Relative Temporal Constraints in the Rete Algorithm for Complex Event Detection

Karen Walzer, Tino Breddin and Matthias Groch
SAP Research CEC
Chemnitzer Strasse 48
Dresden, Germany
karen.walzer@sap.com

ABSTRACT

Complex Event Processing is an important technology for information systems with a broad application space ranging from supply chain management, systems monitoring, and stock market analysis to news services. Its purpose is the identification of event patterns with logical, temporal or causal relationships within multiple occurring events.

The Rete algorithm is commonly used in rule-based systems to trigger certain actions if a corresponding rule holds. Its good performance for a high number of rules in the rulebase makes it ideally suited for complex event detection. However, the traditional Rete algorithm is limited to operations such as unification and the extraction of predicates from a knowledge base. There is no support for temporal operators.

We propose an extension of the Rete algorithm to support the detection of relative temporal constraints. Further, we propose an efficient means to perform the garbage collection in the Rete algorithm in order to discard events after they can no longer fulfill their temporal constraints. Finally, we present an extension of Allen's thirteen operators for time-intervals with quantitative constraints to deal with too restrictive or too permissive operators by introducing tolerance limits or restrictive conditions for them.

Categories and Subject Descriptors

D.1.6 [**Software**]: Logic Programming

General Terms

Algorithms

Keywords

Rete Algorithm, Event, Temporal, Rule-based

DEBS '08, July 1-4, 2008, Rome, Italy

1. INTRODUCTION

Complex event processing (CEP) is a technology to monitor and control information systems driven by events. It is applied, for instance, in business process or supply chain monitoring to detect complex events consisting of single events with logical, temporal or causal relationships. Many designs have been proposed for the necessary event detection, for instance [5] uses finite state automata and [10] showed the efficiency of using the Rete algorithm for the detection of a high number of complex events.

Traditionally, the Rete algorithm [6] is used in expert systems for production-based logical reasoning, matching a set of facts against a set of inference rules. A rule defines a predicate and the action that should take place, if the predicate holds. This corresponds to defining a complex event combining single events and executing an action such as the creation of a new event when the complex event is detected. The algorithm can efficiently handle a high number rules in the rulebase. Thus, it is capable of dealing with a high number of complex event patterns.

Business processes are often long-lasting and contain temporal dependencies. Software for *Business Activity Monitoring* (BAM) therefore needs to be able to react to exceptional situations and determine the current business state considering temporal relationships. Complex event processing can be used for this purpose, but requires the definition of complex events which consider temporal relationships among single events. Relative temporal constraints are needed to capture the order of occurrence of events, for instance to create an *alarm* event only if two single *DeliveryFailed* events occur within five minutes.

The traditional Rete algorithm does not support temporal operators, but is limited to operations such as unification and predicate extraction. A simple matching using comparisons with timestamp attribute values is possible and extensions to allow for the detection of relative relationships have been suggested, e.g. in [2]. However, existing CEP systems (not based on Rete) commonly use time-point semantics to represent event timestamps, e.g. [18]. This can lead to semantic misinterpretations [19], since no duration can be specified for events. For long-lasting events such as those created during business monitoring, we therefore propose the usage of interval-time semantics [3] for event timestamps to allow for the definition of a duration for each event. Furthermore, existing garbage collection mechanism use only a de-

fault life-time for events after which they are discarded and can no longer contribute to complex events. However, this does not consider the temporal relationsships of the events, and can lead the undesired discarding of events.

In this paper, we present an extension of the Rete algorithm to support the detection of relative temporal relationships between events and thus to enable complex event processing. We use interval-based time semantics for event timestamp definition. In particular, we make the following contributions:

- We describe a method of how to incorporate the detection of relative temporal constraints in Rete. The known operators for interval-time introduced by Allen [1] are often too permissive or too restrictive for real-world events and thereby limit the expressiveness of complex event defintions. We propose their extension to allow for quantitative constraints by providing tolerance limits or exact time limits.

- We develop a means to perform time-driven garbage collection in Rete to delete events which can no longer fulfill their temporal constraints. For this, we use reference counting and adopt ideas from incremental and concurrent garbage collection.

The remainder of this paper is organized as follows. We define the used terms, give a motivating example, compare the different time semantics and describe the Rete algorithm in Section 3. Section 4 presents an overview of our approach for relative temporal constraints. Section 5 discusses garbage collection in Rete and explains an algorithm to achieve it. We give a brief discussion of our implementation in Section 6. Then, we present an overview of related work in Section 7. Section 8 concludes the paper.

2. TEMPORAL SUPPORT IN RETE

In the following, we present our notion of instantaneous and complex events. We give examples for Business Activity Monitoring of a Pizza service and present how a point-based semantic for events can result in unintended event detection. Finally, we introduce the Rete algorithm as a basis for our event detection.

2.1 Definitions

Events indicate a state change of the world. They are n-tuples containing an arbitrary number of data items. For instance, an *OrderArrival* event contains data related to an arriving order, e.g. the customer name and address, the ordered items and their quantity as well as a timestamp denoting the occurrence time of the order.

