Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Incremental Pattern Matching in Graph-Based State Space Exploration

Amir Hossein Ghamarian, Arash Jalali, Arend Rensink

12 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Incremental Pattern Matching in Graph-Based State Space Exploration

Amir Hossein Ghamarian[1], **Arash Jalali**[2], **Arend Rensink**[3]

[1] ghamarian@cs.utwente.nl
[3] rensink@cs.utwente.nl
Department of Computer Science,
University of Twente, The Netherlands

[2] arash@netstairs.com
NetStairs.com, Inc.

Abstract: Graph pattern matching is among the most costly operations in any graph transformation system. Incremental pattern matching aims at reducing this cost by incrementally updating, as opposed to totally recalculating, the possible matches of rules in the graph grammar at each step of the transformation. In this paper an implementation of one such algorithm is discussed with respect to the GROOVE toolset, with a special emphasis put on state space exploration. Specifically, we shall discuss exploration strategies that could better harness the positive aspects of incremental pattern matching in order to gain better performance.

Keywords: Graph Transformation, Incremental Pattern Matching, State-space Exploration

1 Introduction

Graph transformation (GT) applications range from model transformation [GBG+06, FUJ06, VB07] to software verification [KK08, Ren04b]. Irrespective of its applications, GT consists mainly of the operations of finding the images of the rules in the host graph, i.e., matching, and transforming the host graph according to the rules. One of the major problems of GT in general is the complexity of the matching operation. Different algorithms have been proposed in the literature as optimized matching algorithms [HVV07, GBG+06, BÖR+08, BGT91]. These algorithms are mainly based on one of the two approaches of *search plan* [GBG+06, HVV07] or *incremental matching* [BÖR+08, BGT91].

In the search plan approach, a match for a rule is found based on a plan, i.e., a set of primitive matching operations, custom-made for each rule using a heuristic algorithm. Once the host graph on which the GT rules are to be applied, is modified by a rule application, the plans must be employed anew to once again find the matches of all the rules in the newly updated host graph.

Incremental matching, on the other hand, relies on a special data structure based on all the rules of the GT system, which is capable of maintaining information about partial and complete matches of all the rules within the host graph. Any changes made to the host graph are also applied to this data structure, and the information about partial and total matches is incrementally

updated where needed. In this way, all the matches of all rules are always readily available, with the extra cost of having to keep the network up to date.

Prior work in the GT literature suggests that incremental matching generally outperforms the search plan approach [BGT91]. In particular, a very efficient and intuitive algorithm for incremental pattern matching is based on the idea of RETE networks [For82]. However, incremental matching has only been used in tools that focus on model transformation [BÖR+08]. GROOVE [Ren04b], on the other hand, is a general purpose graph transformation tool with the main distinguishing capability of generating the entire (finite) state space of a graph transformation system. The current implementation uses search plans in its matching engine. In this paper, we investigate the use of a RETE-based algorithm in GROOVE, with the hypothesis that, because we actually apply all rule matches at every state, it should be possible to gain performance at least as much as for the VIATRA case reported in [BÖR+08]. In addition to extending RETE to support some special features of GROOVE rules, in particular the quantified rules described in [Ren04a, RK09], the main contributions of this paper are as follows:

- It introduces a state space exploration strategy that makes efficient use of RETE as the matching algorithm.
- It reports experiments showing that the resulting RETE implementation outperforms the previous search plan-based approach.

The next section explains the basic functionality of GROOVE. Section 3 specifies the RETE approaches together with our adaptation. Section 4 describes state space exploration and proposes an efficient strategy suitable for RETE. Section 5 shows the experimental results. Finally, Section 6 discusses directions for future work.

2 Introduction to GROOVE

In this section we briefly provide an overview of the GROOVE tool.

