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.

Category, Track, and Topics

Regular paper. Track 1: *Temporal Representation and Reasoning in AI*. Topics: temporal languages and architectures, reasoning about actions and change, time and nonmonotonism.

1 Introduction

Incremental pattern matching algorithms, and in particular the RETE algorithm, are extensively used in industrial applications implementing pattern matching problems. However, time is not specifically considered in these algorithms. As a consequence, applications which monitor a flow of time-stamped events cannot use the pattern matching algorithm for the recognition of time-sensitive patterns.

This paper presents an extension of the RETE algorithm that supports temporal constraints between time-stamped events, and incrementally matches the occurrences of patterns including events as well as regular facts.

In the next section we review some related works, both about incremental pattern matching algorithms and about recognition of temporal patterns. In Section 3 we introduce an example of a rule involving facts and events. Section 4 presents our extension of the RETE algorithm, and illustrates it using the example rule. Finally, Section 5 discusses the benefits of integrating event management in the rule engine, and Section 6 concludes and proposes future work directions.

2 Related Works

Incremental pattern matching algorithms have been studied for some time now. The two most widely known are RETE [7] and TREAT [18]; the Gator algorithm [12] is derived from them. As mentioned before, these algorithms do not specifically consider time and thus offer no time-related constructs. Several industrial

products implement variations of the RETE algorithm: ILOG JRules [13], Rete++ [11], OPSJ [20], or JESS [21] to mention a few of them.

On-line recognition of temporal patterns has been formalized under the term of *chronicle recognition* by Dousson [4, 9, 3]. Several chronicle recognition algorithms have been published [16, 4, 6]. Likewise, several chronicle recognition systems have been developed, such as IxTeT [10], two systems named CRS [8, 19], or FONSYNT [14].

Unlike the work described in this paper, these algorithms and their implementations do not aim at providing advanced pattern matching features. They concentrate on partial order handling, and aim primarily at identifying the sets of events which satisfy the temporal constraints. Only then do they address the pattern matching criteria, as far as such criteria can be formulated in these systems.

All these works either focus on the pattern matching problem without encompassing time, or they only reason on the event occurrence dates without providing more than very basic pattern matching features. Applications which want to both apply elaborate pattern matching operations, and detect the occurrence of temporal patterns, on complex objects and events, would demand the integration of event management into a commonly implemented pattern matching algorithm such as RETE.

A couple of start-up companies (Apama [1], SpiritSoft [22]) have recently announced inference engines with advanced features in temporal patterns recognition. Unfortunately the access to the algorithms used by the products of these companies is restricted by their commercial nature.