Let $\mathbb{T} = (T; \leq)$ be an ordered time domain. Then, let $\mathbb{I} := \{[t_s, t_e] \in T \times T | t_s \leq t_e\}$ be the set of time intervals with t_s as start and t_e as the end time-point of the interval. Let $\mathbb{D}$ be the set of atomic values where atomic values are elementary data types, such as strings. Then, let $\mathbb{E} := \{(k_1, .., k_n, k_{n+1}, ts, te) | k_i \in \mathbb{D}, [ts, te] \in \mathbb{I}\}$ be the *set of events*.

An event is characterized by the time of its occurrence, which is stored as the event's timestamp.

Instantaneous events are single events which occur at a certain point in time. They have a duration of zero, $t_s = t_e$. The aforementioned *OrderArrival* event is an example of an instantaneous event.

Complex events describe the occurrence of a certain set of events (instantaneous or complex) having relationships defined using logical and/or temporal operators. These operators commonly include conjunction, disjunction and negation as logical operators and can include temporal operators such as *before* and *after* to define the order of events. The supported operators vary for the different CEP systems.

2.2 Motivating Example: Pizza Service

Consider a Pizza delivery service *HappyPizza* which offers Pizza using a Webpage and uses handhelds to confirm successful Pizza deliveries. *HappyPizza* is part of a Pizza chain which uses Business Activity Monitoring to automate and monitor its processes. Events are used to inform of the current state of the processes, i.e. for instance at the beginning or the end of processes, e.g. a Pizza delivery, an event is automatically created.

An example for a complex event is the *startOrderProcessing* event which is created when an order arrived from the Pizza service's website and a customer check determined that it came from a valid customer, i.e. the customer has an existing street name and number. The creation of the complex event can be automated using the following rule.

```
IF
    OrderArrival AND CustomerCheck AND
    OrderArrival.Customer = CustomerCheck.Customer
    AND Customer.validCustomer = true
THEN
    create startOrderProcessing (OrderId)
```

As can be seen, the rule consists of a conditional part (IF-part) and a resulting action (THEN-part) to be performed. The rule expresses that after the instanteous events *OrderArrival* and *CustomerCheck* arrived having the same valid customer, the complex event *startOrderProcessing* is created for the order. The complex event can then result in the notification of the delivery service staff to process the order and make the Pizza.

An complex event using temporal constraints is, e.g., the *PizzaForFree* event, which expresses that a customer gets her Pizza for free, if she has to wait for more than hour from the time of her order. The event creation is automated using the following rule.

```
IF
    OrderArrival BEFORE (1h) DeliveryArrival AND
    OrderArrival.OrderId = DeliveryArrival.OrderId
THEN
    create PizzaForFree (OrderId)
```

If the instanteous event *OrderArrival* is occurring 1h before the instanteous event *DeliveryArrival* and both refer to the same order, then the complex event *PizzaForFree* is created for this order. The complex event can result in the notification of the delivery person or be considered in the Pizza service's accounting. This example illustrates the importance of temporal constraints to describe complex events.

2.3 Point-based vs. Interval-based Semantics

An event is characterized by the time of its occurrence, which is stored as the event's timestamp. For the timestamp definition, point-based or interval-based semantics can be used. The former describes an event as being instantaneous, i.e. it has no duration, but just occurs at a point in time when it comes to existence. The latter allows an event to have a duration represented by an interval bound by the start and end-time of the event.

In point-based semantics, a complex event only has the timestamp of the last occurring event contributing to it. In interval-based semantics, the duration of a complex event is bound by the start instant of the first and the end instant of the last contributory event, assuming the first and last event being instantaneous. An interval-based timestamp definition for complex events therefore allows a better expression of the differences in occurence time of the instantaneous events that contribute to a complex event.

Figure 1 illustrates the difference in event detection when using point-based timestamp semantics in contrast to interval-based semantics for complex events. It shows the occurence time of the instantaneous events A, B and C and an example of nested BEFORE operators. A BEFORE operator $BEFORE(E1, E2)$ defines that event $E1$ should occur before or at the same time as $E2$, i.e. the timestamp of $E1 \leq E2$. Adapted from [19] Figure 1 illustrates an example for *HappyPizza*. Our Pizza oven has certain downtimes characterized by a *startMaintenance* event A and an *endMaintenance* event C. A request to use the resource to bake a Pizza is represented by the occurrence of event B. This request is automatically create when a Pizza order arrives. Now, a complex event defines that event B should occur before the maintenance process starts, i.e. before event A and C.