Graphs and rules.
Graphs in GROOVE consist of nodes and labelled edges. An edge is a binary arrow between two nodes, or from a node to itself. Node labels can either be node types or flags, which are a special kind of loop edge. Graphs are transformed by applying rules. A rule consists of: A) A pattern that must be present in the host graph in order for the rule to be applicable; B) Subpatterns that must be absent in the host graph in order for the rule to be applicable; C) Elements (nodes and edges) to be deleted from the graph; D) Elements (nodes and edges) to be added to the graph; E) Pairs of nodes that are to be merged. All these elements are combined into a single graph; colours and shapes are used to distinguish them.

Figure 1 shows a small example. The overall effect of the rule is to search for **A**- and **C**-nodes connected by a child-edge but without a parent-edge to a **P**-node, and to modify this by removing the child-edge and adding a parent-edge to a fresh **P**-node. For instance, the rule can be applied to the graph on the left hand side of Figure 2 in two ways, one of which results in the graph on the right hand side. (The other application removes the other child-edge.)

Quantification.
One of the special features of GROOVE is the support of universal quantification in rules (see

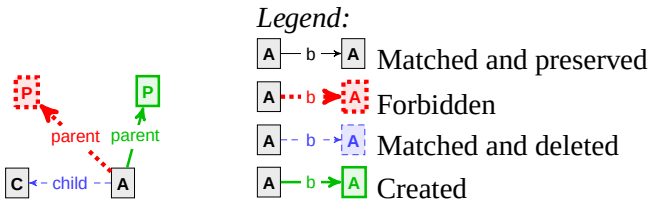
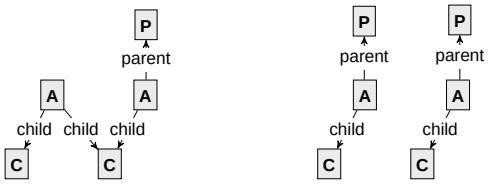


Figure 1: Example GROOVE rule and legend



(a) *Source graph* (b) *Target graph*

Figure 2: Example application of the rule in Figure 1

[RK09]). A universally quantified (sub)rule is one that will be applied to *all* subgraphs that satisfy the relevant application conditions, rather than just a single one as in the standard case. Such a rule can itself be much more concise, and also result in a smaller state space, than the equivalent set of rules that would ordinarily be needed. In fact, quantification can be *nested* in the sense that universally quantified rules can contain existential subrules, and vice versa. Among other things, this makes it possible to formulate powerful application conditions (see [Ren04a]).

State space exploration.
The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph transformation system) to a given start graph, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs, which is a rich source of information for further analysis.

In fact, GROOVE offers a choice of the exploration strategy to be used: *depth-first full exploration*, which also allows on-the-fly LTL model checking; *breadth-first full exploration*, which enables finding shortest paths to certain graphs; and *linear*, *random linear*, and *conditional* exploration which allow simulation without covering all states, for instance if the state space is too large. (In other words, for the latter strategies, GROOVE behaves like other GT tools.)

3 RETE basics

The RETE algorithm was first proposed by Forgy [For82] as an efficient means of pattern matching in production rule systems, in which the system is expected to apply those rules whose left-hand side (LHS) has a matching pattern in a given knowledge-base or state. This original algorithm was meant to be used in text-based expert systems; Bunke et al. generalized and adapted the idea behind the RETE algorithm to graph grammars [BGT91].

The basic idea behind RETE is that the pattern represented by the LHS of a rule can be gradually broken down into smaller sub-patterns all the way down to the basic elements of a graph

pattern, i.e. nodes and edges. By applying the same process to all the rules in a graph grammar, it is possible to construct a network of patterns, starting from simple nodes and edges, to more complex combinations that ultimately lead to the full pattern of the LHS of the rules. This network, called the RETE network, is itself a directed acyclic graph.

3.1 Classic RETE

Figure 3 shows a simple graph grammar with two rules and its corresponding RETE network. To avoid confusion, a node in the RETE network is usually referred to as an *n-node*. As originally defined in [BGT91], a RETE network consists of the following types of n-node:

Root This is the only node with no incoming edges. This node is in charge of receiving and passing nodes and edges down the network during runtime, i.e. when matching is being performed. The root is always succeded by edge-checker and node-checker nodes.