3 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). These are processed according to rules. One of them is given below (another will be given later). It is expressed 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);
  }
};
```

This rule reads as follows:

- 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` satisfying the previous condition;
- For all *events* of the `Confirmation` class, related to an `Alarm` matched by the second condition, and occurring between 1 and 5 clock ticks *after* this alarm;
- Create an instance of the `ConfirmedAlarm` class and assert it as a new fact.

This example introduces some new concepts, compared to constructs of non-temporal rule systems: event as opposed to fact, event condition versus fact condition, temporal constraint. It also presents a rule matching together time-stamped events, and facts which have no temporal reference. All this will be detailed in the next section.

Compared to chronicle recognition systems, the rule language illustrated above is less concise than Carle's CRS, which uses expressions close to regular expressions to describe sequences of events. Also, in contrast with Dousson's CRS, the event time-stamps are implicit[1]: temporal constraints are formulated as if the events themselves were compared. These choices aim at keeping in line with the style of the rest of the pattern matching language. Temporal constraints between the events are written as standard tests applied to the objects matched by the rule.

Yet this language gives the rule developer an appreciable power of expression. Partial order between two events, for instance, can be expressed simply by extending the range of a temporal condition, as in `?this after[-10,10] ?a`. Expressing partial order in regular rule systems is usually painful.

4 Incremental Algorithm for Temporal Pattern Matching

4.1 Summary of RETE

A rule engine implementing the RETE algorithm applies a set of rules to a set of facts. A rule is made of a collection of conditions (the example rule above has three conditions) associated with a sequence of actions to be applied to each collection of facts matching the rule conditions.

The rules are compiled into a graph, commonly known as the *RETE network*. The nodes of this network represent the tests expressed in the rule conditions. There are three types of nodes, corresponding to three types of tests. *Class nodes* filter facts according to their classes; *discrimination nodes* retain facts according to the values of their attributes; *join nodes* combine facts satisfying a given relation into tuples. In figure 1 on page 5 class nodes are not represented; the discrimination nodes representing the three conditions of the rule are respectively *left1*, *right2*, and *right3*; the *join2* and *join3* nodes are the join nodes of the last two conditions.

The rule engine only considers the objects stored in the *working memory* when looking for facts to match the rule conditions. Three operations are defined on the working memory: *assert* adds a fact to the working memory; *retract* removes a fact from the working memory; and *update* instructs the engine to reconsider the matching state of a fact.

When a fact is asserted into the engine working memory, it is submitted to the class nodes of the network. If the fact satisfies the tests held by these nodes, it is stored there and passed to the next level of nodes. This is repeated until either the fact fails on a test, or it waits for another object to match, or it exits the network in a tuple and a rule is fired.

At any time, the RETE network stores the facts and combinations of facts that match as much of the condition rules as possible. Each operation on the working memory incrementally updates the network state.

Using the example rule above, let us assume that one active `Equipment` and two `Alarm` on this equipment have been asserted. At this point, the RETE network is in the state described by figure 1, with the exception that node *right3* is still empty. An instance of the `Confirmation` class is created for the second alarm within 5 time units and asserted into the engine working memory:

- It is stored in the class node for the `Confirmation` class.
- Then it is passed to the *right3* discrimination node, which stores it.

[1] Although they are accessible using the `timeof` primitive.

- Then it is passed to the *join3* join node, which evaluates its tests on the `Confirmation` instance and on the pairs stored in the *left2* node. The tests succeed with the second pair, so the node creates a triple from the matching pair and instance.
- The actions of the rule are then executed on this triple.

4.2 Introducing Time and Events

In order for the 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.[2] When the clock time is actually incremented, is left to the application embedding the rule engine. Observe that the clock needs not be connected to a real-time clock.

In the regular RETE algorithm the facts, that is, the objects stored in the engine working memory, 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 working memory just like a fact, and the engine maintains a time-stamp for each 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, whereas the events are the objects asserted with the `assert-event` primitive. 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. The e_2 `after[`m`,`M`]` e_1 predicate is true iff $d_2 - d_1 \in [m, M]$, where d_i is the time-stamp of the e_i event. The bounds may be infinite. 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.

