Extending the RETE Algorithm for Event Management

Bruno Berstel
ILOG
9, rue de Verdun
94253 Gentilly Cedex, France
berstel@ilog.fr

Abstract

A growing number of industrial applications use rule-based programming. Frequently, the implementation of the inference engine embedded in these applications is based on the RETE algorithm. Some applications supervise a flow of events in which time, through the occurrence dates of the events, plays an important role. These applications need to be able to recognize patterns involving events. However the RETE algorithm does not provide support for the expression of time-sensitive patterns. This paper proposes an extension of RETE through the concepts of time-stamped events and temporal constraints between events. This allows applications to write rules that process both facts and events.

1. Introduction

Incremental pattern matching algorithms have been studied for some time now. The two most widely known are RETE [4] and TREAT [9]; the Gator algorithm [6] is derived from them. However, these algorithms do not specifically consider time and thus offer no time-related constructs.

On-line recognition of temporal patterns has been formalized under the term of *chronicle recognition* by Dousson [2, 5, 1]. Several chronicle recognition algorithms have been published [7, 2, 3]. Unlike the work described in this paper, these algorithms and their implementations do not aim at providing advanced pattern matching features.

In this paper we propose an extension of the RETE algorithm to incrementally recognize patterns including events with temporal constraints, while still benefiting of RETE's pattern matching efficiency.

In Section 2 we give an example of a rule involving facts and events. Section 3 then introduces the concepts we need to add to the standard RETE algorithm. Sections 4 and 5 present our extension of RETE, and illustrate it using the example rule. Finally Section 6 concludes and underlines the benefits of integrating event management in a rule engine.

2. Example

We start with an example of a rule matching both facts and events. The example models the following situation. A supervisor receives events from equipments that may be off-line, on-line, or active. The events are of three types: alarms, related to an equipment; confirmations of alarms, which mean that the reason that triggered the alarm still holds; and cancellations of alarms. The supervisor maintains internal representations of the monitored equipments and of the events, as objects (e.g. instances of Java classes). They are processed according to rules.

Our example is one of these processing rules. It is expressed below in the ILOG JRules language, a Java-like variant of the OPS5 language. The rule monitors a sequence of alarms occurring on an active equipment, made of an initial alarm followed within 5 clock ticks with a confirmation signal.

```
rule AlarmConfirmation {
  when {
    ?e: Equipment(state == ACTIVE);
    ?a: event Alarm(eqpt == ?e);
    ?c: event Confirmation(alarm == ?a;
                           ?this after[1,5] ?a);
  } then {
    assert new ConfirmedAlarm(?a,?c);
  }
};
```

The logic expressed in this rule is: for all facts of the `Equipment` class with a `state` field having the `ACTIVE` value; for all *events* of the `Alarm` class with an `eqpt` field matching an `Equipment` which satisfies the previous condition; for all *events* of the `Confirmation` class, related to an `Alarm` matched by the second condition, and occurring

Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME'02)
1530-1311/02 $17.00 © 2002 IEEE

between 1 and 5 clock ticks *after* this alarm; create an instance of the `ConfirmedAlarm` class and assert it as a new fact.

3. Introducing Time and Events

The example above uses some concepts which are new to non-temporal rule systems: time, events as opposed to facts, time constraints.

In order for our rule engine to support temporal reasoning, we equip it with a *clock*. The operations required on the engine clock are: a function returning the current time, and a function incrementing the current time by one tick. As the choice of these operations indicates, we choose a time model based on a discrete succession of instants [8].

In the regular RETE algorithm the facts bear no temporal information. In particular, our extension, in the absence of events, is strictly equivalent to classical RETE. Here we introduce the concept of *event*. An event can be stored in the engine memory just like a fact, and the engine maintains a time-stamp for each event. An event in our model has no duration (events that take time would be represented by a start event and an end event). In the simplest case, the time-stamp of an event is the value of the engine clock when the event is asserted.

Since there is no difference in structure between a fact and an event, what makes the engine distinguish between them is the way they are asserted. The facts are the objects asserted with the `assert` primitive; they hold true from the moment they are asserted, until the moment they are explicitely retracted, if any. On the opposite, the events are the objects asserted with the `assert-event` primitive; they hold from their occurrence dates, until they are explicitely or automatically retracted. The conditions in rules explicitly indicate whether they match facts or events.

In order to specify *temporal constraints* between events, we use `before` and `after` predicates. Temporal constraints can be combined together and with non-temporal tests, using disjunction, conjunction, and negation operators.

Of course, temporal constraints can only be specified between events. Note that there is always a temporal constraint between two events in a rule. If none is explicitely specified, it means that the events can occur in any order, which corresponds to a constraint with two infinite bounds.

4. Extending RETE for Incremental Temporal Pattern Matching

In a RETE network, the join nodes contain the tests from the rule conditions which control whether objects will be matched together. For instance, the RETE network representing the example rule includes a join node matching together `Alarm` and `Confirmation` events, and containing the `?c.alarm == ?a` and `?c after[1,5] ?a` tests.

When a new alarm is asserted, it is submitted to the join node, which evaluates its tests on the alarm against each of the already asserted confirmations. Symmetrically, when a new confirmation is asserted, the join node evaluates the tests on the confirmation against each of the already asserted alarms. To achieve this, each parent node maintains a storage of the objects it submits to the join node, so that the join node will be able to evaluate tests against them when it is submitted a new object by the other parent node.

