Time to the Rescue - Supporting Temporal Reasoning in the Rete Algorithm for Complex Event Processing

Karen Walzer, Matthias Groch, and Tino Breddin

SAP Research CEC Dresden, Germany
`karen.walzer@sap.com`

Abstract. Complex event processing is an important technology with possible application in supply chain management and business activity monitoring. Its basis is the identification of event patterns within multiple occurring events having logical, causal or temporal relationships.

The Rete algorithm is commonly used in rule-based systems to trigger certain actions if a corresponding rule holds. The algorithm's good performance for a high number of rules makes it ideally suited for complex event detection. However, the traditional Rete algorithm does not support aggregation of values in time-based windows although this is a common requirement in complex event processing for business applications.

We propose an extension of the Rete algorithm to support temporal reasoning, namely the detection of time-based windows by adding a time-enabled beta-node to restrict event detection to a certain time-frame.

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 this purpose, for instance [1] uses a finite state automaton for event detection and Li and Jacobsen [2] showed the efficiency of the Rete algorithm to match a high number of rules and thus complex events.

Traditionally, the Rete algorithm [3] is used 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's linear complexity makes it capable of dealing with a high number of complex event patterns.

Business applications often require the definition of complex events which consider temporal relationships among single events. Then, time-based windows are utilized to ensure that event detection takes place only for the time of an events' relevance. This is achieved by restricting processing to events matching a pre-defined time-frame, e.g. occurring within 1 hour or between 1 and 2 pm. It is often desirable to aggregate events, for instance to calculate the average of a certain event data item over time, such as a temperature sensor value. Time-based windows allow for this event aggregation over a pre-defined time. However, the traditional Rete algorithm does not support such 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 detection of relative relationships have been suggested, e.g. in [4,5]. However, sophisticated temporal operators such as sliding windows were not regarded so far.

In this paper, we present an extension of the Rete algorithm to support temporal reasoning using time-based windows and thus to enable complex event processing. Our method to support temporal constraints in the Rete algorithm significantly extends existing work [2,6] for the detection of event patterns using time-based sliding windows. It uses an incremental window approach to continuously update the current window and its aggregation values.

The remainder of this paper is organized as follows. We present an overview of related work in Section 2. Then, we continue with a definition of terms and a brief description the Rete algorithm in Section 3. Section 4 presents an overview of our approach for time-based sliding windows in Rete and finally Section 5 concludes the paper.

2 Related Work

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

The Padres [2] and the ILOG JRules [4] 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 realization of time-based windows is not addressed.

Gordin and Pasik [6] describe a method to support set-oriented methods for forward chaining rule-based systems. The presented ideas are extended in our work to support event aggregation in Rete.

In [7], 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. Time-based windows are also not considered.

Teodosiu and Pollak [8] present a method to remove obsolete temporal facts where the used Rete network is extended with timers in order to discard events after a specified time interval elapsed. Their concept can be used to discard events after they are no longer part of a sliding time-based window.

3 Temporal Support in Rete

In the following, we present our notion of instantaneous and complex events and introduce the Rete algorithm.

3.1 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. We consider the timestamp of a complex event as an interval consisting of a start and end point for each event to avoid the unintended interpretation occuring for time-point semantics [9,10]. This notion follows [11] and [12]. It allows the usage of Allen's [13] thirteen temporal operators to determine the relationship between two events having interval timestamps. Consequently, complex events always have a duration, i.e. $t_s < t_e$.

3.2 The Rete Algorithm

The Rete algorithm [3] 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

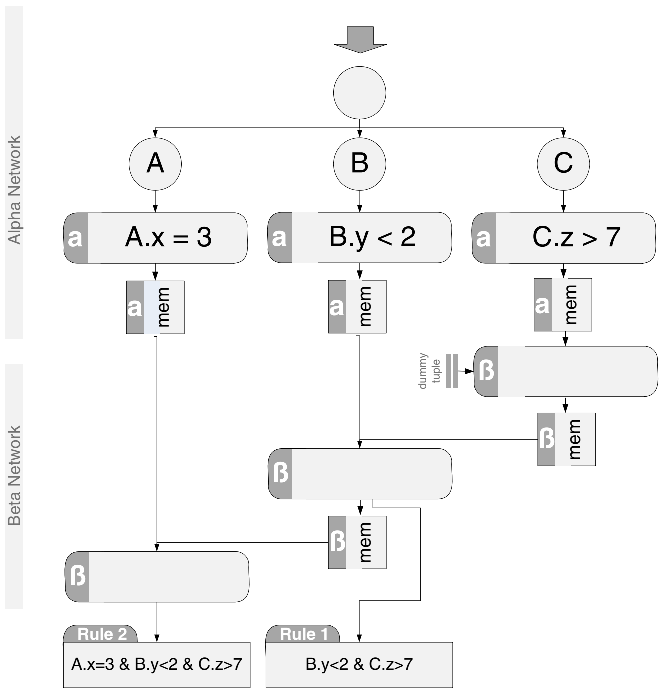

Fig. 1. Example Rete network for two rules

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 1 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 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 to avoid re-evaluation. 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 and results in the execution of the corresponding actions.