4.3 Focus on the Join Node

The join nodes are the places where objects which individually satisfy the conditions of a rule are matched against each other. When they match, they are assembled into tuples on which the actions of the rules are eventually executed.

In the RETE algorithm this assembly is incremental: objects matching the first two conditions of a rule are assembled into pairs, which in turn are assembled into triples with the objects matching the third condition, and so on. Each join node has two inputs and one output. The inputs are an n-tuple coming from the join node's parent *left node*, and an object coming from its parent *right node*; the output is an $(n+1)$-tuple and is sent to a child node, which in turn is the parent left node of another join node.

Figure 1 represents the left, right, and join nodes corresponding to the conditions of the rule introduced in the previous section. Let's illustrate on this figure the behaviour of RETE when the second alarm and its confirmation are asserted.

- When the second alarm (noted `Alarm2` in the figure) is asserted, it is stored in the *right2* node and submitted to the *join2* node. The join node matches the alarm against the sole element in its left node's storage, that is, `[Equipment1]`.
- The match is successful, so a pair `[Equipment1,Alarm2]` is formed and transmitted to the *left2* node, which stores it and submits it to the *join3* node.

[2] We consider time as a discrete succession of ticks.

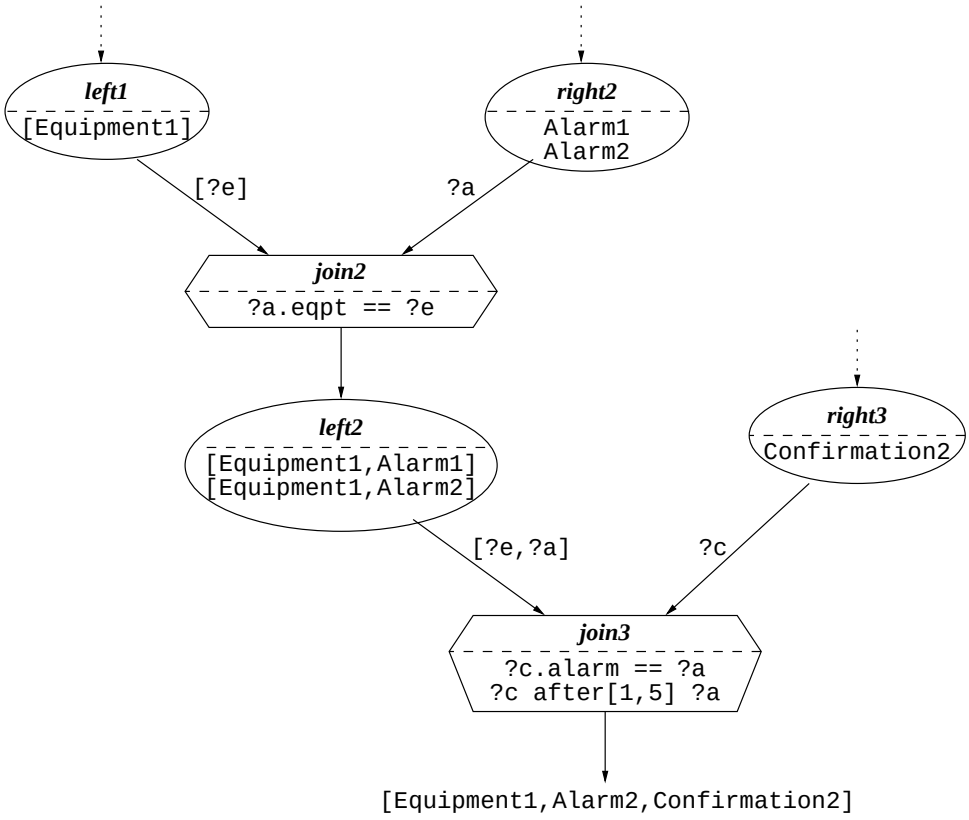

Figure 1: Focus on the join nodes of the RETE network implementing the example rule

- The *join3* node receives the new pair and matches it against all the objects in its right node's storage, that is, none for the time being. This stops the propagation of `Alarm2`.
- When the confirmation for the second alarm (noted `Confirmation2` in the figure) is asserted, it is stored in the *right3* node and submitted to the *join3* node. The join node matches it against all the pairs in its left node's storage. Only the match with the `[Equipment1,Alarm2]` pair is successful, so a triple is formed and sent down.

Note that parent nodes of a join node store tuples and objects, and that these are used by the join node each time a new element is submitted. In other words, the parent nodes store all the candidates for future matches by the join node.

4.4 Extending RETE

When time is not involved, each parent node stores all the tuples or objects it receives, so that the join node may match them with new objects or tuples coming from the other parent node. Furthermore, the node must store them until they are explicitly retracted from the working memory, since a new object matching them may be asserted at any time.

On the other hand, when a condition includes temporal constraints, this gives a limit in time to its satisfiability. This limit is used to bound the time during which objects and tuples are stored in parent nodes of a join node.

For instance, the third condition of the example rule includes the constraint `?c after[1,5] ?a`, that is, "the confirmation must occur between 1 and 5 ticks after the alarm". Assume that the first alarm

occurred at date 10: after date 15, the condition will never match this alarm with any confirmation, because the temporal constraint will never be satisfied. This means that the *left2* node needs only store the `[Equipment1,Alarm1]` pair until date 15, and can then release it.

In contrast, the temporal constraint in *join3* indicates that confirmations only match with alarms that occurred beforehand. This means that the *right3* node never needs to store `Confirmation` instances. Similarly, instances of the `Alarm` class need not be stored by *right2*.

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

- When an element (tuple or object) is submitted to a join node by a parent (left or right) node, the join node computes the element's expiry date with respect to the temporal constraints it stores. This date is computed from the time-stamps of the events borne by the element, and from the bounds in the temporal constraints held by the join node.
- If the expiry date of the element has not yet been reached, the parent node keeps the element at least until this date, so that the join node may match it against new objects or tuples being that it may be submitted. On the contrary, if the element 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 element should be kept or not.
- In the case where the expiry date of the element 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 element. The join node then informs its parent node, which can thus remove the element from its storage.

The dialog takes place at two moments. First when the parent node submits the element to the join node, as an answer from the join node to its parent, telling whether the parent node should store the element for future matches. Second when the expiry date of the element for the join node is reached, as a message from the join node to its parent, telling that the parent node can now stop storing the element. The join node must thus be notified of time changes, or at least of the relevant ones. This can be implemented through a timer mechanism managed by the engine and based on its clock.

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

Observe also that, in the RETE network resulting from the compilation of the rule set, a left or right node can be the parent of several join nodes. This happens when similar conditions appear in several rules. As a result the parent node should store an element as long as at least one of the join nodes requests it.