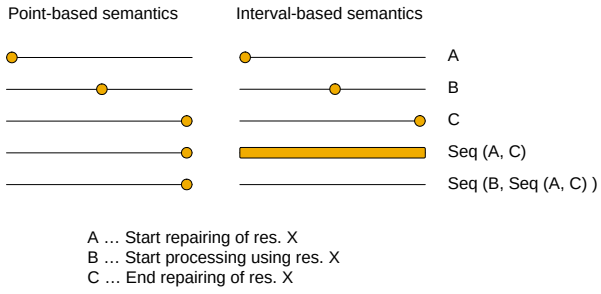

Figure 1: Different time representations.

Now, consider the occurrence of event B during the maintenance process. Using point-based semantics, the complex event $BEFORE(A, C)$ would get the timestamp of the last contributing event, i.e. event C.

The occurrence of event B would then be detected before the complex event $BEFORE(A, C)$, since the timestamp of B is smaller than the one of C. This means that the complex event defined by $BEFORE(B, BEFORE(A, C))$ would be detected using point-based semantics. This would result in the baking the Pizza although the oven is currently maintained. Obviously, this is not the semantic that a user intends when writing such a rule.

In contrast interval-based time representation considers the duration of the complex event $BEFORE(A, C)$. Thus, the complex event $BEFORE(B, BEFORE(A, C))$ would not be detected in our example, since B does not occur before the complex event resulting from $BEFORE(A, C)$. This corresponds to the detection mechanism that one expects when defining such a rule.

To avoid this unintended interpretation, we consider time as an interval consisting of a start and end point for each event. This notion follows [4] and [7]. This means, complex events always have a duration, i.e. $t_s < t_e$.

2.4 The Rete Algorithm

The Rete algorithm [6] is a pattern matching algorithm traditionally used for production-based logical reasoning systems. Its aim is to match a set of facts against a set of inference rules (productions).

Facts reside in the *working memory* and are n-tuples containing any number of data items. They represent information on something that is the case in the world. Facts are valid until they turn out to be false and are changed or retracted from the working memory.

A *rule* contains a premise stating conditions to be met by fact data items and a set of actions to be triggered if the premise holds.

The algorithm creates an acyclic network of the rule premises, the so-called Rete network. Figure 2 shows an example for a network for two rules with a root node (a Rete tree). The *Rete network*, starts with a root node which is split into the *type nodes* which distinguish between different facts. Then, an *alpha node network* is typically followed by a *beta-node network*. Whenever the working memory is changed, i.e. facts are "asserted", "retracted" or "updated', a *working memory element* (WME) is created for the changed fact and then propagated in a forward-chaining fashion through the network nodes from the root to the leaf nodes. Thereby, alpha-nodes perform simple conditional tests, i.e. they act as a filter by passing only the matching WMEs to the next node. At the end of the alpha node network, the resulting WMEs matching all previous nodes are stored in the alpha memory.

Beta-nodes perform joins by combining different WMEs, typically WME lists (from now on called *tuples*) coming from a beta memory with individual WMEs from an alpha memory. A new WME in the input alpha memory leads to a right activation on the beta node. Then, the new WME is

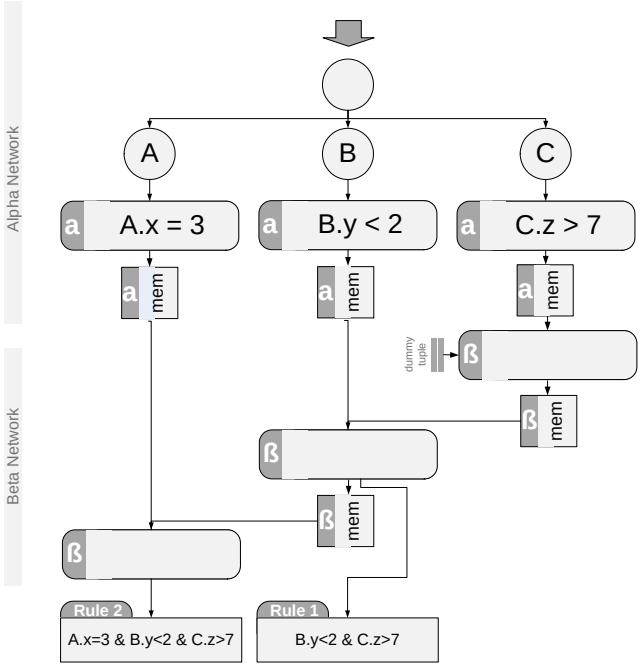

Figure 2: Example Rete network for two rules.

compared to specific WMEs of each tuple of the beta input memory. The specific WMEs to be used are specified in the join criteria. When a new tuple is added to the input beta memory, a left activation of the beta-node takes place, specific values of a predefined set of the new tuples are compared to particular values of each WME in the alpha memory. Upon an occurring match, a new tuple representing the match is added to the beta memory of the beta-node to be passed to subsequent beta nodes or to a terminal node (action trigger).

In other words, beta nodes store partial matches of rules. When a tuple has reached the end of the beta node branch, it is passed to the terminal node. It represents a complete match of the facts contained in the tuple. The terminal node activates a rule instance on the *agenda* which is responsible for execution of the resulting actions - depending on a conflict resolution strategy.