Edge-checkers Edge-checkers pass down the network those edges in the host graph that have a specific label. Some edge-checkers only accept loop edges while some accept any edge. In Figure 3c there are three edge-checkers under the root, one of which only accepts loop edges with the label 'current'.

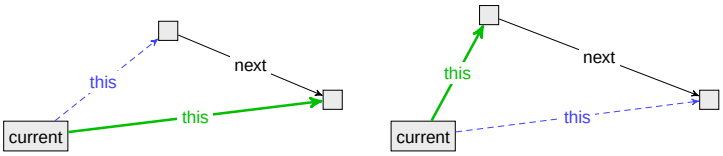
Node-checkers Node-checkers appear only immediately after the root, but their use in GROOVE is much more limited as nodes have no labels of their own in GROOVE. A node-checker can therefore be present in a RETE network only when the LHS of a rule consists of one or more isolated nodes.

Subgraph-checkers They combine the matches they receive from the upper n-nodes, also known as *antecedents*, and combine them into bigger matches if they overlap on certain nodes. These are indicated in Figure 3c by node equality relations at the bottom of each subgraph-checker. Subgraph-checkers always have two antecedents, commonly referred to as *left* and *right* antecedents. Matches received from these antecedents are stored in two memories called the left and right memories.

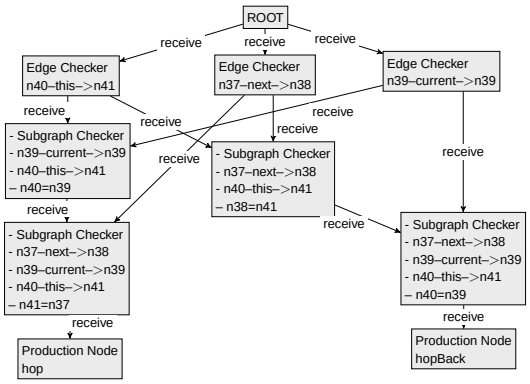
Production nodes These nodes represent the LHS of a rule. If a match reaches a production node, it would mean that it is a valid match for the LHS of that rule. The set of all possible matches of a production node in the host graph is called its *conflict set*.

The classic RETE algorithm for graphs consists of two phases: the *static construction* and the *dynamic* phase. During the static construction phase, the RETE network is built by going through each rule of the grammar starting with the very basic elements in its LHS, i.e. nodes and edges. Initially the network consists only of the root. When processing the first rule in the grammar, a separate edge-checker is created and added under the root for each edge in the LHS. The algorithm will then try to combine those edge-checkers into subgraph-checkers. Each subgraph-checker has a set of node equality relations, which signify the way the left and right antecedents of the subgraph-checker are connected to each other. This n-node merging process continues until a single subgraph-checker corresponding to the LHS of a rule is built. At this point a production node is added under the subgraph checker. For the subsequent rules in the grammar, the algorithm will try as far as possible to reuse the existing n-nodes. For details of the static RETE network construction algorithm please refer to [BGT91].

During the dynamic phase, sometimes simply referred to as *runtime*, the RETE algorithm will make use of the network to collectively find the conflict sets of all the rules in the grammar. At



(a) Grammar rule 'hop' (b) Grammar rule 'hopBack'



(c) Rete Network for 'hop' and 'hopBack'

Figure 3: Two rules and their associated RETE network

the very beginning of the dynamic phase, the RETE network is initialized by feeding all the edges and nodes of the host graph into the root node. The root node will pass down what it receives to its node- and edge-checker successors, and they in turn pass down legitimate matches to their subgraph-checker successors. The subgraph-checkers try to combine the partial matches they receive from their left and right antecedents if possible, and will pass the combined matches down to their own successors. This process continues until all the edges and nodes of the host graph are propagated through the RETE network. The matches that end up in the memories of production nodes will constitute the conflict set of the corresponding rule. The contents of all the memories of n-nodes collectively define the *state* of the RETE network.

