Indexing RETE's Working Memory

Catering to Dynamic Changes of the Ruleset

Simon Van de Water, Thierry Renaux, Lode Hoste, Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
Elsene, Belgium
{svdewate, trenaux, lhoste, wdmeuter}@vub.ac.be

ABSTRACT

Complex Event Processing systems are used to detect patterns in large streams of events. These patterns are described by rules.

Due to changing requirements or updated security policies for example, these rules need to be frequently updated at runtime. Changing the ruleset at runtime, however, is a costly operation because the events that occurred in the past need to be re-evaluated.

In this paper we present an extension to the pattern-matching algorithm called RETE. Our extension aims to improve the performance of the re-evaluation of events when the ruleset is modified. More specifically, we optimise the addition of rules that match events by means of the inequality operators $>$, $\geqslant$, $<$ and $\leqslant$. We do this by introducing indexes to order the events that are stored in the working memory. Afterwards, we measure our performance improvements on targeted micro-benchmarks.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture; I.5.5 [**Pattern Recognition**]: Implementation

General Terms

Algorithms, Design, Performance

Keywords

Rete, Indexing, Dynamic ruleset

1. INTRODUCTION

Nowadays, businesses have to deal with large continuous streams of data. Financial institutions want to process a continuous stream of payments, withdrawals, money transfers, etc. in order to discover fraud by detecting suspicious patterns. For webshops, it is interesting to track purchases in order to automatically detect which products are popular and to ensure that these popular products are available in stock. In security, it is interesting to track a stream of in- and out-going network packets in order to detect system intruders.

Extracting useful information from these huge amounts of data in near real-time is a complicated task. There are many Complex Event Processing Systems (CEP) that offer a solution to this. In this paper we will specifically focus on one family of CEP called *production systems*. A production system is a computer program that uses a set of condition-action pairs (*rules*) in order to reason over a possibly large set of data with acceptable latency [1, 7, 8].
The order in which the rules are matched against the incoming data, as well as what intermediate results are stored, plays a very big role in the performance of the production system. This part of the production system is taken care of by a matching algorithm. This paper focuses on improving one aspect of such a matching algorithm, namely how it handles runtime modifications to the ruleset. The matching algorithm targeted in this paper is the RETE-algorithm [3]. Section 2 discusses the relevant parts of RETE.

As the behaviour of users changes over time (e.g. fraudsters continuously find new ways to avoid the system), the ruleset needs to evolve to cope with these behavioural changes. In current CEP systems based on RETE, rulesets evolve by manually updating existing rules and adding new rules to the ruleset at runtime. Updating the ruleset at runtime is an expensive operation because (i) the algorithm has to reconfigure the underlying data structure it uses and, more notably, because (ii) many updates to rules require a re-evaluation of the data that already resided in the system against rules that have been modified or added. However, it is of utmost importance that the reconfiguration and re-evaluation happens as quick as possible because new data is continuously entering the system and needs to be processed against the ruleset with as little latency as possible.

In this paper we present an extension of the RETE-algorithm that is aimed at improving the performance of modifying the ruleset at run-time. This extension will focus on improving the speed of checking literal constraints that involve a specific set of operators. The set of operators supported by our extension consists of greater than($>$), greater than or equal($\geqslant$), less than($<$), and less than or equal($\leqslant$). We will refer to this set as "inequality operators" in the remainder of this paper. We exclude $=$ and $\neq$ because these are already

supported by an extension that is complementary to ours [5].

The specification and implementation of our extension is described in section 3. In section 4 we validate our extension by comparing it with Drools, a state of the art production system that uses the RETE-algorithm with many extensions as its matching algorithm.
In section 5 we describe how we aim to extend the rule-language in the future to cater to the increasing need of rulesets that can easily be modified.
In section 6 related work is discussed. Section 7 finally comes to a conclusion.

2. RETE-ALGORITHM

The RETE-algorithm [3] was designed to speed up the matching of patterns with data (also referred to as "facts"). The patterns against which the data is matched are defined by a set of rules. Typically, these rules have a left-hand-side (LHS) and a right-hand-side (RHS). A rule (of which examples are shown in Listings 1, 2, 3 and 4) can be considered to be an if-then construct. The LHS describes the condition and the RHS describes the consequent.

