Chronicle Recognition Improvement Using Temporal Focusing and Hierarchization

Christophe Dousson and Pierre Le Maigat
France Telecom R&D,
2 avenue Pierre Marzin,
22307 Lannion cedex, France.
{christophe.dousson,pierre.lemaigat}@orange-ftgroup.com

Abstract

This article falls under the problem of the symbolic monitoring of real-time complex systems or of video interpretation systems. Among the various techniques used for the on-line monitoring, we are interested here in the temporal scenario recognition. In order to reduce the complexity of the recognition and, consequently, to improve its performance, we explore two methods: the first one is the focus on particular events (in practice, uncommon ones) and the second one is the factorization of common temporal scenarios in order to do a hierarchical recognition. In this article, we present both concepts and merge them to propose a focused hierarchical recognition. This approach merges and generalizes the two main approaches in symbolic recognition of temporal scenarios: the Store Totally Recognized Scenarios (STRS) approach and the Store Partially Recognized Scenarios (SPRS) approach.

1 Introduction

Symbolic scenario recognition arises in monitoring of dynamic systems in many areas such as telecommunications networks supervision, gas turbine control, healthcare monitoring or automatic video interpretation (for an overview, refer to [Cordier and Dousson, 2000]).

Such scenarios could be obtained among other things by experts, by automatic learning [Fessant *et al.*, 2004; Vautier *et al.*, 2005] or by deriving a behavioral model of the system [Guerraz and Dousson, 2004]. Due to the symbolic nature of those scenarios, the *engine* performing the recognition is rarely directly connected to sensors. There is often (at least) one dedicated calculus module which transforms the "raw" data sent by the system into symbolic events. Typically this module can compute a numerical quantity and sends symbolic events when the computed value reaches given thresholds. In cognitive vision, this module is usually a video-processing which transforms images into symbolic data.

Often those scenarios are a combination of logical and temporal constraints. In those cases, symbolic scenario recognition can process the scenarios uniformly as a set of constraints (like the Event Manager of ILOG/JRules based on a modified RETE algorithm for processing time constraints [Berstel, 2002]) or separate the processing of temporal data from the others like in [Dousson, 2002]. This article mainly deals with this second approach where temporal constraints are managed by a constraint graph between relevant time points of the scenarios. There are two approaches for dealing with temporal constraints: STRS recognizes scenarios by an analysis of the past [Rota and Thonnat, 2000] and SPRS which performs an analysis of scenarios that can be recognized in the future [Ghallab, 1996]. Two main problems in the SPRS approach are the fact that scenarios have to be bounded in time in order to avoid never expected ending scenario (in practice, when working on real-time systems, it is difficult to exhibit scenario which cannot be bounded in time); and, second, that SPRS engine has to maintain all partially scenarios which possibly leads to use a large amount of memory space. To partially avoid those drawbacks, the implementation of SPRS algorithms in [Dousson, 2002] introduces a clock and deadlines which are used to garbage collect the pending scenarios. On the other hand, the main problem with STRS algorithms is to maintain all previously recognized scenarios. To our knowledge, no work has been published on how long such scenarios should be maintained. In addition, STRS does not provide any kind of "prediction" as SPRS does.

A first attempt to take the benefits of both approaches was made in [Vu *et al.*, 2003]. It consists of a hierarchization of the constraint graph of the scenario. It deals only with graphs where all information about time constraints can be retrieved from a path where temporal instants can be totally ordered. The hierarchy constructs an imbricated sequence of scenarios containing only two events at a time. The principle of the recognition is, at any instant, to instantiate elementary scenarios and when an event is integrated in a high-level scenario, looking for previously recognized elementary scenarios. The purpose of this article is to generalize this method; the starting point will be an SPRS approach and the generalization mixes reasoning on past and future. As a byproduct, STRS and SPRS methods appear as two extreme kinds of the propose focused hierarchical recognition.