The RETE state should be updated after a rule is applied to reflect the changes made to the host graph by that rule. This is where the incremental nature of RETE can be seen, as the network is updated by propagating only those elements that have been deleted or added by the rule's RHS.

3.2 Extensions added to RETE in GROOVE

RETE has already been implemented in other GT tools such as VIATRA [BÖR+08]. Specific features in each tool call for changes to the classic RETE algorithm to make pattern matching for rules that use them possible. Among such features in GROOVE are NACs, quantifiers, and support for rules with disconnected LHSs. In this section, we shall briefly cover these enhancements.

Quantifiers and nested conditions.
Universal and existential quantifiers in GROOVE have been implemented based on the idea of nested transformation rules [RK09]. Essentially in this scheme, a graph predicate is represented by a multi-level nested rule, where each subrule or subcondition is allowed to find its own matches as if it were an autonomous rule. Through a process called *rule amalgamation*, the

matches of the lower level conditions are collected based on the semantics of the quantifier at each level to form the overall set of matches for a grammar rule.

The RETE algorithm has been implemented in GROOVE to accommodate this nested predicate approach. In other words, the RETE network has been extended in GROOVE to support the idea of a subcondition rather than a quantifier, leaving the complexities of amalgamation out of the RETE network data structure. To accomplish this, we have added a new type of n-node to the RETE network called a *condition-checker*. A condition-checker is in many ways like a production node; in fact production nodes are implemented in GROOVE as specialized condition-checkers. During the static construction phase, the construction algorithm not only iterates through the grammar's rules but also through the sub-rules of any complex rule that has one or more levels of quantification in its LHS. During runtime, the RETE network can therefore be queried for the conflict set of any (sub)condition at any level. This approach not only helps keep the complexity of quantifiers and their semantics out of the basic pattern matching algorithm, but it also allows us to exploit any possible overlap of patterns among subconditions, as no discrimination is made between subgraph-checkers based on the level of subcondition they are associated with.

Negative application conditions.
In GROOVE, negative application conditions (NACs) are implemented as subconditions. In other words a rule with both positive and negative parts is represented as an upper level positive rule together with one or more sub-rules that consist of all the negative nodes and edges. A *root map* in the NAC specifies the connection points between the negative and the positive subgraphs.

Following this structure, the RETE implementation in GROOVE provisions for each NAC subcondition a special *composite condition-checker* that matches against the positive part of the main rule connected to the negative part represented by the NAC. Composite condition-checkers are constructed in the RETE network in a way that complete positive matches are gradually augmented with the elements of the negative pattern to form composite matches. So a composite match is prefixed by the positive match that is to be *inhibited* by the trailing negative pattern.

Once a composite match is found, the positive prefix of that match is reported to the positive condition-checker as an inhibited match. The positive condition-checker keeps a record of the number of times a positive match is inhibited, and so when it is queried for its conflict set, it will only return those positive matches whose inhibition counter is zero.

This implementation allows the RETE network to maintain its original simplicity and to provide the possibility of reuse of subgraph-checkers throughout the network while the prefix structure allows for rapid determintaion of inhibited matches. In VIATRA [BÖR+08] NACs are implemented using a special type of subgraph-checker, called a *negative node*, that passes through only those positive matches that do not join with any negative matches.

Rules with disconnected LHSs.
The original RETE algorithm as outlined in [BGT91] assumes that each rule's LHS is a connected graph. There, it is suggested that in order to support disconnected left hand sides, one can assume special *dummy edges* that bridge between the disjoint components of the LHS.

In GROOVE a slight modification has been made to the algorithm as well as to the structure of the RETE network so that any production node (or more generally a condition-checker) with several disconnected components in its LHS has a special type of subgraph-checker as its only antecedent, called a *non-connected subgraph-checker*, that receives the matches of all the disjoint

components of the LHS and produces all the combinations. Unlike the dummy edge approach, this scheme remains faithful to the general philosophy of reusing subgraph-checkers.

Domino removal.
For performance reasons, we have implemented the removal updates to RETE like a domino, i.e. matches are linked together and once a deleted edge match reaches the first subgraph-checker, GROOVE will follow the dependency links between smaller and larger matches and drops them from the memories they reside in rather than going through the process of overlap-checking and combining them in each subgraph-checker.