Storing the partial matches avoids re-evaluation of the complete premise on changes of the working memory. Thus, the detection time is decreased, since only changed facts need to be re-evaluated. In addition, in order to avoid redundancy and thus to save memory, nodes are shared in case rule constraints occur in multiple rules. However, this also results in higher memory consumption compared to a complete re-evaluation.

3. EVENT DETECTION

The detection of complex events is illustrated in Figure 3. Basically, its task is to detect complex events, e.g. the complex events X, Y, Z, from a stream of single input events (instantaneous or complex themselves), e.g. the events A, B, C.

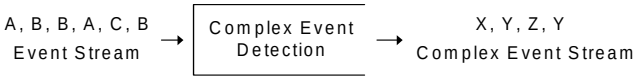

Figure 3: Complex event detection.

We assume a loosely-coupled system with a global clock. Our proposed system architecture is illustrated in Figure 4. Different event consumers transmit asynchronously their events to a central publish/subscribe system. The CEP system subscribes to all events which can lead to complex events. Therefore, the publish/subscribe system forwards these events to the CEP system. There, the arriving events enter an event queue and are then processed by the Rete Engine. The Rete Engine detects the complex events which are defined in a rule set stored in the *Rule* component. On detection of an event, the creation of the resulting complex event is added to an agenda, which has a scheduler to start the different event creations. On creation of a complex event, the event is send to the publish/subscribe system which distributes it to its subscribers. Finally, the configuration of the CEP system is possible using a separate tool, the *CEP config tool*.

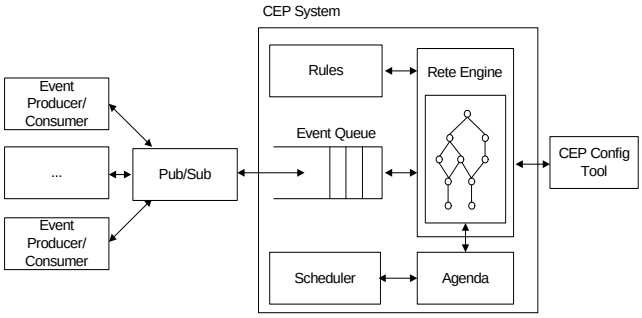

Figure 4: System architecture.

Thus, the occurrence of a complex event can be viewed as the occurrence of a combination of instantaneous events across distributed systems. Lamport's happened-before relation [9] holds. As stated in [17], where the NEXT operator for complex events was analyzed, Lamport's happened-before relation does not always hold. However, techniques to deal with the related issues are known, for instance [14] is using heartbeats to overcome them.

4. RELATIVE TEMPORAL CONSTRAINTS

In many cases, relative temporal relationships determine whether events are of relevance for a complex event. For instance, an complex event *MachineAlert* may be created only if a *MachineOverheated* event occurs 5 minutes before a *MachineFailed* event.

The Rete algorithm does not support the detection of temporal relationships per default. Consequently, we extended the algorithm with support for the thirteen operators defined by Allen [1] for time intervals. After describing these operators shortly, we will state how we extended the operators to also account for quantitative constraints. Finally, we will describe how beta nodes can be used to check for these constraints.

4.1 Allen's Operators Extended

Allen defined thirteen possible operators to describe the relationships between two time intervals. These relations are illustrated in Figure 5. The left side states the relation (with i meaning inverse), then follows the illustration of the relation and its interpretation on the right side.

Allen's operators can be used to describe the relation of two events (instantaneous or complex). The operators allow for the desired expressiveness when defining qualitative constraints between overlapping intervals such as the timestamps of complex events.

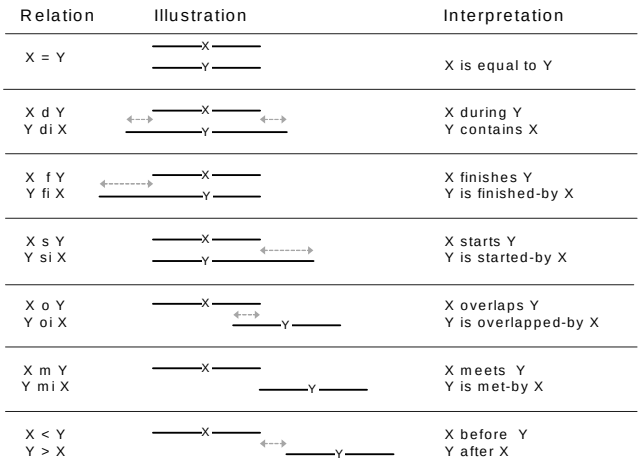

Figure 5: Possible relations between two intervals.

In contrast, point-based semantics [16] support only three qualitative constraints between two objects, namely *before, after* and *equals*. Nevertheless, point-based semantics have their strengths with quantitative constraints such as absolute time points or a duration between time points. Quantitative constraints are hard to express using interval-based semantics.