The next section presents the used SPRS approach and details some aspects which are relevant to this paper. The section 3 is dedicated to the temporal focusing which enables the system to focus on uncommon events prior to others. Events

could be not only basic events coming directly from the supervised system but aggregated indicators. So the temporal focusing could be used in order to control the computation of such indicators on particular temporal windows and to avoid useless computation. As such indicators could be themselves scenarios, section 4 presents how the hierarchical recognition deals with common subscenarios. Finally, we show that both concepts could be merged and experimentally lead to good improvement of performances. This will be the object of the section 5.

We conclude in section 6 by experimentation on detecting naive servers in a Reflected Distributed Denial of Service (RDDoS) attack.

2 Chronicle Recognition System

Our approach is based on the chronicle recognition as proposed in [Dousson, 2002] which falls in the field of SPRS methods. A chronicle is given by a time constraint graph labeled by predicates.

An instance of event is a pair (e, t) where t is the date of the event and e is its type. When no ambiguity results, we sometimes do not distinguish between an event and its type. Figure 1 shows a chronicle which contains four events: the event e (if instantiated) must occur between 1 and 3 units of time after an instantiation of f, the event g must occur between 0 and 3 units of time after e and between -1 and 4 units of time after e'.

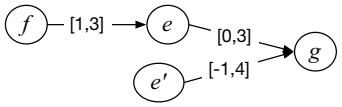

Figure 1: A chronicle.

Note that the complete formalism is based on a reified logic (a chronicle is a conjunctive formula) and introduces also predicates on persistency or event absence. In this article we choose to present in details the focusing from the time constraint graph point of view, other predicates are also taken into account but not discussed here.

2.1 Recognition Algorithms

Let CRS (Chronicle Recognition System) denote the algorithm of recognition. Basically the mechanism of CRS is, at each incoming event, to try to integrate it in all the pending (and partial) instances of the chronicle model and/or created a new instance and calculating (using constraint propagation [Dechter *et al.*, 1991]) new forthcoming windows for all the forthcoming events of each instance. An instance of a chronicle model is then a partial instantiation of this model and forthcoming windows $\mathrm{fc}(e)$ of a non-instantiated event e is the (extended[1]) interval where the occurrence of an event could lead to a recognition[2].

Figure 2 shows the mechanism of the recognition algorithm on a small example: when CRS receives f at 1, it creates the new instance I_1 and updates the forthcoming window of the node e. When a new f occurs at 3, instance I_2 is created (the forthcoming windows of I_1 is updated "using" a clock tick set at 3). When e occurs, I_3 is created (from I_2) and I_1 is destroyed as no more event e could from now be integrated into (instance I_2 is not destroyed, waiting for another potential e between 5 and 6). As all events of I_3 are instantiated the chronicle I_3 is recognized.

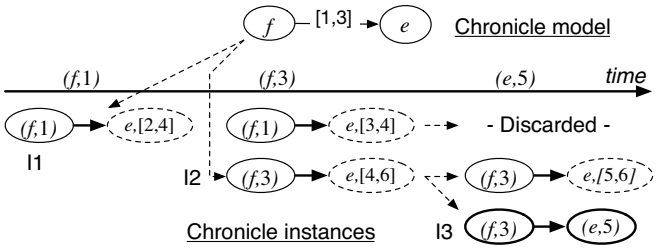

Figure 2: Created instances of a chronicle by the incoming event stream $(f, 1)(f, 3)(e, 5)$.

2.2 From Clock to "AssertNoMore" Event.

In first implementations of chronicle recognition, a clock was introduced in order to discard "impossible" instances when the clock goes past one of their forthcoming windows (see behavior of I_1 in figure 2).

In order to take into account some jitter in data transmission a possible delay δ can be taken into account. This delay bounds the maximum difference observed at reception between two events sending at the same time by the (possibly distributed) system. Basically, the event integration algorithm could be written as following:

$$\mathrm{integrate}((e, t));$$
$$\mathrm{setGarbageClock}(t - \delta)$$

The main drawback is that it implies that events arrive roughly in a FIFO manner (the allowed jitter is bounded by δ): so, when the FIFO hypothesis should be relaxed (and it is often the case when the monitored system is distributed), δ should be increased and the garbage efficiency decreases [3].