4 State space exploration

Having explained the static and dynamic part of the RETE network, in this section, we first briefly show how RETE is used in the context of other existing GT tools in linear strategies. Subsequently, we focus on how RETE can be used to explore the state space exhaustively.

4.1 Linear exploration and random linear exploration

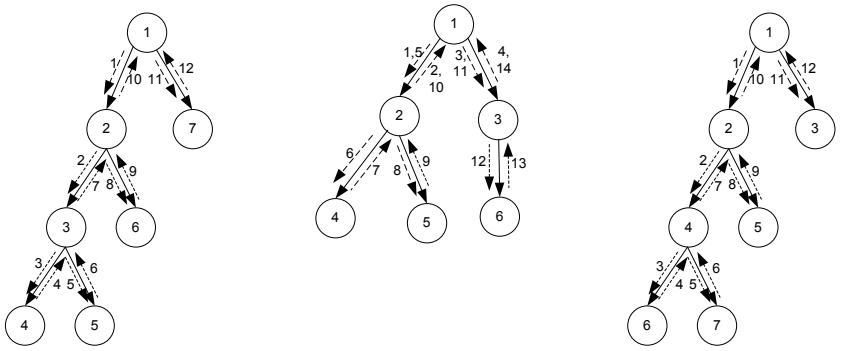
In most GT tools, especially those with a model transformation focus (see [FUJ06, VB07]), it is only needed to support a linear path through the state space. In other words, at each state, only one match out of all the matches of all rules is picked and applied to generate the next new state.

In RETE, the state of the network is updated based on the changes made to the host graph as a result of the application of a rule. This process repeats at each newly generated state. There are different methods for choosing a match at each state. In linear strategy, the selected match can be the very first match that is found, or rules can have priorities and the matches of the rules with higher priorities are chosen first. If this match is selected randomly, the strategy is called random linear. The overall scheme of this strategy is given in Algorithm *(Random) LinearStrategy*.

Algorithm *(Random) LinearStrategy*(G, H)
Input: A Graph Grammar G
Input: Input graph H
1. rete ←buildReteNetwork(G)
2. rete.initialize(H)
3. **while** desirable
4. **do** (rule, match) ←rete.selectAMatch()
5. (∗ match corresponds to the rule ∗)
6. δ ←applyMatch(rule, match, H)
7. Rete.update(δ)

4.2 Exhaustive state space exploration

There are different strategies for state space generation. GROOVE supports the main two algorithms, i.e., Depth-First Search (DFS) and Breadth-First Search (BFS). As explained in the linear strategy, the RETE network must be initialized in the beginning of the strategy, and as changes occur to the host graph, the RETE state also needs to be updated accordingly. The number of RETE updates as well as the size of updates (number of elements added and removed from the



(a) An example of DFS strategy using RETE.

(b) An example of BFS strategy using RETE.

(c) An example of a customized RETE DFS.

Figure 4: Different state space exploration strategies using RETE

graph) can substantially affect the performance of the exploration strategy. Figure 4a and Figure 4b show two sample state spaces traversed using the conventional DFS and BFS strategies respectively. They also show the order in which the RETE state is updated, if these strategies are used together with RETE as the matching algorithm.

Circles represent the states and the solid arrows between them represent transitions (rule applications) between two states. The dashed arrows signify updates performed on the RETE state. The number on each state is the order in which it is generated. The numbers next to each dashed line show the order in which the RETE state is updated, which could be more than once.

In Figure 4a the exploration starts from state 1, then in the DFS scheme the first match is selected and applied, consequently the successor state is discovered. The RETE state is updated according to the changes resulting from the rule application in state 1, which leads to state 2 (dashed arrow 1). The same procedure then repeats untill it reaches state 4, where there are no more successors and the exploration strategy needs to backtrack. The backtrack requires the matches of state 3 to be found again. In other words, the RETE update 4 is the reverse of update 3. This time however, the second match must be chosen from the RETE conflict sets, and this process continues until all states are fully explored. It is important to note that in every backtrack, a new (next) match ought to be selected, which implies that a fixed order among the matches needs to be maintained within RETE; a requirement that can be very expensive. It is also important to note that during each backtrack, the RETE network will inevitably find *all* the possible matches at that state, including those that have already investigated.