4.5 Matching Facts With Events

Let us consider an extension of 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 first condition of this rule concerns the `Alarm` class and contains no discrimination tests, just like the second condition of the `AlarmConfirmation` rule. In the RETE network, these two rule conditions

will share the *right2* node. When asserted, alarms will be stored in *right2*, and then passed both to *join2* and to a new join node (name it *join4*) representing the tests in the second condition of the `AlarmCancellation` rule.

Because this new *join4* node contains the `?c after[1,4] ?a` temporal constraint, the dialog between *right2* and *join4* will conclude that the alarms should be kept for 4 ticks by *right2*. (Whereas *join2* doe not require *right2* to keep the alarms.)

Assume now that an `Alarm3` is occurs (at date 30, say) on an equipment which state is 'on-line' and not 'active'. The *right2* node will keep the alarm until date 34. Suppose that the equipment becomes active at date 32, it will become eligible for a match with `Alarm3` in the *join2* node. Would the `AlarmCancellation` rule not have been in the rule set, this match would not have occurred. This is not normal.

What is not normal is that a condition may or not match, depending on the presence or absence of other rules. The *join2* node should behave identically regardless of the result of the dialog between *right2* and *join4*. Namely, it should behave as if `Alarm3` had not been kept by *right2*, and ignore it when the equipment it submitted from *left1*.

We integrate this behaviour into our extension of RETE by formulating the following principle:

> 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 is sufficient to solve our problem as explained below. Consider a rule with fact conditions and event conditions.

- When an event satisfies the first event condition, it is submitted by the discrimination node of this condition to the child join node, to be matched against the tuples in the parent left node.
- Since this is the first event condition, the tuples in the left node only contain facts. These are the facts that satisfied (and still satisfy) the fact conditions preceding the event condition in the rule.
- For successful matches, augmented tuples are passed down to the join node representing the next condition. Event conditions will simply add matching events to the tuples.
- When the next fact condition is reached, the tuple containing events is matched against the facts stored in the parent right node. These facts satisfied (and still satisfy) the newly reached fact condition.
- Because it is a fact condition, the dialog between the join node representing the condition and its parent left node will conclude that the tuples need not be kept by the left node.
- The propagation continues until either all the conditions have been satisfied and the rule is triggered, or no match succeeds in a join node. Along the propagation path, the tuples containing events are only stored in nodes which are parent nodes of join nodes representing event conditions. Parent nodes of join nodes representing fact conditions do not store tuples containing events.
- In the case where a node is, like *right2*, the parent of several join nodes, some of which represent event conditions and other represent fact conditions, it is the job of those join nodes representing fact conditions to ignore the objects stored in its parent node when a new fact satisfies the fact condition.

The principle exposed above states in which time space facts are "visible" to events. For each event it divides the world of facts into their states before the event was asserted, and their states after the event was asserted. (The state of a fact can be defined as the collection of the values of its attributes.) Only the facts in their states before an event was asserted are considered for matches with the event.

The second part of the principle ("until the last one has been satisfied") addressed symmetrical issues, for example the case where the equipment state would change from 'active' to 'on-line' between the occurrences of an alarm and of its confirmation.

5 Benefits

5.1 Non-monotonism

It is usually difficult to write a rule that recognizes a change in the value of an attribute of a fact. Indeed such a rule would typically include two conditions describing the initial and final values of the attribute, but these conditions can not be matched simultaneously. A common solution is to introduce a control object which states that the initial value has been reached, and to write a rule on this object and the final value. Such a solution artificially adds intermediary objects and increases the number of rules, leading to a program which is harder to write, understand, and maintain.

Instead, we can leverage the event concept to express that a fact has reached a given state, or changed its state from some initial to some final value. This can be done either by introducing state change event classes in the application model, or by augmenting the rule language with dedicated constructs which will synthesize the events. Once using events, the rule programmer can even use temporal constraints to express additional conditions on the state changes.

As an example, consider the following rule. It will be triggered when an alarm occurs on an equipment within two ticks after the equipment became active.