In order to avoid this, instead of a clock, we define a new input message: "AssertNoMore(e, I)", where e is an event type and I an extended interval. It specifies to CRS that, from now on, it will not receive more events of type e with an occurence date in I. This mechanism is implemented in CRS by managing one forthcoming window for each event type which is updated when receiving an "AssertNoMore" message.

We do not describe here how CRS deals with this assertion as it is very close to the previous CRS. Intuitively, all the forthcoming windows of e are reduced ($\mathrm{fc(e)} \setminus \mathrm{I}$), these windows are propagated according to the constraint graph of the

[1] An extended interval is a disjoint union of intervals.

[2] This does not imply that, for all non instantiated events e, if (e, t) occurs with $t \in \mathrm{fc}(e)$ then the instance is recognized. This a difference between SPRS mechanism where integration of events is incremental and some STRS mechanism where integration is made by block in a backward manner.

[3] In case of focusing, an other side effect of using clock is the creation of "false" instances. This will be explained in section 6.

chronicle and, if a forthcoming window becomes empty, the instance is discarded. The previous garbage collect of CRS could be emulated by the following:

$\text{integrate}((e,t))$;
$\forall$ event type e', AssertNoMore$(e',]-\infty, t-\delta])$

It illustrates that the previous clock management forces all the event types to be synchronized: in other words, if one kind of event could be widely jittered, then all event types are supposed to be the same. Notice that, as we allow the use of extended intervals, more complex garbage management could be easily implemented: it could be different from $]-\infty, \text{clock}]$ and non chronological: we could process, for instance, "AssertNoMore$(e, [10,20] \cup [30,40])$" and, then, "AssertNoMore$(e, [0,10])$".

This point of view changes slightly the manner the engine works: the progress of time is no more driven by the observations coming from the supervised system but by an "internal" control, this control could be given by the knowledge of the temporal behavior of the system. We will see in the next section how this new feature is essential for an efficient temporal focusing.

3 The Temporal Focusing

3.1 General Description

In some cases, not all events have the same status: in example on figure 2, event f could be very frequent and event e extremely uncommon. Due to this difference of frequency, the recognition of the chronicle could be intractable in practice, indeed each event f potentially creates a new instance of the chronicle waiting for other events: if a thousand f arrive between 0 and 1, a thousand instances will be created. As event e is extremely uncommon most of those instances would be finally destroyed. In CRS, the number of creation of instances has a great impact on performance of the recognition. In order to reduce this number, we focus on event e in the following manner: when events f arrive, we store them in a collector and we created new instances only when events e occur, then we will search in the collector which f could be integrated in the instances. Potentially the number of created instances will be the number of e and not the number of f. In order to be not limited to uncommon events, we order the recognition process by introducing a *level* for each event type. The principle of the temporal focusing is then the following:

Temporal focusing: *Begin the integration of events of level $n+1$ only when there exists an instance such that all events of level between 1 and n had been integrated.*

So, if $\text{level}(e) \leq \text{level}(f)$, the engine will integrate e at $t=5$ then it will search in the collector all events f between $t=2$ and $t=4$. The collector finds $(f,3)$ and sends it to the engine, this leads to the creation of the instance I_3. Technically we made the choice of sending f not only to instances waiting for events of same level but to all the pending instances. As the number of f sending from the collector to the engine is small, only a small number instances are created. This also ensures that the collector sends an event one and only one time. So, in our example, $(f,3)$ leads to the creation of instance I_2 with $\text{fc}(e) = [5,6]$.

In addition to the recognition engine, we developed some modules (see figure 3): an *Event Router* which routes the events coming from the supervised system to either the engine or the collector; a module named *Finished Level Detector* which detects when a particular instance of chronicle has finished to integrate all events of level lesser than n, then looks at the forthcoming windows of events of level $n+1$ and sends them (through a "Focus" message) to the collector as a request for events. At last, a module called *No Need Window Calculator*, which computes for all current instances the windows where they do not need events of greater level and sends this information to the collector via a "Discard" message. In order to make the buffer management easier, the collector itself is split into *Buffered Streams*: one for each event type.