In order to combine the advantages of both approaches, we adopt the thirteen relations defined by Allen [1] and enhance them by allowing an operator-specific variable number of parameters denoting quantitative time constraints. Our method is similar to Meiri [12] who tried to unify both time-point and interval semantics in his Qualitative Algebra. The arrows in Figure 5 show the possible quantitative constraints which can now be supplied. Allen's operators do assume a predefined distance between the start and end points of the two intervals, but just assume it to be either equal or greater than zero. We extend this definition to, aditionally, allow the definition of exact values or a value range for the distance between the start and end time points of two time intervals.

For instance for *X DURING Y* four parameter settings are possible:

X DURING Y No parameter (default behavior) means the distance between the start points or the end points of X and Y defined is greater than zero, undefined

X DURING (a) Y A parameter a states an exact temporal distance between the start of X and the start of Y, e.g. *X DURING(5) Y* means X starts 5 time units after Y.

X DURING (a, b) Y parameter a is given and an additional parameter b states the distance between the end of X and the end of Y, e.g. *X DURING(5, 1) Y* means X starts 5 time units after Y and ends 1 time unit before Y.

X DURING (a1, a2, b1, b2) Y parameters $a1, a2$ and $b1, b2$ are given which state that the distance of the start of X and the start of Y should be between $a1$ and $a2$ and the distance of the end points of X and Y should be between $b1$ and $b2$, e.g. *X DURING(1, 3, 2, 5) Y* means X starts between 1 and 3 seconds after Y and ends between 2 and 5 time units before Y.

Thereby, permissive operators, such as *DURING* can be used to define restrictive constraints.

Furthermore, we allow for the definition of tolerance limits to make restrictive operators more permissive. In reality, having an *EQUAL* operator checking for the cooccurrence of events at exactly the same time is impractical, since often small differences in occurrence exist. Therefore, the user can specify what she sees as equal. *X EQUALS(2) Y* would mean that the an event X would still be considered equal to another event Y even if it occurs two time units before Y.

It needs to be noted that this freedom of expression comes with more responsibility for the user who defines the complex events. When using the parameters, the operators are no longer exclusive. For instance, if event X occurs 2 time units before Y then, *X EQUALS(2) Y* as well as *X BEFORE(2) Y* would hold. However, by omitting parameters, we obtain the original exclusive relations by Allen expressing only qualitative constraints.

4.2 Realization in Rete

Temporal relationsships can be detected in Rete by explicitly stating the conditions in the rules. For instance, the following rule can be used to describe a complex event Z which is created when an event X occurs before an event Y.

```
IF
  X.startTimestamp < Y.startTimestamp
THEN
  create Z.
```

However, this soon becomes tedious when different relationships need to be checked along with other constraints. Therefore operators realizing the check are more suitable.

The relative temporal constraints can be realized in Rete by an extension of the beta-nodes. The behaviour of a JOIN beta-node is as follows.

Left activation If a new tuple of WMEs occurs at the left input memory, all WMEs of the right input memory

are checked with the WMEs of the tuple to find out if their combination fullfills the join condition.

Right activation If a new WME occurs at the right input memory, all WMEs in all tuples of the left input memory are checked with the new WME to find out if their combination fullfills the join condition.

This behaviour can be used to check for the relative temporal constraints as follows.

Left activation If a new tuple of WMEs occurs at the left input memory, all WMEs of the right input memory are checked with the WMEs of the tuple to find out if their combination fullfills the relative temporal constraint.

Right activation If a new WME occurs at the right input memory, all WMEs in all tuples of the left input memory are checked with the new WME to find out if their combination fullfills the relative temporal constraint.

We realized Allen's operators and different parameter combinations in beta nodes.

Therefore, these operators can now be used in complex event definitions. The rule from above can be expressed as:

```
IF
  X BEFORE Y
THEN
  create Z.
```

5. GARBAGE COLLECTION IN RETE

The traditional Rete algorithm stores all WMEs in the beta-memories forever to allow for their later matching. However, this behaviour is not desired for event processing in Rete.

In most cases, an event is only of interest during a certain time frame, e.g. until a new event of the same type was created. Hence, during pattern matching, the event only needs to be considered for this period of interest. After that, it can be discarded from the working memory.

Furthermore, an event is able to fullfill its temporal constraints only for a limited time. Then it can no longer contribute to complex events and therefore can be discarded.

Not discarding such events would result in a steady growing memory consumption relative to the incoming events.

Consequently, garbage collection mechanisms is a solution to discard events after they are no longer of interest or cannot contribute to complex events. The Rete algorithm lacks garbage collection completely. Therefore, we propose a garbage collection mechanism for the Rete algorithm in the following.

The time that an event must at least be stored in a beta-memory is from now on referred to as the *event's lifetime*. It determines when an event can be discarded. In the following, we present different means of determining an event's lifetime. Then we continue with presenting an algorithm which allows the beta-memories to discard their stored WMEs or tuples as soon as the WMEs can no longer fullfill their temporal constraints.