When time is not involved, the parent nodes store *indefinitely* all the objects they submit to the join node. On the other hand, when a condition includes temporal constraints, this gives a limit in time to its satisfiability. This limit can be used to bound the time during which objects are stored in parent nodes of a join node.

In our example, the join node contains the constraint that "the confirmation must occur between 1 and 5 ticks after the alarm". Assume that an alarm occurred at date 10: after date 15, it will be impossible to match this alarm with any confirmation, because the temporal constraint will never be satisfied. This means that the parent node needs only store the alarm until date 15, and can then release it.

In our extension of the RETE algorithm, we implement this by adding a dialog between the join node and its parent nodes:

- When an event is submitted to a join node by a parent node, the join node computes the event's expiry date with respect to the temporal constraints it stores. This date is computed from the time-stamp of the event, and from the bounds in the temporal constraints held by the join node.
- If the expiry date of the event has not yet been reached, the parent node keeps it at least until this date, so that the join node may match it against new events that it may be submitted. On the contrary, if the event has expired, there is no need for the parent node to store it. In both cases, the join node informs its parent node of whether the event should be kept or not.
- In the case where the expiry date of the event is in the future, the join node posts a request to be notified at that date. When it is notified, the parent node will no longer need storing the event. The join node then informs its parent node, which can thus remove the event from its storage.

Note that the computation of the event expiry date, as well as the dialog between the join node and its parent nodes, always occur. They do not depend on whether the event could be matched by the join node or not. In our example, confirmations are never stored in the parent node, yet

they can match other facts and events. This is because the matching trials are performed on data previously asserted, whereas the expiry date computation addresses future assertions.

5. Matching Facts With Events

We stated in Section 3 that facts hold true from the moment they are asserted. This is applicable as far as the recognition of facts by conditions is concerned, but must be restricted when considering matching a fact with an event. The exact principle that we follow to trigger rules on a set of facts and events is:

> In a rule matching facts and events, all fact conditions must be satisfied as soon as the first event condition has been satisfied, and until the last one has been satisfied.

This principle expresses that from the viewpoint of a given event, only the facts that were asserted before the event hold true. It forbids that facts match events that were asserted beforehand.

The rule engine must enforce this principle when matching a fact with an event, as illustrated below. On the other hand the principle is crucial to the engine, as relieves it from having to keep events indefinitely in join node parents, only to be matched with future facts.

To illustrate that the principle is needed, let us consider the execution of our example on the following scenario. An alarm occurs (at date 30, say) on an equipment which state is 'on-line'. Because the equipment is not 'active', the join node containing the `eqpt == ?e` test will not consider it for a match with the alarm. Since the join node contains no temporal constraint, the alarm will not be stored in its parent node. As a result, if the equipment becomes active at date 32, it will not be matched with the alarm. This conforms to our principle, and is the expected behaviour of a RETE-based system.

Assume now that we extend our example with the following rule:

```
rule AlarmCancelled {
  when {
    ?a: event Alarm();
    ?c: event Cancellation(alarm == ?a;
                           ?this after[1,4] ?a);
  } then {
    retract ?a;
  }
};
```

The join node corresponding to this rule contains the `?c after[1,4] ?a` constraint. Thus the alarm that occurred at date 30 will be kept until date 34. When the equipment is actived at date 32, it becomes candidate to a match by the `AlarmConfirmation` rule with the (now stored) alarm. But the principle states that this match is forbidden, because the condition on the equipment was not satisfied at date 30, when the alarm satisfied the second condition. This is what we want, as it ensures that the semantics of a rule is not impacted by the presence or absence of another rule.

We implement the principle by ignoring facts in join nodes where the other parent node sends events.

6. Conclusion

In this paper we presented an extension of the RETE algorithm to integrate event management in a rule engine. This extension introduces the concepts of event, and of temporal constraint between events. It allows rule-based programs to recognize patterns involving time-independent facts and time-stamped events.

In addition, by managing the expiry dates of events, this extension allows the rule engine to automatically retract an event when it can be deduced from the temporal constrainsts expressed in the rules that the event can no longer be matched. This relieves the rule programmer from the burden of managing the event lifecycle.

The work described in this paper has been implemented in the ILOG JRules™ product. This product includes a rules engine whose implementation is based on the RETE algorithm, as well as advanced rule programming tools, such as business rule languages, a debugger, and an extensible rule management environment.

References

[1] M.-O. Cordier and C. Dousson. Alarm driven monitoring based on chronicles. In *SafeProcess*, 2000.

[2] C. Dousson. *Suivi d'évolutions et reconnaissance de chroniques*. PhD thesis, Université Paul Sabatier, Toulouse, 1984.

[3] D. Fontaine and N. Ramaux. An approach by graph for the recognition of temporal scenarios. *IEEE Transactions on System, Man and Cybernetics*, 1997.

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

[5] M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 1996.

[6] E. Hanson and M. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Technical Report TR93-036, Univ. of Florida, 1993.

[7] F. Lévy. Recognising scenarios: a study. In *Int'l Workshop on Principles of Diagnosis*, 1994.

[8] D. V. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155.

[9] D. Miranker. Treat: A better match algorithm for ai production systems. In *National Conf. on AI*, 1987.