes comparable to the model manipulation phase, which is a significant advantage[3].

In the second scenario, which mimics the characteristics of common model transformations, the advantage is reduced to a factor of 3 (RETE runs scaled at approximately 33%). This is explained by the different execution path: in every iteration, **all** matchings for multiple transformation steps are determined (as described by the *forAll* scheme), and manipulation steps can proceed without further expensive match calculations. While this does not significantly impact the RETE matcher (absolute execution times were nearly identical), it clearly makes a large reduction for the local search-based approach.

5.3 Conclusion of measurements

In this section, we provided a preliminary overview about the possible performance gains with incremental pattern matching. It is important to stress that these gains can vary largely depending on factors such as:

- how model manipulation intensive a transformation is, and how the matching sets of patterns are affected (this manifests itself in the RETE network update overhead);
- transformation execution characteristics (ratio of *forAll* and ALAP style execution paths);
- memory constraints – speed gains can be reduced by memory management issues (such as garbage collection) arising due to the increased memory requirement of RETE networks.

Consequently, while the RETE matcher (and the incremental approach in general) may be a straightforward choice in certain cases, for general transformations the optimal solution can be attained by mixing pattern matching strategies appropriately. Thus, the implementation presented in this paper has been integrated into the newest version of the VIATRA2 framework in such a way that the transformation designer can specify on a per-pattern basis which matching strategy should be used.

6. RELATED WORK

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.

Attribute updates. The PROGRES [15] graph transformation tool supports an incremental technique called attribute updates [9]. At compile-time, an evaluation order of pattern variables is fixed by a dependency graph. At run-time, a bit vector is maintained for each model node expressing whether it can be bound to the nodes of the LHS. When model nodes are deleted, some validity bits are set to false, which might invalidate partial matchings immediately. On the other hand, new partial matchings are only lazily computed.

Incremental pattern manipulation. The transformation engine of TefKat [10] 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 [8]. The uniform, incremental handling of model elements and patterns can be considered a unique, advanced feature of the approach.

View updates. In relational databases, materialized views, which explicitly store their content on the disk, can be updated by incremental techniques like Counting and DRed algorithms [7]. As reported in [20], these incremental techniques are also applicable for views that have been defined for graph pattern matching by the database queries of [18]. However, [20] suffered from the inadequate support of incremental algorithms by the used underlying database and the strong restrictions being posed on the structures of the select query that defined the view.

Notification arrays. [21] 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. As a novelty, notification arrays are introduced for speeding up the identification of such partial matchings that should be incrementally modified. 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 [21] 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 used for graph transformation. RETE networks [6], 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 [4], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [11]. Our contribution extends this approach by supporting a more expressive and complex pattern language.

Other rule-based production systems. Improvements and alternatives of the RETE algorithm are now shortly surveyed. In the first two cases, the main goal is to reduce the high memory consumption of the RETE network.

TREAT [12] aims at minimizing memory usage while retaining the incremental property of pattern matching and instant accessibility of conflict sets. Only the input model elements and the (complete) matchings are stored, but no memories are used for partial patterns. TREAT is considered faster in certain conditions but less flexible than RETE.

RETE* [22] is a generalization of RETE that attempts to strike a balance between memory size and performance by keeping beta memories stored for frequently used nodes and generating them on-the-fly for the rest. The two extreme cases for the memory retention policy correspond to TREAT and RETE, respectively.

The LEAPS algorithm [2] is claimed to be substantially better than RETE or TREAT at both time and space complexity. The ap-

[3]RETE update costs are included in the "model management" category and the "pattern matching" category includes the RETE network construction phase.

proach can be characterized by lazy evaluation to avoid manifesting tuples unnecessarily, by depth-first firing, and by the introduction of timestamps to set up temporal constraints, which can be used for handling deletion efficiently.