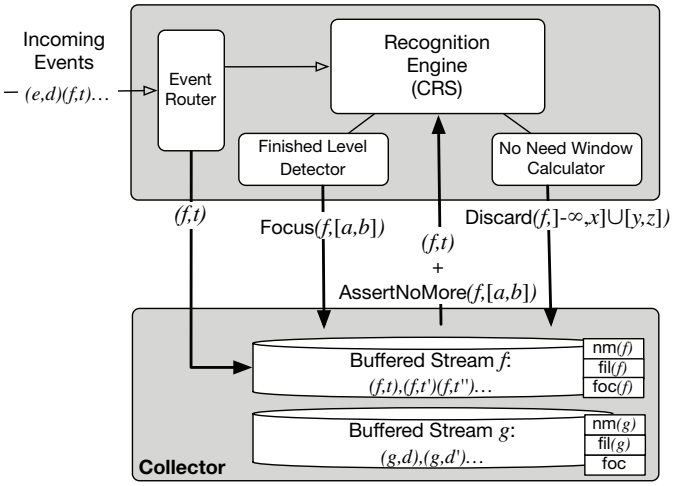

Figure 3: Architecture of the focused recognition.

At this stage, we have defined two messages for a feedback control on the production of event of high level: CRS is able to focus on specified time windows (by sending a "Focus" message) but also could discard events in irrelevant time windows (by sending a "Discard" message).

3.2 The Buffered Streams

The collector is composed of many buffered streams, each of them is dedicated to one event type. This section presents how it works: each buffered stream manages three particular temporal windows (see figure 4) :

i) the *assert no more window* (nm) which contains all the occurrence dates for that no more event will be received from now,

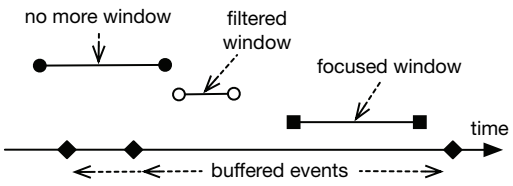

Figure 4: The 3 control windows used to manage the buffer.

ii) the *filtered window* (fil) which contains all the occurrence event dates for which the pending instances don't care, and

iii) the *focus window* (foc) which contains all the occurrence dates for which incoming events are required and should be integrated immediately by CRS without be stored.

The figure 5 illustrates the different cases of processing an event according to its position against the temporal windows of the buffered stream.

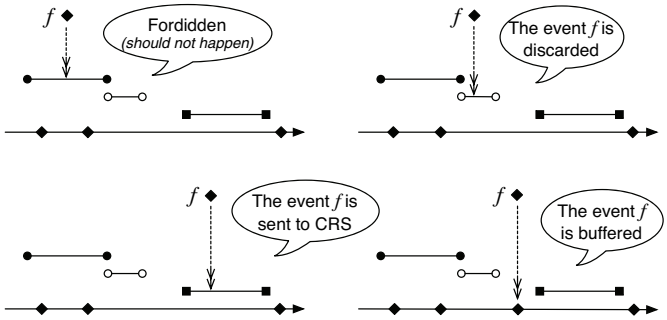

Figure 5: The different cases of processing an event occurrence into a buffered stream (according to the event date).

Reception of control messages ("AssertNoMore", "Discard" or "Focus") updates also the buffered stream:

- **"AssertNoMore**(f, w)**"** : it updates the windows (see fig. 6) $\mathrm{nm} \leftarrow \mathrm{nm} \cup w$, $\mathrm{fil} \leftarrow \mathrm{fil} \backslash w$ and $\mathrm{foc} \leftarrow \mathrm{foc} \backslash w$ and sends to CRS "AssertNoMore$(f, w \cap \mathrm{foc})$".
- **"Discard**(f, w)**"** : it destroys all events in w, then updates the windows : $\mathrm{fil} \leftarrow \mathrm{fil} \cup (w \backslash \mathrm{nm})$, $\mathrm{foc} \leftarrow \mathrm{foc} \backslash w$ and sends back to CRS "AssertNoMore(f, w)"[4].
- **"Focus**(f, w)**"** : it sends all events in w, then updates the focused window: $\mathrm{foc} \leftarrow \mathrm{foc} \cup (w \backslash (\mathrm{nm} \cup \mathrm{fil}))$ and sends back to CRS "AssertNoMore$(f, w \cap (\mathrm{nm} \cup \mathrm{fil}))$".