```
rule AlarmOnActivation {
  when {
    ?act: event change ?e: Equipment(state == ONLINE) to (state == ACTIVE);
    event Alarm(eqpt == ?e; ?this after[0,2] ?act);
  } then {
    modify ?e { state = OFFLINE; }
  }
};
```

This rule uses the "`event change ... to`" construct which recognizes an initial and a final state on a fact, and generates an event when the state change occurs. Temporal constraints can then be expressed between this and other events.

This illustrates how integrating event management in the rule engine allows to directly address the expression of non-monotonism in the rules.

5.2 Garbage Collection of Events

Another benefit of integrating event management in the rule engine is the management of event retraction. In an application which monitors a flow of events, the facts in the working memory are commonly used to describe the current state of the monitoring system, and events are asserted as they occur to be processed according to the rules. Facts remain in the working memory until the application decides to retract them, based on its model of the monitoring system. Events need to remain in the working memory as long as they participate to the processing logic, but should be retracted as soon as possible once they no longer play a role, in order not to affect performance.

The handling of event retraction by the application requires the programmer to take into account all the rules and the temporal constraints they include. It can be implemented using additional rules or in the application procedural code. In all cases this part of the program is very fragile, and difficult to test, debug and maintain.

When event management in integrated in the rule engine, this task can be performed by the engine itself. The expiry date of each event is computed at the join node level, as explained in section 4.4, each time a temporal constraint involves the event. The engine can thus detect when an event has expired throughout the RETE network, and then retract it from the working memory.

This automatic retraction mechanism mimics the work of garbage collectors in regular programming languages. Here it applies to events in the engine working memory, to be compared with objects in the program heap.

6 Conclusion

In this paper we presented an extension of the RETE algorithm to integrate event management in a rule engine. This extension uses 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. The expression power available to rule programmers is augmented, thanks to a more integrated handling of non-monotonism and to the automatic retraction of obsolete events. More generally, the rule-based programs become easier to write, debug, understand, and maintain.

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.

Future work directions could be twofold. The algorithm presented here is in essence incremental. Other work on chronicle recognition use techniques such as domain propagation to leverage properties at the rule or at the ruleset level, and it could be interesting to integrate these techniques crosswise to the incremental approach. Also, the acquisition of expertise is a common problem in A.I. systems, and important pieces of work [2, 15, 17, 5] exist on the subject of extracting and learning temporal patterns from sets of data such as alarm logs, which often exist in monitoring applications.

References

[1] Apama. `http://www.apama.com`

[2] S. Bibas, M.-O. Cordier, P. Dague, F. Lévy, L. Rozé. Scenario generation for telecommunication network supervision. *Workshop on A.I. in Distributed Information Networks,* 1995.

[3] M.-O. Cordier, C. Dousson. Alarm driven monitoring based on chronicles. *SafeProcess (to appear),* 2000.

[4] C. Dousson. Suivi d'évolutions et reconnaissance de chroniques. *Thèse de doctorat, Université Paul Sabatier, Toulouse,* 1984.

[5] C. Dousson, T. Vu Duong. Discovering chronicles with numerical time constraints from alarm logs for monitoring dynamic systems. *IJCAI,* 1999.

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

[7] C. Forgy. RETE: A fast match algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence,* n° 19, pp. 17–37, 1982.

[8] France Télécom R&D. `http://crs.elibel.tm.fr`

[9] M. Ghallab, On chronicles: Representation, on-line recognition and learning. *KR,* 1996.

[10] M. Ghallab, A. M. Alaoui. Extended planning through preprocessing of knowledge: the IxTeT approach. *AAAI Workshop on Automated Planning for Complex Domains,* 1990.

[11] The Haley Enterprise. `http://www.haley.com`

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

[13] ILOG. `http://www.ilog.com`

[14] J.-P. Krivine, O. Jehl. The AUSTRAL system for diagnosis and power restoration: an overview. *ISAP,* 1996.

[15] P. Laborie, J.-P. Krivine. Automatic generation of chronicles and its application to alarm processing in power distribution systems. *DX,* 1997.

[16] F. Lévy. Recognising scenarios: a study. *DX*, 1994.

[17] E. Mayer. Inductive learning of chronicles. *ECAI*, 1998.

[18] D. Miranker. TREAT: A better match algorithm for AI production systems. *National Conf. on AI*, 1987.

[19] ONERA. `http://www.onera.fr/dtim/intartdis/crs.html`

[20] Production Systems Technologies. `http://www.pst.com`

[21] Sandia National Laboratories. `http://herzberg.ca.sandia.gov/jess`

[22] SpiritSoft. `http://www.spirit-soft.com`