4 Proposed Solution

In the following, we will describe the assumptions underlying our system and its semantics. Then, we will introduce a possible approach to sliding window definition in Rete.

4.1 Preliminaries

We assume a loosely-coupled system with a global clock. The occurrence of a complex event can be viewed as the occurrence of a combination of instantaneous events across distributed systems. In our system, the instantaneous events are transmitted asynchronously to the central CEP engine which is based on the Rete algorithm. Lamport's happened-before relation [14] holds. As stated in [15], Lamport's happened-before relation does not always hold. However, techniques to deal with the related issues are known, for instance [16] is using heartbeats to overcome them.

4.2 Event Aggregation Using Time-Based Windows

A window is a limitation of the view on data, i.e. instead of considering all input elements, only an extract of them is considered. Two common types of windows are distinguished: *tuple-based* and *time-based windows*, cf. to [17,18]. Tuple-based windows limit the view to the last n input items whereas time-based windows provide only elements which arrived within a fixed time span. We focus on time-based windows with relative time-constraints, e.g. 5 minutes. We do not support windows with given absolute time constraints, e.g. a window to occur between 1 and 2 pm.

We define a time-based window as the set of events whose end time-point is in a given window interval, i.e. $\omega(t_{s_w}, t_{e_w}) := \{x \in \mathbb{E} | t_{e_x} \geq t_{s_w} \wedge t_{e_x} \leq t_{e_w}\}$ where t_{s_w} denotes the start and t_{e_w} the end time of the window. The *size of a window* $d_w = d(t_{e_w}, t_{s_w})$ states its duration in time units, e.g. 5 minutes. It is defined by the user for each window. Using this definition means that an event can start before the window in which it is considered.

Assuming, the current time is t_0, then the boundaries of the *last finished window* can be calculated as follows. The *end time* of it is the current time, i.e. $t_{e_w} = t_0$. The *start time* is the difference between its end time and the window size, i.e. $t_{s_w} = d(t_{s_w}, t_{e_w})$. The data items with end timestamps between the start and the end time are part of the window. Using this calculation, the window is sliding, i.e. its boundaries are constantly shifted with ongoing time. It is referred to as a *sliding time-based window*.

The Rete algorithm performs eager evaluation, i.e. the node network's inner state is updated with every inserted, modified or retracted tuple or fact, respectively. Hence, we currently consider only sliding time-based windows without a slide parameter. For window evaluation, we pursue an incremental approach, where the current window is continuously updated depending on incoming events and passing time. The resulting changes, i.e. the addition or deletion of window elements, are propagated to successive nodes in the Rete tree. This approach is suited for eager evaluation.

We extend the existing beta nodes with features for window evaluation and event aggregation. Using a beta-node for this purpose allows for the evaluation of windows for instantaneous events as well as for tuples. From now on, we will refer to these extended nodes as *time-driven aggregation node*s (TDA). Besides the join-function of beta nodes, TDAs are capable of keeping track of the current state of a time window as well as performing on-the-fly aggregation functions to the elements of the window. A TDA node is used just as a normal beta-node in the Rete network. Depending on the rule definition, either its aggregation function, the window function or both are used. For the last case, a TDA node first performs the join of the incoming tuples and WMEs, then it checks for the window constraints and finally performs the aggregation function.

The events that should be considered for the window and/or for the aggregation need to be specified explictly, e.g. consider the rule $IF(A.x = 3 \wedge B.y < 2 \wedge window[5min, AVG(C.z) > 40](C.z > 7))THENX$. It describes the conjunction of particular events A and B with an average value of an attribute of event C. The average is calculated time-frames of for 5 minutes using all attributes with $C.z > 7$. The resulting Rete tree would look like in Figure 1, except that the first beta-node (rightmost) would be a TDA node. This means, the TDA node is inserted right below the alpha node of event C and obtains the WMEs which have matched the filter criteria $C.z > 7$.

The process of the window evaluation will be described in the following. Our approach is inspired by the work of Gordin and Pasik [6] as well as Ghanem *et al.* [19].

Tuples (from the left) and WMEs (from the right) arriving at the TDA are filtered depending on whether their end timestamps are outside the window bounds or not. The stored window boundaries are adjusted continuously in the TDA node. Every time an event $x \in \mathbb{E}$ arrives at the node, the current window end time t_{e_w} is set to the arrival time of that event. The corresponding window start time can easily be determined by simply subtracting the window size from the window end time $t_{s_w} = d(d_w, t_{e_w})$. The arrived event is propagated to be considered in the aggregation function or in the child nodes of the TDA node. Furthermore, each new event is sent to a garbage collection (GC) thread which creates a callback entry in a queue for it in order to discard the event once it is out of the window bounds. The time after which the event can be discarded is the window size. After this time passed, the positive event is out of window-bounds and a negative event is send to the TDA node by the GC thread. Then, the TDA node sends a message that the corresponding event can be discarded to its child nodes. If the child nodes, e.g. other windows, do not need the event anymore, the WME reference counter for the event can be decreased by one. If no references exist any longer, i.e. the reference counter is zero, the WME can be deleted. In any case, the aggregation function is updated, when an event is outside a window.