So, during the recognition process, the following properties are always true : i) The windows nm, fil and foc are mutually exclusive. ii) There is no buffered event in the filtered and the focused windows. Whatever the levels in a chronicle, this mechanism performs exactly same recognitions as CRS with no focusing; only the performances are affected.

3.3 Partial order and relative event frequencies

In figure 2, if e is much more frequent than f, there is no need to define level, as, if no f was arrived, no e initiates new instances, thus the number of partial instances equals number of f. So relatively to an other event e, event f (very frequent) should be leveled only if $e \not\leq f$ (in the partial order induced by the constraints), in particular when e and f are not ordered, for example if $e \xrightarrow{[-1,2]} f$. We can decompose instances into two categories: the instances where f is before e for which the focusing will be particular efficient and the other part where f is after e, in this case f should be directly sent to the recognition engine which is done by the use of the maintained focus window. An other case where focusing is useful is when $f \xrightarrow{[a,b]} e$, with $a, b \geq 0$, with e FIFO but f having a jitter delay greater than b. To sum up, setting a level to an event is based on an *a priori* knowledge of its incoming frequency and on its position in the constraint graph.

The focusing must be combined with hierarchization in the case where one can identify uncommon sub-patterns or when more than one occurrence of a frequent event is required for the recognition (see section 5 and 6 for an example).

[4]This message is a kind of acknowledgment used to synchronize the event buffer state and the CRS state.

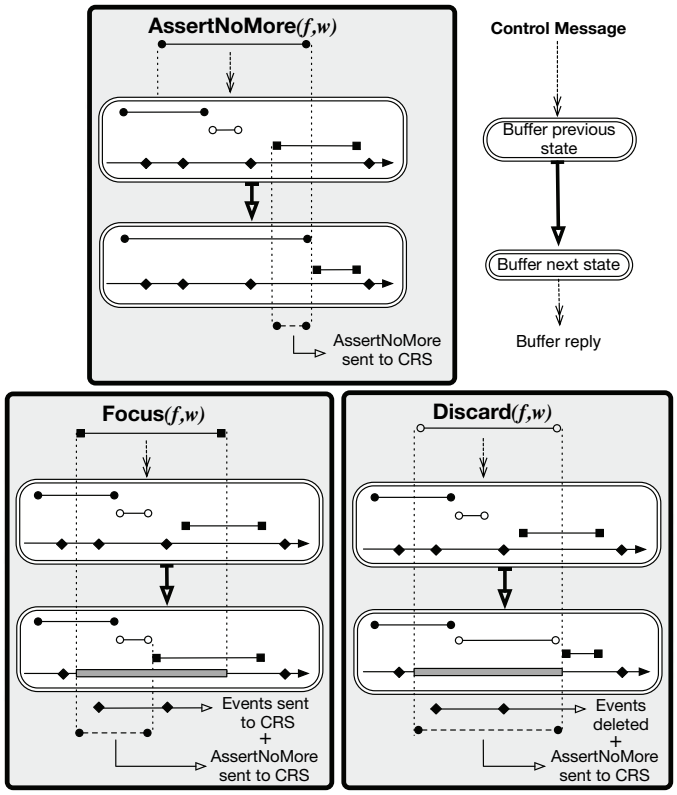

Figure 6: Behavior of a buffer when it receives one control message ("AssertNoMore", "Discard" or "Focus").

