ECAI 2008
M. Ghallab et al. (Eds.)
IOS Press, 2008

doi:10.3233/978-1-58603-891-5-194

Chronicles for On-line Diagnosis of Distributed Systems

Xavier Le Guillou and **Marie-Odile Cordier** and **Sophie Robin** and **Laurence Rozé** [1]

Abstract. The formalism of chronicles has been proposed to monitor and diagnose dynamic physical systems. Even if efficient chronicle recognition algorithms exist, it is now well-known that distributed approaches are better suited to monitor real systems. In this article, we adapt the chronicle-based approach to a distributed context and illustrate this work on the monitoring of software components.

1 Introduction

Monitoring and diagnosing dynamic systems have become very active topics in research and development for a few years. Besides continuous models based on differential equations, essentially used in control theory and discrete event systems based on finite state machines (automata, Petri nets, ...), a formalism commonly used for on-line monitoring, in particular by people from the artificial intelligence community, is the one of chronicles. This formalism, proposed in [10], has been widely used and extended [7, 9, 6]. A chronicle describes a situation that is worth identifying within the diagnosis context. It is made up of a set of events and temporal constraints between those events. As a consequence, this formalism fits particularly well problems that consider a temporal dimension. The set of interesting chronicles constitutes the base of chronicles. Then, monitoring the system consists in analyzing flows of events, and recognizing on fly patterns described by the base of chronicles. Efficient algorithms exist and this approach has been used for industrial applications as well as medical ones [7, 14, 2].

One of the key issues of model-based approaches for on-line monitoring is the size of the model which is generally too large when dealing with real applications. Distributed or decentralized approaches have been proposed to cope with this problem, like [5, 8, 1, 13]. The idea is to consider the system as a set of interacting components instead of a single entity. The behavior of the system is thus described by a set of local component models and by the synchronization constraints between the component models.

Considering chronicle-based approaches, to our knowledge, no distributed approaches exist and the contribution of this paper consists in adapting the chronicle-based approach to distributed systems.

This work has been motivated by an application that aims at monitoring the behavior of software components, and more precisely of web services within the context of the WS-DIAMOND (Web Service DIAgnosability, MONitoring and Diagnosis) European project. In this context, a request is sent to a web service which collaborates with other services to provide the adequate reply. Faults may propagate from one service to another and diagnosing them is a crucial issue, in order to react properly. We use a simplified example of an e-foodshop to illustrate our proposal.

We first recall the principles of the chronicle recognition approach and give basic definitions in section 2. We introduce in section 3 the simplified example that will be used all along this paper. In section 4, we show how to extend the chronicle-based approach to distributed systems. We first describe the architecture of a chronicle-based distributed system (4.1). Then we extend the chronicle formalism to deal with synchronization constraints (4.2). We describe in 4.3 a push-pull algorithm able to compute a global diagnosis from the local diagnoses, computed by locally distributed chronicle recognition systems, checking the synchronization constraints. After an illustrative example in 4.4, we compare our proposal to related work in section 5 and conclude in section 6.

2 Chronicle recognition approach

The chronicle recognition approach (first introduced in [10]) relies on a set of patterns, named chronicles, which constitutes the chronicle base. Let us recall the formalism and the chronicle recognition algorithm.

2.1 Formalism of chronicles

A chronicle is a set of observable events which are time-constrained and is characteristic of a situation.
An **event type** defines what is observed within the system, for instance the name of an activity act, the name augmented with the fact that the activity is starting (namely act^-) or ending (namely act^+), the name enriched with observable parameters $act(?var_1, \ldots, ?var_n)$ or a combination of those possibilities. $\mathcal{E}$ denotes the set of possible event types. An **event** is a pair $(e, ?t)$ where $e \in \mathcal{E}$ is an event type and $?t$ the occurrence date of the event.
A **chronicle** (model) $\mathcal{C}$ is a pair $(\mathcal{S}, \mathcal{T})$ where $\mathcal{S}$ is a set of events and $\mathcal{T}$ a set of constraints between their occurrence dates. When its variables and its occurrence dates are instantiated, a chronicle is called a **chronicle instance**.