RETE compiles these rules into a graph called the RETE-network which exists out of a Root node, Alpha-nodes, Join-nodes (also referred to as beta-nodes) and Terminal nodes. The Root node is an entry point for new facts and propagates them to the Alpha-nodes. The Alpha-nodes, which precede the Join-nodes in the RETE-network, evaluate literal conditions of the rules (e.g. Client.age == 18). Although the specification of the algorithm only supports literal equality checks, all well-known implementations of the algorithm today also support other operations such as "greater than" (>).
Once a fact successfully matches the Alpha-node(s) (i.e. the condition that is checked by the Alpha-node evaluates to `true`), it propagates to the Join-nodes. Join-nodes are responsible for upholding constraints that check conditions across multiple facts and will combine them pairwise if there is a match. The resulting pair is called a *token*. These tokens are the entities that are actually propagated throughout the RETE-network. When a token reaches a Terminal node (i.e. each condition of the LHS successfully matched with a token), the RHS of the rule that was matched by the token is executed.
We clarify this with the following example:

```
1 rule "ProcessPurchasePremiumMember"
2   when
3     c: Client()
4     PremiumMember( clientId == c.clientId )
5     p: Purchase(amount > 25, ! paidFor, c.clientId ==
        clientId)
6   then
7     c.chargePurchase(p, 5);
8 end
```

Listing 1: Example of a rule called `ProcessPurchasePremiumMember`

This rule expresses that whenever a `Client`, who is also a `PremiumMember`, makes a purchase for an amount that is bigger than $25, the `Client` will receive a discount of 5%.
In our example, the `Client` class has the fields `clientId`, `age` and `points`. The `Purchase` class has a `purchaseId`, an `amount` and `paidFor`, a boolean that indicates whether that purchase has been paid for already. Finally, the `PremiumMember` class simply refers to a `clientId`. The LHS of the rule is described between lines 2 and 6 whereas the RHS is described on line 7 (i.e. after the "then" keyword).
Line 3 takes a `Client` and matches it to the condition (i.e. in this example the condition is "is the fact a `Client`"). If a fact successfully pattern matches the condition (i.e. it satisfies the condition) on line 3, the client is bound to the variable `c` so that `c` can be used to refer to the matched `Client` fact in the remainder of the rule. Line 4 unifies the `clientId` of a `PremiumMember` with the `clientId` of `c`. Finally, in line 5, we check whether the amount of the `Purchase` is bigger than 25, whether it still needs to be paid and if the `Client` is a `PremiumMember`. The compilation of this rule results in a network with one root-node, five Alpha-Nodes, three Join-nodes and one Terminal node as illustrated in Figure 1. Every RETE-network has a root node which forms the entry point of the network. Each fact that is added to the *working memory* –which is a collection of all the facts that have been added to the network– is propagated to the root-node. The facts that pass through this root-node then get propagated further down to the nodes that are connected with the root-node. In this example, the Root node is connected to three Alpha-nodes (annotated as $\alpha 1$, $\alpha 2$, $\alpha 3$). These Alpha-nodes simply match the type of the fact that has just entered the network with a specific *fact type* (i.e. `Client` (line 3), `PremiumMember` (line 4) or `Purchase` (line 5)). Apart from the Alpha-node that only propagates facts of the type `PremiumMember`, the condition of line 4 was also compiled into a Join-node ($\beta 1$) that matches facts of the type `Client` with facts of the type `PremiumMember` based on their `clientId`. The condition on line 5 results in three Alpha-nodes; $\alpha 3$ matches each fact with the fact type `Purchase`, $\alpha 4$ matches each `Purchase` with the literal constraint `amount > 25` and $\alpha 5$ checks