4 Hierarchical Chronicles

A *hierarchical chronicle* is a pair $(C, \{h_1, \ldots, h_n\})$ where C is the *base* chronicle and h_i are the *sub-chronicles*; we assume that events involved in C can take value in the set of sub-chronicle labels $\{h_1, \ldots, h_n\}$. We treat only *deterministic* hierarchical chronicles[5], i.e. we do not allow two sub-chronicles having the same label, so, in the following, we make no distinction between the chronicle and its label. Moreover we suppose that each h_i has a distinguished event b_i. The hierarchical chronicle C will be *recognized* if it is

[5]There is no technical difficulty to consider non-deterministic hierarchical chronicles. The main difference is that expansion (see below) leads to more than one chronicle. This is a possible way to introduce disjunction in chronicle formalism.

(classically) recognized with integrated events labeled by a sub-chronicle h_i have the date of an integrated event b_i in a recognized instance of h_i. In other words, when a sub-chronicle h_i is recognized, it "sends" to all chronicles the event $(h_i, \text{date of } b_i)$.

In figure 7, the hierarchical chronicle $H = (C, \{h, k\})$ possesses two sub-chronicles h and k, the chronicle k is composed of two types of events: a basic one, f (which comes from the supervised system) and one from the sub-chronicle h. The distinguished events are respectively e and h.

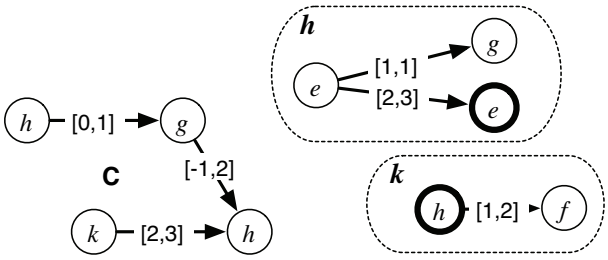

Figure 7: The hierarchical chronicle $(C, \{h, k\})$.

Let C be a chronicle with constraint graph G and with an event h, where h is a subchronicle with distinguished element b and with constraint graph V. *Expanding* the chronicle C (or its graph) is replacing the node $h \in G$ by the graph V, specifying that the constraints between b and $G \backslash V$ are the constraints of $h \in G$. Let the relation $h_i \rightarrow h_j$ be defined if h_j contains the event type h_i. The hierarchical chronicle is *structurally consistent* if $\forall i, \nexists h_i \rightarrow^* h_i$. In this case the *expansion* of a hierarchical chronicle is well defined (the graph of the figure 8 is the expansion of $(C, \{h, k\})$). A hierarchical chronicle C is *consistent* if there exists a set of events s.t. C is recognized. It is straightforward that a (structurally consistent) hierarchical chronicle is consistent iff when expanding sub-chronicles the obtained constraint graph is consistent.

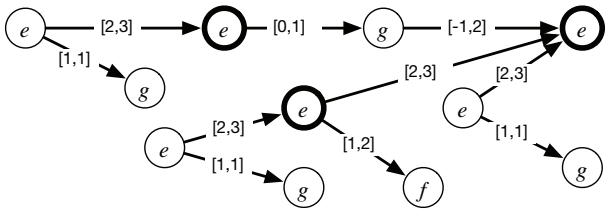

Figure 8: The expansion of the hierarchical chronicle C.

Hierarchical chronicles can come from two ways: the first one is that the chronicle is initially specified in a hierarchical manner, for example if the architecture of the system is itself hierarchical; the second one is, starting from a flat chronicle, we identify identical patterns inside this chronicle. The next proposition is a necessary and sufficient condition to the hierarchization of a (temporal) pattern and shows we have to take care in pattern factorization: two subgraphs could be identical (with same constraints) but one of them can satisfy the condition and the other not.

Proposition 4.1 *Let G be a minimal constraint graph and $U \subseteq G$ a subgraph. Let $b \in U$ be a (distinguished) node and H the hierarchical chronicle defined by $G' = (G \backslash U) \cup \{h\}$ and $h = U$, where $h \in G'$ has the same constraints as node b. Then H and G recognize the same events iff*

$$\forall a \in U, \forall c \in G \backslash U, D_{ac} = D_{ab} + D_{bc} \quad (1)$$

where D_{xy} is the time constraint between nodes x and y in the graph G.