2.2 Chronicle recognition

A chronicle recognition tool, called CRS (Chronicle Recognition System), has been developed by C. Dousson[2]. It is in charge of analyzing the input stream of events and of identifying, on the fly, any pattern matching a situation described by a chronicle. Chronicles are compiled into temporal constraint networks which are processed by efficient graph algorithms. CRS is based on a complete forecast of the possible dates for each event that has not occurred yet. This set (called temporal window) is reduced by propagation of the dates of observed events through the temporal constraint network. When a new event arrives in the input stream, new instances of chronicles are generated in the set of hypotheses, which is managed as a tree.

[1] Irisa – Université de Rennes 1, France, email: xleguill@irisa.fr

[2] http://crs.elibel.tm.fr/

Instances are discarded as soon as possible, when constraints are violated or when temporal windows become empty.

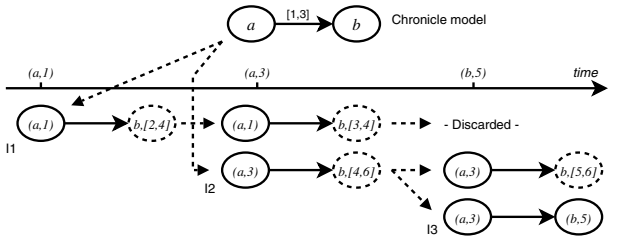

Figure 1. Principle of chronicle recognition

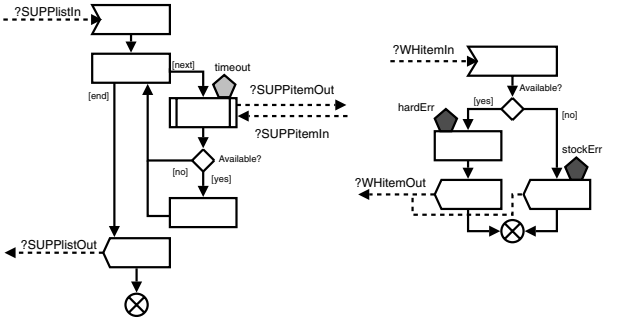

Figure 3. Simplified workflows of the SUPP and the WH

Figure 1 shows the principle of the recognition algorithm on a very simple example: a single chronicle model is defined, containing only two events: $(a, ?t_a)$ and $(b, ?t_b)$, with $?t_a + 1 \leq ?t_b \leq ?t_a + 3$. When event $(a, 1)$ is received, instance $I1$ is created, which updates the temporal window of the related node b. When a new event $(a, 3)$ occurs, a new instance $I2$ is created and the forthcoming temporal window of $I1$ is updated. When event $(b, 5)$ is received, instance $I3$ is created (from $I2$) and $I1$ is destroyed as no more event $(b, ?t_b)$ could match the temporal constraints from now on. Instance $I2$ is still waiting for another potential event $(b, ?t_b)$ before $?t_b > 6$. As all the events of $I3$ are instantiated, this instance is recognized.

3 Motivating example

To illustrate the ideas developed in this paper, we consider an orchestration of three web services, a shop, a supplier and a warehouse, that provide e-shopping capabilities to users. This application keeps the essential properties of the applications we aim to monitor. In particular, we consider closed environments, where a workflow-like description of each web service (Figures 2 and 3) involved in the processing of the request is supposed to be available.

A customer wants to place an order and selects items on the shop. This list of items is transferred to a supplier which sends a reservation request to a warehouse, for each item of the list. The warehouse returns an acknowledgement to the supplier for each item request and, at the end of the item list, the supplier sends a list of the available items to the shop which forwards it to the customer. The customer agreement terminates the process.

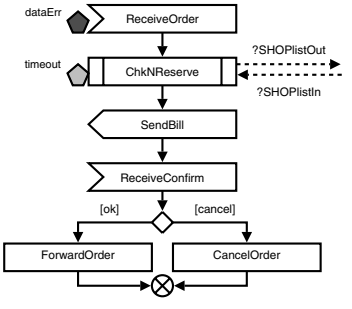

Figure 2. Workflow of the SHOP service

Faults may happen during this process. Figure 2 shows two of them (represented by pentagons), related with the shop. First, when placing his order, the customer may make a data acquisition error, which may result in unexpected items on his reservation list. Then, a timeout may occur when calling the supplier.