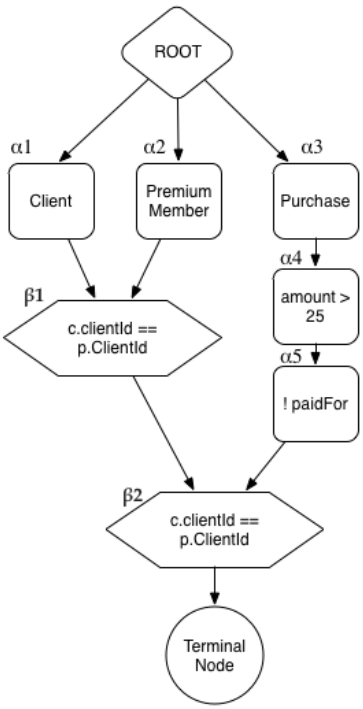

Figure 1: An example of a simple RETE-network.

whether the `Purchase` has been paid for already. If the `Purchase` was not paid for, Join-node $\beta 2$ will match the `clientId` of the `Purchase` with the `clientId` of a `Client` that matched a `PremiumMember` in Join-node $\beta 1$. If this match also succeeds, it means that the purchase was made by a premium member and that the fact can propagate to the Terminal node. The fact is then said to have *matched* the pattern (i.e. the purchase was made by a premium member and is eligible for a discount). Subsequently, the RHS (line 7) can be fired. The RHS in this case is a normal method invocation that is responsible for charging the `amount` of `p` to client `c` with a discount of 5%.

An essential part of RETE that is not yet discussed is that Alpha- and Join-nodes keep track of intermediate results. Because of this, facts only need to be matched by a node once. The Alpha-node that checks whether a fact is of the fact type `Client` will store all the facts that match with the condition of this node in its *Alpha-memory*. The same holds for the other Alpha-nodes in the network. Facts that are stored in memory are called tokens (which was already described above). Whenever a Join-node manages to unify two such tokens, it will store a tuple of these matching tokens in its memory (which is called the Beta-memory). In the example listed above, the advantage of this strategy becomes clear whenever a new Purchase that has an amount that is bigger than $25 and has not yet been paid for enters the system. All the `Client`s that are `PremiumMember`s are already unified by Join-node $\beta 1$, which means that the Beta-memory of this node already contains tokens for each `Client` that is a `PremiumMember`. Because of this, when the token of the `Purchase` fact is propagated from the Alpha-node $\alpha 5$ –which checks that the `Purchase` has not yet been paid for– to Join-node $\beta 2$, it is not necessary to start iterating over *all* `Client`s and `PremiumMember`s. The `clientId` of the `Purchase` just needs to be matched with the `clientId` of the tokens that were already residing in the Beta-memory of $\beta 1$, rather than with every `Client` in the working memory.

3. INDEXING WORKING MEMORY

In section 2 we observed that current systems are optimised to process events. However, rules and pattern definitions can also change. One challenge is that the running system cannot be shut down to change the ruleset because patterns should not be missed. Fortunately, the RETE-algorithm allows one to change the ruleset at runtime. However, this is a costly operation. Not only does the RETE-network have to be reconfigured at runtime, many updates to rules also require that facts that were already processed by the algorithm, need to be re-evaluated with the updated configuration of the network. If for example a new pattern is introduced to detect security threats, system administrators not only want to find threats in the stream of data from the moment the rule was added to the ruleset. They also want to find out whether the pattern has occurred in the (recent) past.

The modification or addition of a rule to the ruleset will typically lead to the creation of new nodes and dependencies in the RETE-network. Once the RETE network has been reconfigured, all the facts that reside in the *working memory* are propagated to new nodes. The longer the production system is used, the more events are stored. This means that changing the ruleset becomes more expensive because a lot of facts need to be re-evaluated. The extension of the RETE-algorithm presented in this paper improves the performance of matching facts with Alpha-nodes that involve the inequality operators $>$, $\geqslant$, $<$ and $\leqslant$. We do not focus on Alpha-nodes performing equality tests because this is already taken care of by an extension called Alpha-node-Hashing [5] as described in section 6.

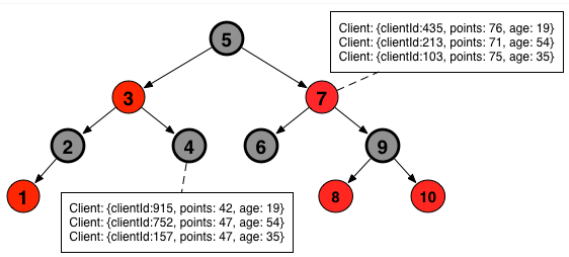

Figure 2: An example of a Red-black tree.