5 Focused Hierarchical Recognition

In this section, we present how focusing and the hierarchization could be mixed. We need to add a module which transforms the windows of high level events representing sub-chronicles into those of the different buffered streams representing the events in the sub-chronicles.

We only present the formula for updating the filtered window: when the collector receives a "Discard" message for f, it transforms it into a "Discard" message for all events of the sub-chronicle f, but needs also to take into account other high-level events. In order to do that, we also introduce a filtered window for all high level events representing a sub-chronicle.

So, the window of the "Discard" message receives by the buffered stream of event type e (included in f) is given by: $\bigcap_{e \in f_i} \text{fil}(f_i)/D_{e,b_i}$ where D_{e,b_i} is the time constraint from e to the distinguished node b_i of f_i and where:

$$[u, v]/[a, b] = \begin{cases} [u - a, v - b] & \text{if } u - a \leq v - b \\ \varnothing & \text{otherwise} \end{cases}$$

with natural extension to the set of extended intervals.

Remark: An example of gain of focusing vs. techniques presented in [Vu *et al.*, 2003] is that for subgraphs with $[0, 0]$ constraints (co-occurrence), we don't have to explore combination of partial instantiation but only store events and wait for an event in the future in order to extract from the collector events with the "good" parameters; thus all the combinatorial explosion of elementary scenarios is avoided.

6 Experimentation

This experiment was done on real data coming from the real case application which motivates this work: the detection of RDDoS (Reflected Distributed Denial of Service) attacks in IP networks.

In such attack, machines with the spoofed IP address of the victim send SYN messages to naives servers. Then, those servers will reply by sending to the victim SYN/ACK messages generating a massive flooding. Characteristic of the attack is that SYN traffic to naive server is low, persistent and, taken alone, do not trigger an alarm. We want to identify the naive servers. In our experiment, information on the global traffic are sent way up by dozens of core routers. Preprocessing is done by a particular numerical algorithm which computes throughput between pairs of IP addresses and sends alarms when this throughput is greater than two thresholds: a low one (L events) and a high one (H events).

So CRS receives two kinds of events: H[ip_dest] and L[ip_scr, ip_dest] where variables are IP addresses of the source and of the destination. To identify the IP addresses of the naive servers, the focused hierarchical chronicle to recognize is :

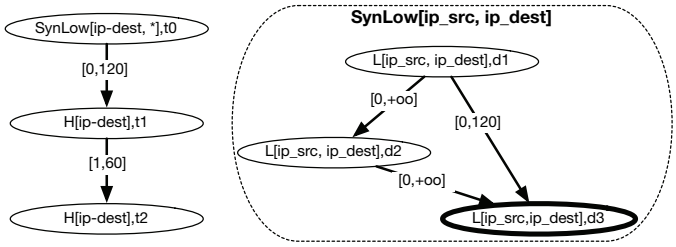

Figure 9: The focused chronicle to detect naive servers.

where event H is of level 1 and SynLow of level 2. We do not discuss here the choice of threshold nor the relevance of the chronicle, our aim is to present performance of focusing face to huge amount of data. The log contains $\simeq$240,000 events (L and H), its period is equal to $\simeq$6min, frequencies are $\simeq$660 L/sec and $\simeq$3 H/sec. Due to a lack of synchronization of routers the jitter delay is set to $60s$. We compare processing time[6] for 4 different cases: on one hand, using previous CRS with clock management (clk) [Dousson, 2002] or using CRS with "AssertNoMore" extension (anm), and on the other hand by using a hierarchical chronicle with focusing (fh) or the corresponding flat chronicle.

	processing time	created instances	maximal collector's size
CRS+clk	17 min.	236 600	*n.a.*
CRS+anm	14 min.	236 600	*n.a.*
fh-CRS+clk	3 min.	3550	4800
fh-CRS+anm	7 s.	1840	1330