We consider that only timeouts may occur on the supplier (Figure 3), when calling the warehouse. On the warehouse, things are more complicated. First, an item may be out of stock, resulting in an incomplete reservation list. Then, an internal error may happen, resulting in a denial of service.

Figure 4 presents two processes that may result in the same observation on the shop, *i.e.* a cancellation of the order due to an incorrect reservation list: (a) a data acquisition error, ordering "eggs and teak" instead of "eggs and tea", for instance, and (b) a stock error happening on the warehouse. Here, we notice that two distinct errors that happen on two distinct services can result in the same local problem, hence the necessity of diagnosing the system globally in order to repair in an adequate way.

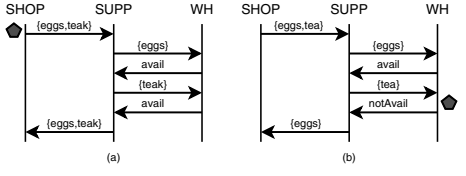

Figure 4. Two scenarii that may result in a cancelled order

4 Extension to distributed environments

Diagnosing distributed systems thanks to chronicles requires to define a modular diagnosis architecture capable of merging diagnoses provided by local chronicle-based diagnosers and to enrich the chronicle formalism with synchronization constraints.

4.1 Architecture

Figure 5 summarizes our chronicle-based approach architecture. This decentralized system is composed of a global diagnoser (or broker) in charge of merging the local diagnoses sent by each service and sending global diagnoses to a repair module. Services are composed of the web service itself, logs generated in real time by the web service, a base of chronicles generated off-line, a local diagnoser that uses the logs to instantiate chronicles from the base.

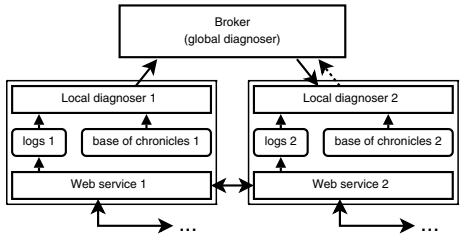

Figure 5. General architecture of a distributed system

4.2 Extension of the formalism of chronicles

As a fault occurring on a service often propagates to other services, we base our approach on the merging of local diagnoses. As a consequence, we enrich the initial formalism of chronicles with synchronization constraints that allow the broker to spot homologous chronicles and merge them.

4.2.1 Synchronization point

Before defining a distributed chronicle, let us firstly define what is a synchronization point.

The **status of a variable** is a Boolean that denotes if the value of a chronicle variable is normal ($\neg err$) or abnormal (err) in a given execution case. A **synchronization variable** is a pair $(?var, status)$ where $?var$ is a (non temporal) chronicle variable and $status$ the status of this variable inside a given chronicle model.

A **synchronization point** is a tuple $(e, \{vars\}, serv_{type})$ where e is an event type, $\{vars\}$ a set of synchronization variables linked with this event type and $serv_{type}$ a type of remote service the local service communicates with. An **instance of a synchronization point** is a synchronization point in which variables are instantiated, and $serv_{type}$ is instantiated as the effective address of the remote service. A synchronization point is **incoming** if it corresponds to a $serv_{remote} \rightarrow serv_{local}$ communication, **outgoing** for the contrary (see example chronicle in section 4.2.2).

Referring to Figure 2 and section 3, here is one of the two synchronization points on the SHOP, which is instantiated as follows, in the execution case of an external error (see Figure 7):
$(ChkNReserve^{+}, \{(?SHOPlistIn, err)\}, SUPP)$.

It expresses the fact that the error is coming from the supplier, through the ?SHOPlistIn variable, which is received by the SHOP at the end of the ChkNReserve activity.

4.2.2 Distributed chronicle

A distributed chronicle is a classical chronicle enriched with a "color" and a "synchronization" part, so that we can merge it with chronicles from adjacent services.

The **color of a chronicle** $\mathcal{K}$ represents the degree of importance of a chronicle and its capacity to trigger a global diagnosis process. Two colors are used: red for faults that may trigger the broker and $green$ for normal behaviors and non critical faults.