We aim to improve the performance by adding the notion of *Indexes* to RETE. Indexes, which stem from databases [4], are data structures that organise data in such a way that only a limited number of steps is necessary to navigate to a certain record. We take this idea and use it in RETE by introducing a data-structure that will order the working memory in such a way that every fact can –on average– be retrieved in $O(\log_2(n))$ steps where `n` is the amount of facts that reside in the working memory. Unfortunately, there are extreme cases in which the retrieval will have a performance of O(n). For example, when all of the facts in the working memory will successfully match with the new Alpha-node, the presented extension will not perform better than current CEP systems based on RETE. We further elaborate on this in section 4.
To get a better understanding of what an index is and how it improves the performance of modifying the ruleset, consider the following method-invocation:

```
1 createIndex("idxPoints", "Client", "Points", 10);
```

Listing 2: Creation of an index

This method will introduce an index called `idxPoints`. Such an index is implemented as a red-black tree [2]. Red-black trees are self-balancing binary search trees that offer a worst-case performance of O(log(n)) for insertions, deletions and searches. This index will consider the `points` of a `Client` as its key, whereas the value will be a bucket of facts. The last argument is used to define how big the range of the buckets should be. To understand this better, consider Figure 2. The Figure shows the red-black tree that was created by invoking the method of Listing 2. Every circle corresponds with a key in our red-black tree and each key has a bucket of facts as its value. In Figure 2, we only visualised the bucket for the keys 4 and 7. Figure 2 illustrates that each bucket (visualised by the rectangles) contains clients that fall within a range of `10` points. When a new key-value pair is added, the key that will be used is divided by the number that was passed as the last argument (in this case `10`). The resulting number is rounded down and we use that number

as the key. Because of this, node 7 for example, corresponds with a bucket that stores all Clients with points that fall in the range [70-79].

The current implementation of the indexes improve the performance of the addition of a rule whenever a rule introduces a new Alpha-node in the network. Consider a portion of the following simplified ruleset:

```
rule "BronzeMember"
    when
        c: Client( points > 10)
    then
        sendBronzeDeals(c);
end

rule "SilverMember"
    when
        c: Client( points > 20)
    then
        sendSilverDeals(c);
end
//Additional ruledefinitions that define GoldMember and DiamondMember.
```

Listing 3: A simple ruleset

These rules are responsible for sending special offers to our loyal clients. The more points a client has, the better the special offers the client receives. When a Client reaches a certain level (e.g. SilverMember, DiamondMember, etc.) the client will not only receive the offers for her/his level, but also the offers for all the preceding levels. SilverMembers for example (i.e. Clients that have more than 20 points) will receive offers for SilverMembers as well as for BronzeMembers.
Now imagine the following rule was added at runtime:

```
rule "PlatinumMember"
    when
        c: Client( points > 50)
    then
        sendPlatinumDeals(c);
end
```

Listing 4: PlatinumMember rule

This rule will add a new Alpha-node to the RETE-network that will check whether a Client's points are bigger than 50. In traditional RETE, each fact that sits in the working memory needs to be matched against this new Alpha-node. By employing the indexes, this is no longer the case. Instead of propagating all facts to the new Alpha-node, we only need to propagate the values of the buckets that are stored with keys 5 - 10. We can omit the propagation of the facts that are stored with keys 1-4 because the red-black tree, which uses the points of a Client as its key, ensures that all the facts that are associated with these keys have too few points to actually match with the test of the new Alpha-node. The values that correspond with keys 5-10 can automatically be propagated to the node that depends on the new Alpha-node because we know that the points of all these values are bigger than 50. In other cases, however, it can be that a limited set of facts need to be checked by the new node. If the new rule would check for all Clients that are bigger than 55 for example, it would have to recheck all the facts that are stored in the bucket that is associated with key 5 because some of the facts in this bucket will have more than 55 points whereas others will not. This process could be sped up as well by implementing the buckets as an ordered data-structure (e.g. a red-black tree, AVL-tree, etc.). The facts that correspond with keys 6-10, however, can still be propagated without performing checks.