Figure 4b follows the same principle but in a BFS fashion. As can be seen, there are many duplicate RETE updates, as a state needs to be visited several times.

In order to avoid duplicate updates, as well as to avoid having to keep the conflict sets of the rules sorted in a fixed order, we propose another strategy which is more suitable for a matching algorithm like RETE, in which all matches are readily available at each state. Figure 4c shows this customized strategy, which we call *rete-dfs*. In this approach, once a state is reached, *all* the successor states are generated and stored in a stack. The stack a global data-structure that contains the generated but not yet fully-explored states. The next state to be explored is thus the state at top. This will cause the rest of the exploration strategy to continue in a DFS fashion.

The rete-dfs algorithm is explained in detail in Algorithm *reteDfsStrategy*. The RETE network

is built based on the grammar G, and then gets initialized with the host graph H. The host graph is added to the stack, and as long as there are states in the stack, the *next* procedure is called.

Algorithm *reteDfsStrategy*(G, H)
Input: A Graph Grammar G
Input: Input graph H
1. rete ←buildReteNetwork(G)
2. rete.initialize(H)
3. stack.push(H)
4. **while** stack is not empty
5. **do** *next(G, rete)*

Algorithm *next*(G, *rete*)
Input: A Graph Grammar G
Input: Updated RETE state *rete*
1. atState ←stack.top()
2. matchSet ←reteState.getMatchSet()
3. **for** every (rule, match) in matchSet
4. **do** newState ←applyMatch(rule, match, atState)
5. stack.push(newState)
6. *updateAtState(atState, rete)*

In procedure *next* all the matches of the current state are queried from the updated RETE state. Each rule match is applied according to its corresponding rule and the newly generated states are added to the stack. Then *updateAtState* is invoked in order to update the RETE state.

Algorithm *updateAtState*(*atState*, *rete*)
Input: The currrently explored state *atState*
Input: Updated RETE state *rete*
1. $\delta \leftarrow 0$
2. **if** atState $\neq$ stack.top()
3. **then** atState ←stack.top()
4. **else** (∗ no new state is put on stack in the last call to *next* ∗)
5. **repeat**
6. $\delta \leftarrow \delta +$ atState.getFromParentDelta() (∗ getFromParentDelta returns the changes when going from the parent of atState to atState ∗)
7. triedState ←atState
8. stack.pop()
9. atState ←stack.top()
10. **until** atState.parent() = triedState.parent() **or** atState = null
11. **if** atState $\neq$ null
12. **then** $\delta \leftarrow \delta^{-1} +$ atState.getFromParentDelta()
13. rete.update(δ)

If in procedure *next* any new state was added to the stack (atState is not equal to the top of stack), that would mean that the exploration is going forward. In that case, *updateAtState* just applies the changes made by the last rule application (δ) to that state and the RETE state is updated according to δ. Each state saved on the stack maintains the changes occurred when moving from its parent to itself; information which is obtained by calling *getFromParentDelta*.

However, if during the course of the procedure *next* nothing new was added to the stack, *updateAtState* would start backtracking by removing states from the stack. During backtrack, *updateAtState* accumulates in δ all the parent-to-child changes of the states taken from the stack.