Distributed chronicle: a distributed chronicle is a tuple $\mathcal{C}_{\mathcal{D}} = (\mathcal{S}, \mathcal{T}, \mathcal{O}, \mathcal{I}, \mathcal{K})$ where $\mathcal{S}$ is a set of events, $\mathcal{T}$ a graph of constraints between their occurrence dates, $\mathcal{O}$ and $\mathcal{I}$ are respectively two sets of outgoing and incoming synchronization points, and $\mathcal{K}$ is the color of the chronicle.

Let us consider the chronicle describing the external error case. We have the distributed chronicle model $\mathcal{C}_{\mathcal{D}} = (\mathcal{S}, \mathcal{T}, \mathcal{O}, \mathcal{I}, \mathcal{K})$:

$$
\begin{aligned}
\mathcal{S} = \{ \quad & (ReceiveOrder^{-}(), ?t_1), \\
& (ChkNReserve^{-}(?SHOPlistOut), ?t_2), \\
& (ChkNReserve^{+}(?SHOPlistIn), ?t_3), \\
& (SendBill^{+}(), ?t_4), \\
& (ReceiveConfirm^{+}(), ?t_5), \\
& (ForwardOrder^{+}(), ?t_6) \\
\}& \\
\mathcal{T} = & \{?t_1 < ?t_2, ?t_2 < ?t_3, ?t_3 < ?t_4, ?t_4 < ?t_5 < ?t_6\} \\
\mathcal{O} = & \{(ChkNReserve^{-}, \{(?SHOPlistOut, \neg err)\}, SUPP)\} \\
\mathcal{I} = & \{(ChkNReserve^{+}, \{(?SHOPlistIn, err)\}, SUPP)\} \\
\mathcal{K} = & \, red
\end{aligned}
$$

This chronicle triggers the broker, hence its red color. Having defined chronicles for each behavior of each service taking part in the foodshop orchestration, we have the tables shown in Figure 7, in which we only give the synchronization part of the chronicles. red chronicles are written in bold case.

4.3 Algorithms

Our approach consists in merging local chronicles in order to compute a set of candidate global diagnoses. This set of diagnoses is represented by a diagnosis tree as explained in section 4.3.2. There are two steps in the global diagnosis process (Figure 6). In a first step, at "push" time, local diagnosers send recognized chronicles to the broker, which triggers the global diagnosis process. In a second step, at "pull" time, *i.e.* when the global diagnoser needs information, it queries local diagnosers about their chronicles recognized previously or in future. This push-pull mechanism is implemented through a filter as explained below.

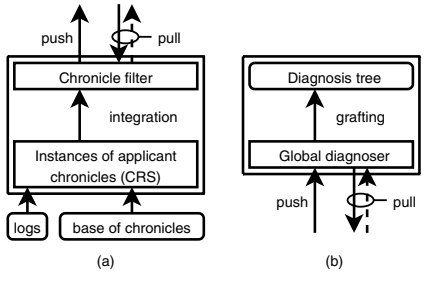

Figure 6. Operation of the (a) local and (b) global diagnosers

4.3.1 Local diagnosis and filtering

The computation of the local diagnosis relies on a CRS module fed by the logs of the web service and sending its recognized chronicles to the global diagnoser (Figure 6.(a)).

In order to avoid sending useless chronicles, a filter $\mathcal{M}$ is set for each running process. In $filter$ mode, only red recognized chronicles are sent to the global diagnoser. Green chronicles are stored in a chronicle buffer $\mathcal{C}_{buf}$. Nevertheless, at "pull" time, the global diagnoser can change $\mathcal{M}$ from $filter$ to $open$, which flushes $\mathcal{C}_{buf}$ in order to provide the global diagnoser with all the available information. In $open$ mode, both red or $green$ newly recognized chronicles will be directly sent to the global diagnoser. Algorithm 1 illustrates this operation.

init: mode $\mathcal{M} := filter$, chronicle set $\mathcal{C}_{buf} := \emptyset$;
on event *chronicle c recognized* **do**
  if $(\mathcal{M} = filter \wedge c.color = red) \vee \mathcal{M} = open$ **then**
    $Broker.push(c)$;
  else
    $\mathcal{C}_{buf} := \mathcal{C}_{buf} \cup \{c\}$;
  end
end
on event *LocalDiagnoser.pull()* **do**
  foreach $c \in \mathcal{C}_{buf}$ **do** $Broker.push(c)$;
  $\mathcal{C}_{buf} := \emptyset$, $\mathcal{M} := open$;
end