Another use case that could benefit from using indexes is a RETE-network with more than one Alpha-node performing a test on the same property as illustrated in Listing 3. Because the red-black tree already orders the facts as they enter the system, they no longer need to be explicitly checked by each Alpha-node. This works in a similar way as how the system decides whether a fact should be propagated to the new Alpha-node in the use-case explained above with the new Alpha-node. Our current implementation, however, does not yet cater to this use-case.

3.1 Implementation

Our index extension to RETE is implemented in Drools 6.2. Rather than implementing the red-black tree in the root-node, we decided to implement the red-black tree in the ObjectTypeNodes. ObjectTypeNodes, which are specific to Drools[1], exist to make sure that the algorithm does not do redundant work. Instead of matching every new fact with each node that depends on the Root-node, the Root-node knows – by means of hashing – to which ObjectTypeNode it should propagate each new fact. Because of this, facts no longer need to be explicitly checked against every type that exists in the RETE-network (i.e. the first three Alpha-nodes in Figure 1 are replaced by ObjectTypeNodes).

Each ObjectTypeNode holds a set of all facts that were propagated to it. Whenever an index is created for the type that corresponds with the ObjectTypeNode, an instance of RbTreeDuplicates is added. RbTreeDuplicates is a modified version of the RbTree which stores a bucket of values for each key rather than one value per key.
In order to fill our indexes with data, we made sure that whenever an object is asserted to a certain ObjectTypeNode, it is also added to our indexes.
To ensure that not all facts are propagated to a new Alpha-node that was connected to an ObjectTypeNode, we modified the updateSink-method. This method is responsible for propagating facts from the ObjectTypeNode to new nodes that are added to the network. Before propagating the facts, we check whether we have an index for the property that is checked by the new Alpha-node. If such an index exists, we use the red-black tree of that index to find the value that is nearest to the value that is being compared with in the Alpha-node (e.g. 50 in the example of Listing 4). After finding the nearest node, we just add the values of its buckets and navigate to the next or previous key if necessary (i.e. if the Alpha-node performs a bigger-than or smaller-than test). If no such index is found, we fall back to the standard implementation of Drools.

4. BENCHMARKS

For the evaluation of our extension, we conducted a series of benchmarks. We used a server containing an AMD Opteron 6300 "Abu Dhabi" 16 core processor and 128GB of memory with NUMA properties. Dynamic frequency scaling was disabled to reduce measurement errors. The server uses Linux Ubuntu 15.04 with Oracle Java 1.8.0_51.

[1]for more information, visit https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch03.html

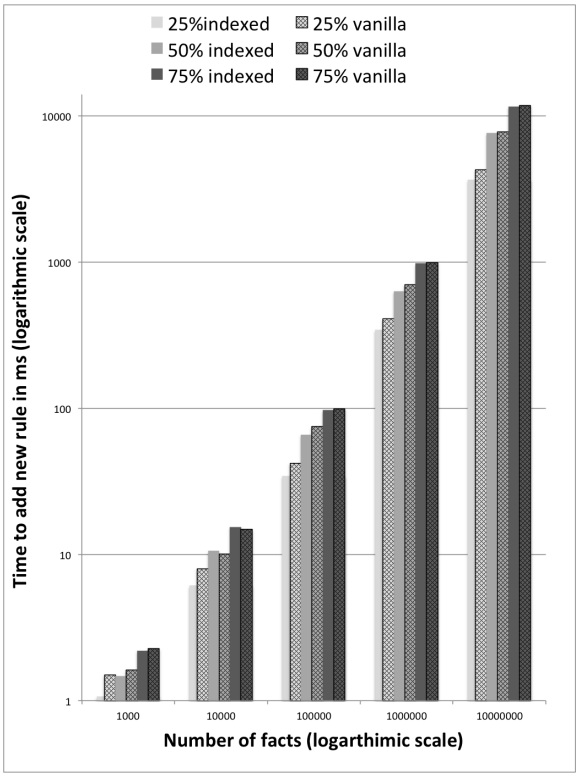

Figure 3: A comparison of the runtime of our extension and Drools with a varying number of facts in the working memory.