The TDA node keeps track of the current state of the aggregation function. For instance, in case of the *average* function, it stores a double value representing the sum of the event attribute of interest and an integer denoting the number of events contributing to the aggregated value. With every arriving or expired event, the beta memory and the aggregation result are updated.

Whenever, an event is no longer part of any sliding windows, it can be discarded from the alpha/beta-memory of the TDA node and possibly also from the working memory. Consequently, the event can no longer fullfill its temporal constraints. The other nodes in the Rete network can be informed to determine if they can also discard the event. Garbage collection mechanisms based on this sliding window timeout, a default event lifetime or a calculated lifetime based on an event's relative relationships are outlined in [8, 5].

5 Conclusion

We have presented concepts of how the Rete algorithm can be extended with the detection of event patterns containing time-based sliding windows by the introduction of time-enabled beta-nodes.

We are currently evaluating the proposed concepts by extending the business rule management system JBoss Drools [20] with support for sliding windows. The current version of the Drools extension is online available at [21].

In conjunction with the detection of relative temporal constraints, event garbage collection [5] and the original features of the Rete algorithm, the proposed concepts form a good basis for temporal reasoning in Rete.

Acknowledgements. We wish to acknowledge Maik Thiele, Michael Ameling and Thomas Heinze 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 the ongoing implementation.

References

1. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M.: Cayuga: A general purpose event monitoring system. In: CIDR, pp. 412–422 (2007)
2. Li, G., Jacobsen, H.A.: Composite subscriptions in content-based publish/subscribe systems. In: Middleware, pp. 249–269 (2005)

3. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
4. Berstel, B.: Extending the RETE Algorithm for Event Management. In: TIME 2002: Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME 2002), Washington, DC, USA, vol. 49. IEEE Computer Society, Los Alamitos (2002)
5. Walzer, K., Breddin, T., Groch, M.: Relative Temporal Constraints in the Rete Algorithm for Complex Event Detection. In: DEBS 2008: 2nd International Conference on Distributed Event-Based Systems (to appear, 2008)
6. Gordin, D.N., Pasik, A.J.: Set-oriented constructs: from Rete rule bases to database systems. In: SIGMOD 1991: Proceedings of the 1991 ACM SIGMOD international conference on Management of data, pp. 60–67. ACM Press, New York (1991)
7. Maloof, M.A., Kochut, K.: Modifying Rete to Reason Temporally. In: ICTAI, pp. 472–473 (1993)
8. Teodosiu, D., Pollak, G.: Discarding unused temporal information in a production system. In: Proc.of the ISMM International Conference on Information and Knowledge Management CIKM 1992, Baltimore, MD, pp. 177–184 (1992)
9. Bohlen, M.H., Busatto, R., Jensen, C.S.: Point-versus interval-based temporal data models. In: ICDE, pp. 192–200 (1998)
10. Yoneki, E., Bacon, J.: Unified semantics for event correlation over time and space in hybrid network environments. In: OTM Conferences (1), pp. 366–384 (2005)
11. Bry, F., Eckert, M.: Temporal order optimizations of incremental joins for composite event detection. In: Proceedings of Inaugural Int. Conference on Distributed Event-Based Systems, Toronto, Canada, June 20–22, 2007. ACM, New York (2007)
12. Galton, A., Augusto, J.: Two approaches to event definition. In: Hameurlain, A., Cicchetti, R., Traunmüller, R. (eds.) DEXA 2002. LNCS, vol. 2453, pp. 547–556. Springer, Heidelberg (2002)
13. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843 (1983)
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
15. White, W.M., Riedewald, M., Gehrke, J., Demers, A.J.: What is "next" in event processing? In: PODS, pp. 263–272. ACM, New York (2007)
16. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: PODS 2004: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 263–274. ACM Press, New York (2004)
17. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (2006)
18. Ghanem, T.M., Aref, W.G., Elmagarmid, A.K.: Exploiting predicate-window semantics over data streams. SIGMOD Rec. 35(1), 3–8 (2006)
19. Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Incremental evaluation of sliding-window queries over data streams. IEEE Trans. Knowl. Data Eng. 19(1), 57–72 (2007)
20. JBoss: JBoss Rules (2007), `http://labs.jboss.com/drools/`
21. Red Hat, Inc.: Drools development branch for temporal reasoning, `http://anonsvn.labs.jboss.com/labs/jbossrules/branches/temporal_rete/`