7. CONCLUSION

In this paper, we presented an incremental graph pattern matching approach by adapting the well-known RETE networks to the rich pattern language of the VIATRA model transformation framework. In addition to the core RETE technique, we introduced new composite nodes for negative and disjunctive patterns or pattern composition.

We evaluated the performance of the incremental engine on a benchmark problem assessing the speed-up of incremental processing in case of as-long-as-possible rule applications using (extracts from) a Petri net simulator as a demonstrating example. Our measurements have also showed an incremental engine can outperform traditional (local search based) graph pattern matching techniques in the case of as-long-as-possible rule application. This speed up was not detected in previous benchmarking [19] due to the unavailability of supporting tools.

The incremental graph pattern engine is available in the current (new) VIATRA release, which allows the designers to select the most appropriate (incremental or non-incremental) pattern matching strategy for each pattern.

In the future, we plan to implement additional heuristics for optimizing the size of the RETE network, e.g. to automatically detect common subpatterns appearing in different graph transformation rules. Moreover, we also plan to carry out a more detailed experimental evaluation of the incremental matching strategy to identify those situations and problems when an incremental approach is beneficial.

8. REFERENCES

[1] Á. Horváth et al.. Generic search plans for matching advanced graph patterns. In *Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 57–68, Braga, Portugal, March 31- Apr. 1 2007. Electornic Communications of the EASST.

[2] D. Batory. The LEAPS algorithm. Technical Report CS-TR-94-28, 1, 1994.

[3] E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[4] H. Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammar based on the RETE-matching algorithm. In *Proc. Graph Grammars and Their Application to Computer Science and Biology*, volume 532 of *LNCS*, pages 174–189, 1991.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[6] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.

[7] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Proceedings*, pages 157–166, Washington, D.C., USA, 1993.

[8] D. Hearnden et al. Incremental model transformation for the evolution of model-driven systems. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 321–335, Genova, Italy, October 2006. Springer.

[9] S. E. Hudson. Incremental attribute evaluation: an algorithm for lazy evaluation in graphs. Technical Report 87-20, University of Arizona, 1987.

[10] M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *Proc. of the International Workshop on Model Transformation in Practice*, October 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/`.

[11] A. Matzner, M. Minas, and A. Schulte. Efficient graph matching with application to cognitive automation. In *Proc. of the 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance*, pages 293–308, Kassel, Germany, October 2007.

[12] D. P. Miranker and B. J. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10, 1991.

[13] Object Management Group. *QVT: Request for Proposal for Queries, Views and Transformations*. `http://www.omg.org`.

[14] A. Rensink. Representing first-order logic using graphs. In *Proc. 2nd International Conference on Graph Transformation, Rome, Italy*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.

[15] A. Schürr. Introduction to PROGRES, an attributed graph grammar based specification language. In *Graph–Theoretic Concepts in Computer Science*, volume 411 of *LNCS*, pages 151–165, Berlin, 1990. Springer.

[16] D. Varró and A. Balogh. The model transformation language of the viatra2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.

[17] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[18] G. Varró, K. Friedl, and D. Varró. Graph transformation in relational databases. *Journal of Software and Systems Modelling*, 5(3):313–341, September 2006.

[19] G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88, Dallas, Texas, USA, September 2005. IEEE Press.

[20] G. Varró and D. Varró. Graph transformation with incremental updates. In *Proc. of the 4th Workshop on Graph Transformation and Visual Modeling Techniques*, volume 109 of *ENTCS*, pages 71–83, Barcelona, Spain, December 2004. Elsevier.

[21] G. Varró, D. Varró, and A. Schürr. Incremental graph pattern matching: Data structures and initial experiments. In *Graph and Model Transformation (GraMoT)*, volume 4 of *Electronic Communications of the EASST*. EASST, 2006.

[22] I. Wright and J. Marshall. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *International Journal of Intelligent Games and Simulation*, 2(1):36–48, February 2003.

[23] YourKit, LLC. YourKit Java profiler homepage. `http://yourkit.com/overview/index.jsp`.