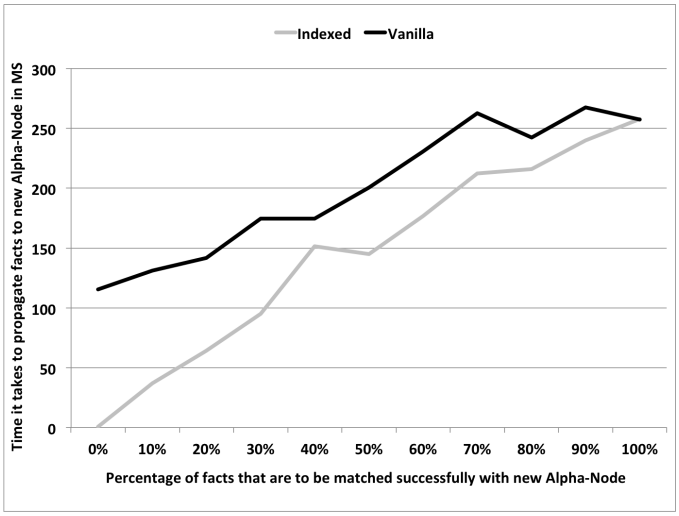

Figure 4: A comparison of the propagation of the working memory to the new Alpha-node of our extension and Drools with a varying percentage of facts that successfully match the new Alpha-node.

% of successful matches	Our extension to new Alpha-node	Drools
0%	0,7ms	115,43ms
50%	144,8ms	200,53ms
100%	257,9ms	257,23ms

Table 1: Exact timings of propagation of the facts in the working memory to the new Alpha-node

4.1 Scalability

The first set of benchmarks indicate how our extension scales compared to an unmodified version of Drools 6.2. To do so, we created different sets of data that contain a varying number of facts. We then pass all the facts to the algorithm and fire all the rules that were already present in the ruleset. We then continue by updating the ruleset and re-evaluating all the facts that were inside the working memory. We measure the time it takes to update the ruleset and re-evaluate all the facts.

In total we measure the performance of 15 datasets. We created datasets of different sizes; 1.000 facts, 10.000 facts, 100.000 facts, 1.000.000 facts and 10.000.000 facts. For each of these sizes we created 3 datasets; each containing facts of which 25%, 50% and 75% match successfully with a new Alpha-node. The results are presented in Figure 3. The numbers represent the average value of processing each dataset 30 times. To observe how the extension of the algorithm deals with a varying number of facts, we needed to make our x- as well as y-axis logarithmic in order to present meaningful information. The graph illustrates that up until 10.000.000 facts our extension to RETE as well as Drools have a similar increase in time with respect to the size of the working memory. The graph also shows that as soon as the working memory grows larger than 1.000 facts, our extension performs better than Drools. It can also be observed that as the number of facts that successfully match the new Alpha-node grows smaller, the performance gain of our extension compared to Drools grows bigger. This observation is better illustrated in the next set of benchmarks.

4.2 Propagation of facts

This set of benchmarks illustrates how long it takes for facts to propagate from the working memory to a new Alpha-node The time it takes to actually evaluate the facts is not taken into consideration in these benchmarks.

To do so, 10 different datasets were created, each containing 1.000.000 facts The difference between the datasets is the percentage of facts that successfully match with the new Alpha-node. The values presented in Figure 4 are the averages of processing each dataset 30 times.

This graph clearly illustrates that if there are no facts that successfully match the new Alpha-node, our extension needs virtually no time (0.7ms) to propagate the facts to the new Alpha-node as illustrated in Table 1. Drools on the other hand takes up to 115,43ms. This means that in the best case, our extension is 164 times faster than Drools in propagating the facts from the working memory to the Alpha-node. This behaviour is expected since no fact is propagated to the Alpha-node by our extension.

It can also be observed that as the number of facts that successfully match the new Alpha-node grows, the difference between our extension and Drools of the time that is needed to propagate the facts reduces. For the average case in which 50% of the facts successfully match the new Alpha-node, our extension (144,8ms) performs 28% faster than Drools (200,53ms).