Algorithm 1: Local diagnoser management

4.3.2 Global diagnoser algorithm

The global diagnoser algorithm relies on a diagnosis tree $\mathcal{D}_t$ in charge of treasuring all the candidate diagnoses under the shape of partially recognized global chronicles (Figure 6.(b)). Each candidate diagnosis is represented by a path leading to a constraintless node in $\mathcal{D}_t$. The global diagnoser algorithm (Algorithm 2) manages this tree and queries local diagnosers in order to make it grow and complete the pending paths.

The initial diagnosis tree only contains the $emptynode$ which, being constraintless, is compatible with any recognized chronicle. When a recognized chronicle c is sent by a service s to the global diagnoser, two operations are performed. First, $\mathcal{D}_t$ is traversed, trying to combine each node n with c thanks to the status of corresponding variables. In case of a compatibility between n and c, a child node containing c and the synchronization constraints that remain to check is grafted under n in $\mathcal{D}_t$. Then, the global diagnoser changes to $open$ the mode of all the services mentioned in c in order to collect all the information needed for a global diagnosis (Algorithm 2).

init: diagnosis tree $\mathcal{D}_t := emptynode$;
on event *Broker.push(chronicle c)* **do**
  foreach *node* n *of* $\mathcal{D}_t$ **do**
    if c *compatible with* n **then**
      $n.addChild(c)$;
    end
  end
  foreach *service* s *mentioned in* c **do**
    $s.LocalDiagnoser.pull()$;
  end
end

Algorithm 2: Global diagnoser management

When a candidate diagnosis (*i.e.* a constraintless node) is computed in $\mathcal{D}_t$, the broker forwards it to an external repair module and proceeds with the exhibition of other candidate diagnoses.

4.4 Illustration on the example

We consider a customer placing an order on the SHOP, order which is forwarded to the SUPPlier. For each product of the item list, the SUPP calls the WareHouse so as to book the corresponding product. Unfortunately, a product is missing which provokes the recognition of the *WH:stockErr* chronicle, the color of which is $green$, because the WH doesn't consider being out of stock as an error. The broker is not triggered and the execution goes on. But when the SUPP receives the negative answer of the WH, the red chronicle *SUPP:extErr* is recognized and the SUPP "pushes" this chronicle towards the broker, triggering a global diagnosis process while the service execution goes on.

SHOP	*?listOut*	*?listIn*
normal	$\neg err$	$\neg err$
dataErr	err	err
extErr	$\neg err$	err
timeout	$\neg err$	$undef$

SUPP	*?listIn*	*?itemOut*	*?itemIn*	*?listOut*
normal	$\neg err$	$\neg err$	$\neg err$	$\neg err$
fwdErr	err	err	err	err
extErr	$\neg err$	$\neg err$	err	err
timeout	$\neg err$	$\neg err$	$undef$	$undef$

WH	*?itemIn*	*?itemOut*
normal	$\neg err$	$\neg err$
fwdErr	err	err
stockErr	$\neg err$	err
hardErr	$\neg err$	$undef$

Figure 7. Chronicles of the three web services

$\mathcal{D}_t$ only contains the empty root node, at this point. This node is compatible with the constraints of *SUPP:extErr*, listed in Figure7, and a new node containing *SUPP:extErr* and its constraints is grafted under the root node. After this, the broker changes to $open$ the mode of WH, "pulling" the previously recognized *WH:stockErr* chronicle towards it.

$\mathcal{D}_t$ now contains two nodes. *WH:stockErr* is compatible with the empty root node, which results in the grafting of a child node under the root, containing *WH:stockErr* and its constraints. *WH:stockErr* is also compatible with *SUPP:extErr*, as the homologous variables have the same status: $?SUPPitemOut$ and $?WHitemIn$ are $\neg err$, $?SUPPitemIn$ and $?WHitemOut$ are err. This way, a child node is grafted under *SUPP:extErr*, containing *WH:stockErr* and the remaining unchecked constraints (Figure 8).