Considering only the anm extension, the processing time decreases a little since the garbage collect is more efficient: partial instances have a shorter lifetime in memory. The amount of created instances is considerably reduced when using focusing. The difference between number of instances when using clock (3550) and when using anm (1840) is explained on this example: for the leveled chronicle of figure 2, when f arrives at $t = 1$, we store it and the clock is set to 1; but when $(e, 3)$ arrives, the clock can not be set to 3 otherwise all current instances would be discarded; so, even if in the FIFO hypothesis, it is necessary to set the delay to 3 (the upper bound of the constraint between f and e). By this artifact, a (possible large) number of "false" instances are uselessly created. The introduction of this delay has also an incidence on the collector's size.

7 Conclusion

This paper presents two improvements of chronicle recognition. The first one is the focusing on particular events which allows the system to reason on the past and on the future in a homogeneous manner. The second concerns the hierarchical recognition based on subpatterns. We also showed that mixing both improvements increases the efficiency and also fills the gap between SPRS and STRS approaches which are completely covered. In practice, our approach is sufficiently adaptive in order to fine-tune the recognition system. For instance, the order of event integration could be different from their arrival order. Moreover, focused events could postpone some expensive numerical computations to generate events and avoid useless ones.

Future works will take two directions: the first one is to define a more flexible way to factorize common patterns - condition 1 (proposition 4.1) is too restrictive for many practical cases. As dealing with the event frequency substantially increases the efficiency of CRS, the second direction will be focused on online analysis of event frequency in order to adapt dynamically the hierarchical temporal focusing.

[6]Performed on a PPC G4 biprocessor 2×1GHz with 1Gb SDRAM. Java implementation was run on JVM 1.4 / Mac OS X.

References

[Berstel, 2002] B. Berstel. Extending the RETE algorithm for event management. 9^{th} *International Symposium on Temporal Representation and Reasoning (TIME'02)*, pages 49–51, 2002. IEEE Transactions.

[Cordier and Dousson, 2000] M.-O. Cordier and C. Dousson. Alarm Driven Monitoring Based on Chronicles. In *Proc. of the* 4^{th} *Symposium on Fault Detection Supervision and Safety for Technical Processes (SAFEPROCESS)*, pages 286–291, Budapest, Hungary, 2000. IFAC.

[Dechter *et al.*, 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence, Special Vol. on Knowledge Representation*, 49(1-3):61–95, 1991.

[Dousson, 2002] C. Dousson. Extending and unifying chronicle representation with event counters. In *Proc. of the* 15^{th} *ECAI*, pages 257–261, Lyon, France, 2002. IOS Press.

[Fessant *et al.*, 2004] F. Fessant, C. Dousson, and F. Clérot. Mining of a telecommunication alarm log to improve the discovery of frequent patterns. 4^{th} *Industrial Conference on Data Mining (ICDM'04)*, 2004.

[Ghallab, 1996] M. Ghallab. On chronicles : Representation, on-line recognition and learning. *Proc. of the* 5^{th} *International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pages 597–606, 1996.

[Guerraz and Dousson, 2004] B. Guerraz and C. Dousson. Chronicles Construction Starting from the Fault Model of the System to Diagnose. 15^{th} *International Workshop on Principles of Diagnosis (DX'04)*, pages 51–56, 2004.

[Rota and Thonnat, 2000] N. Rota and M. Thonnat. Activity recognition from video sequences using declarative models. 14^{th} *ECAI*, pages 673–677, 2000. IOS Press.

[Vautier *et al.*, 2005] A. Vautier, M.-O. Cordier, and R. Quiniou. An inductive database for mining temporal patterns in event sequences. *Workshop mining spatio-temporal data (in PKDD05)*, 2005.

[Vu *et al.*, 2003] V.-T. Vu, F. Bremond, and M. Thonnat. Automatic video interpretation: A novel algorithm for temporal scenario recognition. 18^{th} *IJCAI*, 2003.