In the worst case scenario where all of the facts in the

working memory match the new Alpha-node, we even see that our extension (257,9ms) performs slightly worse than Drools (257,23ms). This is due to the fact that Drools fetches facts from an array in O(1). If our extension however needs to jump from one node to the subsequent node in the red-black tree it takes O(log(n)) where n is the number of keys in the red-black tree.
Except for this extreme case, however, we see that our extension outperforms Drools for every percentage.

5. FUTURE WORK

The current extension of the algorithm requires developers to explicitly create indexes. We envision to extend the algorithm such that it can deduce when it could be beneficial to create an index for a certain property by analysing the ruleset and facts. For Listing 3 for example, the algorithm should be able to deduce that it would be beneficial to create an index on the `points` field of the `Client`-class because there are so many rules that put a literal constraint on `points`.

Apart from improving the performance when the ruleset is modified, we also believe that the rule-language should be extended with support for *meta-variables* and *meta-rules* in order to make it easier for rule-developers to update the ruleset.

```
1 Var X = 500;
2 rule "DetectPremiumMember"
3     when
4         c: Client(points > X)
5     then
6         insertLogical(new PremiumMember(c.clientId));
7 end
```

Listing 5: PremiumMember rule

Listing 5 illustrates the purpose of *meta-variables*. This rule decides which `Client`s become `PremiumMember`s based on their `points`. The rule-definition illustrates that for the moment, a `Client` is considered to be a `PremiumMember` when they have more than 500 `points`. To attract more people to their store, the owners can decide that `Client`s will be treated as `PremiumMember`s as soon as they gathered 200 `points`. Current matching engines forces developers to explicitly change a ruleset. By adding the notion of variables, this could be done by simply changing the variable. We envision that these variables can be changed through other rules (see below) or by regular imperative Java-code (in the case of Drools). Moreover, our extension could anticipate to rules with such variables by automatically creating an index for them. When the developer introduces a variable, it would be safe to deduce that this rule (with its corresponding Alpha-node) is expected to change often and would thus benefit from the extension presented in this paper.
The behaviour of meta-variables can be mimicked on existing systems by introducing a new fact-type that could be used to replace `X` in the example above. The details of this workaround are omitted because it requires an additional Join-node in the RETE-network which are easily outperformed by the Alpha-nodes that would be required by our extension.

Due to changing functional requirements and updated security policies, it is important you can change rulesets. Because of this, we envision a rule language that goes further than merely describing which patterns should be matched.

This rule language will expose information about rules themselves, making it possible for developers to write rules that reason over this information. Such rules are called *meta-rules*.
Apart from reasoning over the information of other rules in their LHS, meta-rules can also change the ruleset in their RHS. Writing meta-rules allows developers to already anticipate to changes that are expected to happen in the future. Listing 6 provides an example of how such a rule could be expressed. This code is pure fictional.

```
1  meta-rule "DetectBadSalesDay"
2      when
3          dpmRule: Rule(name = "ProcessPurchase")
4          dpmRuleFires: dpmRule.fires
5          dpmRuleFiresToday: dpmRuleFires.date (Today)
6          hoursOpened: CurrentHour - 10
7          (dpmRuleFiresToday / hoursOpened) < 20
8      then
9          X = 200;
10 end
```

Listing 6: An example of a meta-rule

This rule describes a *bad day* for a store. A day is considered to be bad when we have an average of less than 20 purchases per hour after being open for at least 4 hours. If the store is having a bad day they will try to improve their sales for that day by letting `Client`s with as little as 200 `points` enjoy the discounts that `PremiumMember`s benefit. Line 3 binds `dpmRule` to the rule called "ProcessPurchase". Line 4 is responsible for retrieving meta-information (i.e. every time a certain rule was fired). Since we are only interested in the amount of times the rule has fired today, line 5 filters out all the `fires` that happened before today. On line 6, we calculate how long the store has been opened today. Since the store opens at 10:00, we just have to deduce `10` from `CurrentHour` which holds the current hour in 24-hour format. After checking whether the store has been open for more than four hours, we check how many purchases per hour we averaged on that day. If that amount is below 20, we decide to change the value of `X` to `200`. This will lead to a re-evaluation of the rule of Listing 5, ensuring that all `Client`s with more than 200 `points` get a discount on every purchase that day since they are considered to be a `PremiumMember` for the remainder of the day.
Apart from knowing how many times a rule was fired, other meta-information is useful as well. Currently, we are investigating which meta-information should be exposed in our envisioned meta-rule language.