The backtracking continues until *updateAtState* reaches a state where forward exploration is once again possible. This happens when the top of the stack is a sibling of (shares a parent with) the last state popped out of the stack. At this stage the backtracking phase is over and it is time to apply the accumulated changes stored in δ so far. However, since the changes stored in δ are from parent to child and we have been travelling in the opposite direction, δ needs to be inverted first (as indicated by δ^{-1} in line 12). In addition to the inverted δ, the update to the RETE state

Grammar	States	Trans.	32-bit 3GB			64-bit 48GB		
			Incr. [s]	Search Plan [s]		Incr. [s]	Search Plan [s]	
			rete-dfs	DFS	BFS	rete-dfs	DFS	BFS
append1	10535	28819	1.9	2.2	1.8	2.1	1.9	2.0
append2	707280	2603238	250.8	-	-	53.7	63.0	68.2
carPlat1	110366	369601	9.3	8.7	9.3	11.9	9.9	10.4
carPlat2	2988061	11929077	1859.1	-	-	221.5	206.4	209.8
crashCars1	278528	1236992	21.3	27.0	32.6	33.1	41.7	52.1
crashCars2	589824	2768896	242.7	276.9	470.8	66.8	91.4	108.8
petrinet1	2377	4645	2.1	4.4	4.5	2.7	5.1	5.2
petrinet2	23828	47656	29.9	93.5	94.5	36.9	81.7	84.0

Table 1: Exploration strategies' execution times

should also include one forward step from the shared parent to the sibling found at the stack top.

It is important to note that these δ's between the states are accumulated and the state of RETE gets updated only once. This approach decreases the amount of changes needed to be propagated through RETE. Maximum gain is achieved in cases when the changes made by successive rules cancel each other out, e.g. an edge is added by one transition and removed by another.

5 Experimental results and evaluation

We compared the rete-dfs exploration strategy, which uses RETE as an incremental matching algorithm, with DFS and BFS strategies, which use search plan for matching. We executed all of these strategies on four different grammars *car platooning*, *crashing cars*, *append* and a *petri net* (all available from GROOVE repository [Ren04b]). Each grammar was run on two different start graphs. *Car platooning* was taken from this year's GT tool contest, and the *crashing cars* and *append* have been devised by the GROOVE team. *Crashing cars* models a traffic light, and *append* simulates concurrent invocations of the append method of a list. The Petri Net example was inspired by the simple production system in [Han96]. These grammars are chosen as they not only make use of the special features of GROOVE grammars such as NACs and quantifiers but they also cover various levels of complexity in terms of the size of the rules' LHS and the size of host graph. The experiments were run on a machine with a dual Intel Xeon X5550 2.67GHz processor. Two series of experiments were performed over 32-bit and 64-bit Java Virtual Machines with 3GB and 48GB of RAM respectively. The results are shown in Table 1.

The first column shows the name of the grammar along with a sequence number representing the size of the start graphs used to run the exploration. The larger the sequence number, the larger the size of the corresponding start graph. The second and third columns are the numbers of states and transitions in the state space, respectively. The next six columns show the execution times of the three strategies in seconds, over the 32-bit and the 64-bit JVMs. A hyphen (-) in the execution time means that the experiment did not finish due to an out-of-memory exception. Please note that the reported time is the whole execution time, and the matching time is a fraction of this time. To make the total time a more precise indicator of the matching time, isomorphism checking has been turned off in these experiments.

Table 1 shows that as the size of the start graph increases, with the exception of the *car platooning* example, in all the other examples rete-dfs's speed performance surpasses the other two strategies, becoming more than three times faster in the *Petri Net* example. Speed is however not the only area where rete-dfs performs better than the other two. In the very case of *car platooning* as well as *append2*, it is evident that RETE is more well-behaved in terms of memory usage than Search Plan, as rete-dfs is the only strategy that has not run of out memory in the 32-bit mode. DFS and BFS have only been able to close in on rete-dfs after far more memory had been made available to them in the 64-bit case.

For the case of the *car platooning* example, RETE's inherent advantage in finding all the matches of a rule seems to have worked against it, as the grammar contains a number of non-modifying rules with often numerous matches in many states. Since in state space exploration, it is not necessary to find more than one match for each non-modifying rule at each state, we conjecture that considerable improvement could be gained in this example too, if a mechanism is put in place that could help to avoid finding redundant matches. One possible solution for this will be alluded to in section 6.