The "pulling" process goes on, interrogating the SHOP and waiting for its recognized chronicles. At the end of the orchestration execution, $\mathcal{D}_t$ exhibits a single constraintless node, which is then the unique candidate diagnosis: *SHOP:extErr, SUPP:extErr, WH:stockErr*.

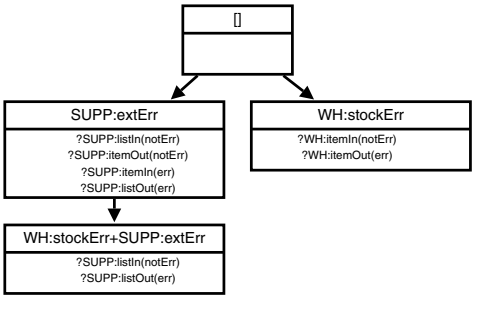

Figure 8. Intermediate diagnosis tree

4.5 A word about complexity

Let us consider the complexity of such an approach. On the local side, the complexity only depends on CRS, which has already been successfully used in large scale systems. Some basic rules about

chronicle writing allow to optimize the use of CRS: PID filtering avoids the recognition of useless cross-process chronicles, delays in chronicle models flush chronicle instances automatically, etc.

On the broker side, the size of the tree only depends on the number of chronicles recognized on each service, hence a need for discriminating and exclusive chronicles. In the worst case, considering all the chronicles are compatible, we demonstrate that the maximum number of nodes in $\mathcal{D}_t$ is

$$n_{max} = \prod_{s \in S} (|\mathcal{C}_s| + 1)$$

with S the set of implied services and $\mathcal{C}_s$ the set of chronicles recognized on s.

5 Related work and discussion

Within the context of the supervision of dynamic systems, many approaches use the formalism of chronicles [2, 7, 9, 6, 15] but few deal with using chronicles in a distributed context.

The way we approach the problem of monitoring dynamic systems from a distributed chronicle-based modeling of the system may be compared with distributed approaches of monitoring discrete-event systems, such as [1, 5, 16, 13]. In each of those works, local diagnoses computed by the different components of the system are synchronized in order to compute a diagnosis taking into account the constraints between components. For instance, the approach of [1] is not so far away from ours, as it fits parts together to build the system diagnosis, like in a puzzle. Those parts, called *tiles*, are labelled by alarms and represent pieces of trajectories. The main difference between the two approaches, apart from the Petri-net-based formalism they use, is that the architecture they adopted is fully distributed, without supervisor. In our decentralized case, a supervisor is in charge of fitting local chronicles together after having synchronized them, which results in the computation of a global chronicle, aiming at taking a repair decision.

Concerning the monitoring of software components and more precisely web services, we can cite among others [11, 3, 4, 12]. The authors of [4] are interested in checking on line the consistency between what a web service should do, called a contract, and its effective execution. Contracts are expressed in a constraint-oriented language, and integrated into the web services files under the shape of annotations. Then, monitors, implemented as web services, observe the behavior of the web services and are capable of detecting timeout problems or functional errors. In [3], the decentralized architecture is close to ours. Each web service is equipped with a local diagnoser generating hypotheses that are consistent with the local model and the observations. A supervisor merges local diagnoses to compute a global one, by propagating hypotheses from a local diagnoser to its neighbors. The main difference is that they rely on a static diagnosis approach: using dependencies between state variables, their approach consists in explaining the alarms that have arisen at a given time. In our case, we monitor the behavior of the components as it evolves.

6 Conclusion

Our contribution in this paper is to propose a *distributed* chronicle-based monitoring and diagnosis approach. Even if it is now recognized that distributed approaches are the only realistic way to monitor large-scale systems, no work exists, to our knowledge, as far as chronicle-based approaches are concerned. We propose a distributed architecture in which a broker service is in charge of synchronizing the local diagnoses computed from chronicles at the component level. We extend the formalism of chronicles and introduce synchronization points that express the synchronization constraints which are checked by the broker according to a push-pull mechanism. We describe the main algorithms and illustrate them on a simplified e-shopping example. A platform has been developed and allows us to make experiments in the framework of the WS-DIAMOND European project, dedicated to the monitoring of software components.