5.1 Determining an Event's Lifetime

Garbage collection is supposed to retract only events which are no longer used by the system, i.e. whose lifetime has passed. This ensures the correctness of the garbage collection. Furthermore it guarantees that the system's behaviour is not altered by incorrectly retracting events from the system which can still satisfy matches in the Rete network. Different means exist to determine an event's lifetime and their choice depends on the application's needs. The lifetime can be a fixed time, be based on the desired event interpretation or be calculated, e.g. based on the event's temporal relationships.

Maximum event lifetime The maximum lifetime of an event can be defined by the user to tell the system when an event can be discarded. The algorithm described in 5.2 can then be used to delete events after their lifetime passed. The definition of a maximum event lifetime is pursued by ILOG JRules [2] where a system-wide event lifetime is specified by the user. This method can also be used, if an event has no temporal relationships to other events.

Time-based windows The lifetime of an event can be described individually for rule conditions by using window operators which determine for how long events are of interest. The window size determines the event lifetime. After this time has passed, an event can be discarded. Using windows is similar to using a maximum event lifetime but more fine-grainded working on a per rule level not system-wide. It can also be realized using the algorithm in 5.2.

Consumption modes Different applications have variable demands on how events are considered for event detection, e.g. if an event can be matched multiple times or only once. These modes of event combinations and usage characterise different consumption modes [19]. For instance, when considering events coming from sensor readings only the newest event from one reader may be of interest, since the others just represent old values. This so-called *recent consumption mode* can be used for garbage collection. Then, existing WMEs/-tuples are deleted from the beta-memories of the Rete network whenever a new WME/tuple arrives at them.

Lifetime calculation Finally, the life-time of events can be calculated based on the temporal constraints given in the complex event definitions. For instance, if event B arrives and a rule states that event B should occur two minutes before event A, then B can be discarded after two minutes.

Teodosiu and Pollak [15] present a method to calculate the lifetime of events by calculating the closure of the dependency matrix containing temporal distances

between all events. We use this as a basis for the calculation of the lifetime of tuples/WME arriving at the beta-memories in the Rete tree.

5.2 Incremental Garbage Collection

The basic idea of our garbage collection algorithm is to calculate the lifetime of an event as the temporal distance of an event based on its relationsship to other events. This distance states how long an event can contribute to complex events, i.e. how long it can fullfill its temporal constraints. The lifetime of a WME/tuple is calculated for each left/right input memory separately depending on the available WMEs and their temporal relationships. We use an incremental garbage collection mechanism which uses a timer-based approach to discard each WME/tuple individually after its corresponding lifetime passed.

Next, we describe the initial lifetime calculation which form the basis of our garbage collection mechanism. We continue with a description of the timer-based deletion of WME/tuple and conclude a consideration of the effects of node sharing and node independency.

5.2.1 Initial lifetime calculations

An event can be considered unnecessary, if it can no longer contribute to the detection of complex events. Then it can be discarded. An event can no longer lead to complex events, if it can no longer fulfill its temporal constraints. For instance, suppose an event A should occur 2 time units before an event B to create a complex event X. Now, if event B occurs, but no event A occured before, then event B can be discarded straight away, since it can never lead to the detection of event X.

The constraints an event has to fulfill depend on the complex event definitions. Consequently, the relations of events can once be calculated when a new complex event definition (a rule) is added to the system or when a complex event definition is updated. The relations can be represented in a matrix M_{ij} containing the temporal distances between two events.

Initially, the distance matrix is filled with the direct distances between the single events of the complex event definition and the other distances are set to infinite. Then the closure is calculated to determine the complete distance matrix. Details of the calculation can be found in [15]. The distance matrix is calculated for every complex event definition separately and updated or deleted whenever the complex event is changed or deleted.

The distance matrix can be used to determine the lifetime for each event depending on the existance of other events it is related to. In the Rete network, the aim is to discard the WMEs or tuples of the beta input memories as soon as it is noted that they can no longer fullfill their constraints. Thereby, the beta nodes represent certain system states, i.e. depending on their position in the Rete tree, the tuples of the left input memory contain more or less WMEs.

It can be calculated for each left and right input memory of each beta node separately how long a WME/tuple should be kept in it depending on which WME/tuple it combines with. The event lifetime for the left and the right input memory of a node is again constant as long as the complex event (rule) does not change that the node belongs to.

In the calculation of the lifetime, it has to be distinguished between right and left activation of the node.

On right activation, a WME arrives. The type of WME can be directly identified by checking the constraint of the preceding alpha or type node.

In contrast, on left activation, a WME tuple arrives. There, the constraints of all preceding beta-nodes have to be taken into account to find out all single types of WMEs which form part of the tuple. The tuple/WME is then used to find all corresponding entries in the distance matrix. Then the intersection of the existing distances is calculated. It represents the maximum time that an event can fullfill all its temporal constraints. The calculated WME/tuple lifetime is calculated once and then stored in the temporal beta-node.