Overall, it can be inferred that RETE is ideal for cases in which the size of the rules' LHS or the start graph is relatively large but the effects of the rules are small and local, as RETE does not re-find matches that remain intact after a rule application. This is easily observed in the *Petri Net* example in which the host graph is large and the rule makes small local changes to it to move the tokens without changing the structure of the Petri Net. This is why the time gap between rete-dfs and the other two strategies has widened as the size of the Petri Net had increased.

6 Conclusion and future work

We proposed an adapted version of a RETE-based incremental pattern matching algorithm that could accommodate the special features and requirements of GROOVE. As our main contribution, we used this incremental pattern matching algorithm in exhaustive state space exploration. To harness the full potential of RETE we proposed the rete-dfs exploration strategy. The comparison of our proposed strategy with both DFS and BFS which use search plan showed that under common execution configurations rete-dfs outperforms other existing strategies (especially with larger start graphs) in most of the examples, and that in some of those examples the gap between rete-dfs and the other two can be narrowed only under very special circumstances with an unusual abundance of resources (e.g. 48GB of memory), while in cases where the host graph is large and the effects of the rules are local, this gap will only widen further as the size of the start graph increases irrespective of how much memory is made available to the strategies.

In our agenda for future work, first and foremost is the adding of support for some of the more advanced features in GROOVE, e.g. mergers, regular expressions and attributes. Furthermore, possible optimizations of the structure of RETE need to be more thoroughly investigated. Currently, the network is built in an ad-hoc way, leaving a lot of room for further optimization as the order in which the rules and their edges are processed during the static phase can affect the efficiency of the RETE network both in terms of speed and memory. Another improvement to the RETE implementation, currently under way, is the incorporation of a demand-based propagation of updates through the RETE network, in which updates are only propagated downwards when-

ever a condition checker node really needs a new match. Another clue for further optimization of the exploration strategy is that sometimes during backtracking, the size of update (δ) to be performed on the network can be very large; so much so that it might occasionally be less costly if the RETE states are stored in their entirety at all or some of the states.

Bibliography

[BGT91] H. Bunke, T. Glauser, T.-H. Tran. An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*. Pp. 174–189. Springer-Verlag, London, UK, 1991.

[BÖR+08] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró. Incremental pattern matching in the VIATRA model transformation system. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*. Pp. 25–32. ACM, NY, USA, 2008.

[For82] C. Forgy. RETE, a fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence* 19:17–37, 1982.

[FUJ06] The FUJABA Toolsuite. 2006. Homepage: http://www.fujaba.de.

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski. GRGEN: A Fast SPO-Based Graph Rewriting Tool. In Corradini et al. (eds.), *International Conference on Graph Transformations (ICGT)*. LNCS 4178, pp. 383–397. Springer, 2006.

[Han96] Z. Hanzálek. Petri Net Models for Manufacturing Systems. In Zalewski (ed.), *Proc of the IEEE Workshop on Real-Time Systems Education, Daytona Beach*. Pp. 99–105. IEEE Computer Society Press, Los Alamitos, Calif., 1996.

[HVV07] Á. Horváth, G. Varró, D. Varró. Generic Search Plans for Matching Advanced Graph Patterns. *ECEASST* 6, 2007.

[KK08] B. König, V. Kozioura. Augur 2 — A New Version of a Tool for the Analysis of Graph Transformation Systems. In Bruni and Varró (eds.), *Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*. Electronic Notes in Theoretical Computer Science 211, pp. 201–210. 2008.

[Ren04a] A. Rensink. Representing First-Order Logic Using Graphs. In Ehrig et al. (eds.), *International Conference on Graph Transformations (ICGT)*. LNCS 3256, pp. 319–335. Springer Verlag, 2004.

[Ren04b] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*. LNCS 3062, pp. 479–485. Springer, 2004. See http://sourceforge.net/projects/groove.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the geraniums: on nested graph transformation rules. In Boronat and Heckel (eds.), *Graph transformation and visual modelling techniques (GT-VMT)*. Electronic Communications of the EASST 18. EASST, 2009.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68(3):187–207, 2007.