The main perspectives are twofold. The first one is to couple the diagnosis service with a repair service, the goal being to ensure a good QoS, even in case of fault occurrences. The second one is to build acquisition tools to help building the set of local chronicles, starting from workflow descriptions. A first step in this direction can be found in [17].

REFERENCES

[1] A. Aghasaryan, E. Fabre, A. Benveniste, R. Boubour, and C. Jard, 'Fault detection and diagnosis in distributed systems : an approach by partially stochastic petri nets', *Discrete Event Dynamic Systems*, **8**(2), 203–231, (1998).

[2] J. Aguilar, K. Bousson, C. Dousson, M. Ghallab, A. Guasch, R. Milne, C. Nicol, J. Quevedo, and L. Travé-Massuyès, 'Tiger: real-time situation assessment of dynamic systems', Technical report, (1994).

[3] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. Theseider Dupré, 'Cooperative model-based diagnosis of web services', in *Proceedings of DX05, International Workshop on the Principles of Diagnosis*, Pacific Grove, California, (2005).

[4] L. Baresi, C. Ghezzi, and S. Guinea, 'Smart monitors for composed services', in *Proc. of the 2^{nd} Int. Conf. on Service-Oriented Computing (ICSOC'04)*, pp. 193–202, (2004).

[5] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella, 'Diagnosis of a class of distributed discrete-event systems', *IEEE Transactions on systems, man, and cybernetics*, **30**(6), 731–752, (2000).

[6] M.-O. Cordier and C. Dousson, 'Alarm driven monitoring based on chronicles', in *Proc. of Safeprocess'2000*, pp. 286–291, (2000).

[7] M.-O. Cordier, J.-P. Krivine, P. Laborie, and S. Thiébaux, 'Alarm processing and reconfiguration in power distribution systems', in *Proc. of IEA-AIE'98*, pp. 230–240, (1998).

[8] R. Debouk, S. Lafortune, and D. Teneketzis, 'Coordinated decentralized protocols for failure diagnosis of discrete event systems', *Discrete Event Dynamic Systems*, **10**(1-2), 33–86, (2000).

[9] M. Dojat, N. Ramaux, and D. Fontaine, 'Scenario recognition for temporal reasoning in medical domains.', *Artificial Intelligence in Medicine*, **14**(1-2), 139–155, (1998).

[10] C. Dousson, P. Gaborit, and M. Ghallab, 'Situation recognition: representation and algorithms', in *Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI'93)*, pp. 166–172, (1993).

[11] I. Grosclaude, 'Model-based monitoring of component-based software systems', in *Proc. of the 15^{th} Int. Workshop on Principles of Diagnosis (DX'04)*, pp. 51–56, (2004).

[12] A. Lazovik, M. Aiello, and M. Papazoglou, 'Planning and monitoring the execution of web service requests', in *Proc. of the 1^{st} Int. Conf. on Service-Oriented Computing (ICSOC'03)*, volume 2910 of *Lecture Notes in Computer Science*, pp. 335–350, (2003).

[13] Y. Pencolé and M.-O. Cordier, 'A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks', *Artificial Intelligence Journal*, **164**(1-2), 121–170, (2005).

[14] Y. Pencolé, M.-O. Cordier, and L. Rozé, 'Incremental decentralized diagnosis approach for the supervision of a telecommunication network.', in *IEEE Conf. on Decision and Control (CDC'02)*, (2002).

[15] R. Quiniou, M.-O. Cordier, G. Carrault, and F. Wang, 'Application of ilp to cardiac arrhythmia characterization for chronicle recognition', in *ILP'2001*, volume 2157 of *LNAI*, pp. 220–227, (2001).

[16] N. Roos, A. Teije, A. Bos, and C. Witteveen, 'An analysis of multi-agent diagnosis', in *Proc. of the 1^{st} Int. Joint Conf. on Autonomous Agents and MultiAgent Systems (AAMAS'02)*, (2002).

[17] Y. Yan, Y. Pencolé, M.-O. Cordier, and A. Grastien, 'Monitoring web service networks in a model-based approach', in *3^{rd} European Conf. on Web Services (ECOWS)*, (2005).