5.2.2 Runtime garbage collection

We use incremental garbage collection to discard each WME/-tuple of a node's right and left input memory separately. During event detection, i.e. during runtime of the CEP system, a timer is created for each newly arriving tuple/WME to automatically delete the WME/tuple after its calculated lifetime passed. ra The garage collection mechanism is illustrated in Figure 6 for a sample beta node. It can be divided into six separate steps.

1. The beta node has lifetimes for the WME/tuples arriving at its right/left input memory stored. These lifetimes were calculated as described above.

   a) Whenever a new tuple arrives at the left input memory of the beta node, the node informs the working memory of this arrival and its corresponding life-time (2 minutes in the example in Figure 6).

   b) Whenever a new WME arrives at the right input memory of the beta node, the node informs the working memory of the new WME and its corresponding life-time (3 minutes in the example in Figure 6).

2. The working memory contains the originals of all WMEs. The Rete network only propagates copies of these WMEs. To keep account of the number of copies, the WME keeps a reference counter for the WMEs. When the working memory is informed of a new WME/tuple arrival at a right/left input memory, the working memory increases its reference counter for the single WME or all WMEs in the obtained tuple. Then, the working memory informs the garbage collector to initiate the garbage collection for this tuple/WME after the corresponding lifetime passed.

3. The garbage collector creates a timer-request for the WME/tuple at the callback service. This request contains the lifetime and a reference to the corresponding WME/tuple.

4. The timer-service manages a pool of timer threads. An idle thread is used to setup the timer with the defined lifetime.

5. After the time passed, the deletion process is initiated for the WME/tuple. A task is created to perform the deletion directly in the beta node using the tuple/WME reference.

6. Finally, the working memory is informed by the beta node, to decrease the reference counter for the corresponding WME or all WMEs in the tuple. If the reference counter of a WME is zero, i.e. no copies of it exist in the Rete tree, the WME can be deleted from the working memory.

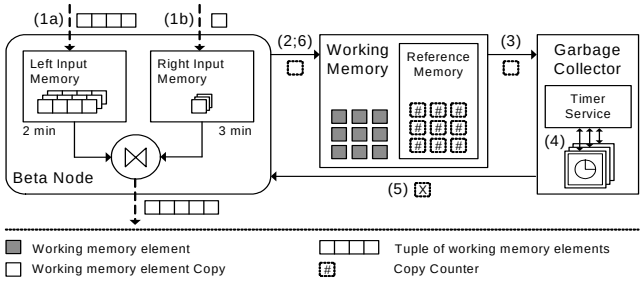

Figure 6: Garbage Collection mechanism.

Rule-based production systems are used in scenarios where already little downtime or unresponsiveness of the system can have a huge impact on the business. That leads to the fact that halting the system to do garbage collection tasks is in-acceptable. Instead the garbage collector needs to run concurrent to the system and compute its tasks. The production system needs to stay responsive even when the garbage collection retracts objects from the Rete network. The different timer threads allow a parallel execution of the garbage collection. Thus, it is not influencing the detection mechanism.

The integration of garbage collection into Rete is an extension which is supposed to help limit memory consumption. But instead of altering the algorithm it should stay unmodified. Consequently, the influence of the extension on the existing algorithm, the temporal-beta nodes and the working memory is kept minimal by having an independent garbage collector. The garbage collection is independent of the workings of the Rete algorithm. Thus, it is no problem to combine facts which do not need the garbage collection with events which are discarded when no longer needed.

5.2.3 Node sharing

Node sharing of beta-nodes implies that the condition of the node occurs in several rules, thus having different temporal constraints. A simple way to deal with this problem is taking the maximum of the calculated lifetimes resulting from the different rules the node belongs to. Thus, the event is not discarded as long as it can still be used.

5.2.4 Dealing with node independency

A disadvantage of the proposed garbage collection mechanism is the independency of nodes. Consider a Rete tree without node sharing. It is possible, that a tuple may be discarded from a temporal-node in the Rete tree, but the involved WMEs of the tuple and the tuples in the beta-nodes preceding the temporal-node may not be deleted, since no temporal relationship exists between them. Nevertheless, they could never lead to a rule activation and thus an event detection. One solution for this problem is to propagate the deletion upwards in the tree, deleting all preceding WMEs contributing to the tuple. However, this is not possible for Rete trees with node sharing, since the WMEs/tuples may be used further contributing to another rule.

6. IMPLEMENTATION

We have extended the JBoss Drools [8] rule engine with our proposed concepts for relative temporal constraints and event support. We are currently adding the support to process data streams and time-based windows.

A grammar extension provides the means to specify that a given object type should be handled as an event instead of a regular fact. This is achieved by using a special declaration statement. Thus, facts can be distinguished from events and temporal reasoning as well as specific optimization techniques (e.g. event garbage collection) be limited to events.