6. RELATED WORK

Over the past decades, many extensions to RETE have been suggested. To the best of our knowledge however, none of these extensions aim to specifically reduce recomputation of Alpha-nodes when changing the ruleset at runtime. In this section, we discuss a number of optimisations that improve the performance of dynamically changing the ruleset.
TREAT [6] differs from the original specification of RETE in that it does not maintain a Beta-network in memory over multiple operations. TREAT only keeps track of the Working Memory and the conflict set[2]. Because of this, inserting facts becomes very expensive as the beta-network needs to

[2]The conflict set is the set of facts that have reached a terminal node

be rebuilt every time a fact is inserted. On the other hand, deleting facts becomes inexpensive because there is no need to backtrack through the RETE-network to ensure that the fact is removed from the Beta-memories. Another advantage of TREAT is that its memory consumption is significantly lower than RETE. Because the Beta-network is not maintained, adding rules dynamically will perform better compared to RETE. However, when existing facts also need to be matched against the new rules, the performance deteriorates significantly because the beta-network needs to be created for every fact that is inserted.
The matching algorithm implemented in Drools is called RETE OO, which is an extension of the RETE-algorithm. Amongst other optimisations, the most interesting one, which is called Alpha-node-Hashing, improves the performance of matching Alpha-nodes that have an equality constraint (==) [5]. When there are multiple Alpha-nodes that match an equality constraint on the same field, a HashTable is created. The keys are the literal values against which the facts are checked. The value is the Alpha-node itself. This allows the algorithm to decide in O(1) to which Alpha-node a fact should be propagated, rather than matching the fact against each Alpha-node. Unlike the optimisation suggested in this paper, it does not take care of improving the performance of matching Alpha-nodes that involve inequality operators.

7. CONCLUSION

In this paper we presented an extension to RETE that aims to improve the performance of the algorithm in two specific cases:

- When the ruleset is modified in such a way that new Alpha-nodes are added to the RETE-network that involve inequality operators ($>$, $\geqslant$, $<$, $\leqslant$).
- When the RETE-networks contains multiple Alpha-nodes that involve inequality operators ($>$, $\geqslant$, $<$, $\leqslant$) on the same field of a certain object.

The extension integrates the notion of an index (implemented as a red-black tree) into the *working memory* of the RETE-algorithm. This means that the working memory will be an ordered collection of facts in which its predecessors and successors can be found in O(log(n)) (where n is the number of keys). We can use this property to easily find all the facts that will successfully match the Alpha-nodes without actually performing the test.
Benchmarks have shown that our extension to RETE performs better than Drools on average. In the best-case scenario (viz. no facts should be propagated to the new Alpha-node) our extension is 164 times faster than Drools. In the average-case scenario (viz. 50% of the facts should be propagated to the new Alpha-node) our extension is 28% faster than Drools. Finally, worst-case (viz. 100% of the facts should be propagated to the new Alpha-node) our extension is 0.2% slower than Drools.

8. REFERENCES

[1] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

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

[4] H. Garcia-Molina. *Database systems: the complete book.* Pearson Education India, 2008.

[5] D. Liu, T. Gu, and J.-P. Xue. Rule engine based on improvement rete algorithm. In *Apperceiving Computing and Intelligence Analysis (ICACIA), 2010 International Conference on*, pages 346–349. IEEE, 2010.

[6] D. P. Miranker. *Treat: A better match algorithm for ai production systems; long version.* University of Texas at Austin, 1987.

[7] M. Proctor. Drools: a rule engine for complex event processing. In *Applications of Graph Transformations with Industrial Relevance*, pages 2–2. Springer, 2012.

[8] G. Riley, C. Culbert, R. T. Savely, and F. Lopez. Clips: An expert system tool for delivery and training. In *Proceedings of the Third Conference on Al for Space Applications*, 1987.