To address multiple usage scenarios, four different timestamp assignment strategies were identified. They all provide a common and general framework to determine and assign a timestamp and a duration for an event when it is asserted into the working memory. Interval-based time semantics are used for the timestamps, i.e. a duration can be specified for an event. The discussion of these strategies is out of the scope of this paper.

Further, Drools was extended with a framework for pluggable operators to ease the addition of operators. Using this framework, all the relative temporal constraints proposed in Section 4.1 were implemented and are available now for event correlation. The operators can have at most four optional parameters. Negative time limits can be given as parameters which leads to the possibility to use one temporal operator to express others, e.g. $A\ after(-4s, 4s)\ B \equiv A\ before(4s, 0s)\ B\ \vee\ A\ after(0s, 4s)\ B$.

The current version of the Drools extension is online available at [13].

7. RELATED WORK

This section introduces related work in the area of temporal support for the Rete algorithm.

The Padres [10] and the ILOG JRules [2] event processing systems are based on the Rete algorithm. In both cases, the Rete algorithm is extended to incorporate clocks. Timestamps are used and *before* and *after* predicates are introduced. However, the applied time-point semantics causes misinterpretations which we overcome using interval time semantics. Furthermore, the realization of time-based windows is not addressed.

In [11], a traditional production system based on the Rete algorithm is extended with temporal reasoning by storing past and developing events in a temporal database, a so-called time map. An interval time representation is used. The system supports detection of events occurring *during,*

before or *after* other events. It is further possible to model uncertain relationships. However, the semantics of the operators as well as the conceptual details remain unclear. It is not stated whether the start or the end time-point of the interval are used for the *before* and *after* operators.

As mentioned before, Teodosiu and Pollak [15] proposed a method for garbage collection in Rete. We calculate our event lifetimes similar to them. However, in contrast to us, [15] created a separate Rete network for each rule which is not how the common Rete implementations work and which does not facilitate the advantages of node sharing of Rete.

8. CONCLUSION

We have developed a concept to enable the Rete algorithm to detect relative temporal constraints between events. For this purpose, we have extended Allen's thirteen temporal operators [1] with quantitative constraints and introduced a method for garbage collection of events that can no longer fulfill their temporal constraints. Finally, we have described how the concepts were implemented in JBoss Drools [8] and we demonstrated that the garbage collection results in a reduction of used memory while only deleting unused events.

In conjunction with the original features of the Rete algorithm, the proposed concepts offer the ability to detect temporal relationships between events in Rete. Thus, they facilitate more expressiveness in complex event processing using Rete.

9. ACKNOWLEDGEMENTS

We wish to acknowledge Michael Ameling and Maik Thiele for helpful comments on earlier drafts of this paper. Furthermore we thank the Drools developer team, especially Edson Tirelli, for the interesting discussions and their contribution to the presented concepts and implementation.

10. REFERENCES

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[2] B. Berstel. Extending the RETE Algorithm for Event Management. In *TIME '02: Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME'02)*, page 49, Washington, DC, USA, 2002. IEEE Computer Society.

[3] M. H. Bohlen, R. Busatto, and C. S. Jensen. Point-versus interval-based temporal data models. In *ICDE*, pages 192–200, 1998.

[4] F. Bry and M. Eckert. Temporal order optimizations of incremental joins for composite event detection. In *Proceedings of Inaugural Int. Conference on Distributed Event-Based Systems, Toronto, Canada (20th–22nd June 2007)*. ACM, 2007.

[5] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[6] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[7] A. Galton and J. Augusto. Two approaches to event definition. In *Lecture Notes In Computer Science. Proceedings of the 13th International Conference on Database and Expert Systems Applications*, volume 2453, pages 547 – 556, 2002.

[8] JBoss. *JBoss Rules*, 2007. http://labs.jboss.com/drools/.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[10] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, pages 249–269, 2005.

[11] M. A. Maloof and K. Kochut. Modifying Rete to Reason Temporally. In *ICTAI*, pages 472–473, 1993.

[12] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. In T. Dean and K. McKeown, editors, *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 260–267, Menlo Park, California, 1991. AAAI Press.

[13] Red Hat, Inc. Drools development branch for temporal reasoning. Online. http://anonsvn.labs.jboss.com/labs/jbossrules/branches/temporal_rete/.

[14] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274, New York, NY, USA, 2004. ACM Press.

[15] D. Teodosiu and G. Pollak. Discarding unused temporal information in a production system. In *Proc. of the ISMM International Conference on Information and Knowledge Management CIKM-92*, pages 177–184, Baltimore, MD, 1992.

[16] M. Vilain, H. Kautz, and P. van Beek. *Constraint propagation algorithms for temporal reasoning: a revised report*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[17] W. M. White, M. Riedewald, J. Gehrke, and A. J. Demers. What is "next" in event processing? In *PODS*, pages 263–272. ACM, 2007.

[18] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM Press.

[19] E. Yoneki and J. Bacon. Unified semantics for event correlation over time and space in hybrid network environments. In *OTM Conferences (1)*, pages 366–384, 2005.