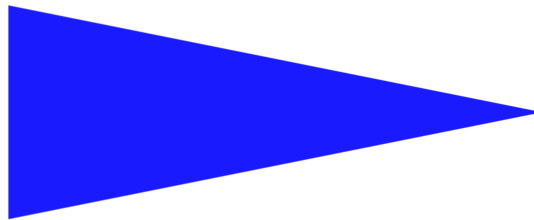
inria-00282294, version 1 - 27 May 2008

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE

N° 1890



CHRONICLES FOR ON-LINE DIAGNOSIS OF DISTRIBUTED SYSTEMS

XAVIER LE GUILLOU , MARIE-ODILE CORDIER ,
SOPHIE ROBIN , LAURENCE ROZÉ

ISSN 1166-8687

CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

inria-00282294, version 1 - 27 May 2008

inria-00282294, version 1 - 27 May 2008

IRISA

Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Chronicles for On-line Diagnosis of Distributed Systems

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

Systèmes cognitifs
Projet DREAM

Publication interne n ° 1890 — Mars 2008 — 47 pages

Abstract: The formalism of chronicles has been proposed a few years ago to monitor and diagnose dynamic physical systems and has been successfully used for real-time applications. A chronicle, expressed by a set of time-constrained events, describes the situations to monitor. Even if efficient algorithms exist to analyze a flow of observed events and to recognize on-the-fly the matching chronicles, it is now well-known that distributed approaches are better suited to monitor actual systems. In this report, we adapt the chronicle-based approach to a distributed context. We propose a decentralized architecture and the corresponding software, implemented under the name of CarDeCRS (*Chronicles Applied to error Recognition in Distributed Environments, through CRS*), which in charge of synchronizing the local diagnoses, computed by chronicle-based local diagnosers, and merging them into a global diagnosis.

This work is motivated by an application that aims at monitoring the behaviour of software components within the WS-Diamond 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 a e-foodshop web service to illustrate our proposal.

Key-words: on-line diagnosis, distributed systems, chronicle recognition

(Résumé : tsvp)

[*] IRISA/UR1, Campus de Beaulieu, Rennes, France, xleguill@irisa.fr
[**] IRISA/UR1, Campus de Beaulieu, Rennes, France, cordier@irisa.fr
[***] IRISA/UR1, Campus de Beaulieu, Rennes, France, robin@irisa.fr
[****] IRISA/INSA, Campus de Beaulieu, Rennes, France, roze@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

inria-00282294, version 1 - 27 May 2008

Chroniques pour le diagnostic en-ligne de systèmes distribués

Résumé : Le formalisme des chroniques a été proposé il y a quelques années dans le but de surveiller et diagnostiquer des systèmes physiques dynamiques en temps réel. Une chronique, exprimée sous la forme d'un ensemble d'évènements temporellement contraints, décrit les situations à surveiller. Même s'il existe des algorithmes efficaces d'analyse de flot d'évènements et de reconnaissance à la volée des chroniques correspondantes, on sait maintenant que les approches distribuées sont plus adaptées à la surveillance de systèmes réels. Dans ce rapport, nous adaptons l'approche à base de chroniques à un contexte distribué. Nous proposons une architecture décentralisée et l'implémentation logicielle correspondante, CarDeCRS (*Chroniques Appliquées à la Reconnaissance Distribuée d'Erreurs, via CRS*), qui est en charge de la synchronisation des diagnostics locaux, lesquels sont calculés par des diagnostiqueurs locaux basés sur des reconnaisseurs de chroniques, et de leur fusion en vue d'obtenir un diagnostic global.

Ce travail est motivé par une application visant à surveiller le comportement de composants logiciels au sein du projet européen WS-Diamond. Dans ce contexte, une requête est envoyée à un web service collaborant avec d'autres services dans le but de fournir une réponse adaptée à la requête de départ. Des fautes peuvent se propager d'un service à l'autre et, afin d'y réagir convenablement, leur diagnostic est un problème crucial. Nous utilisons un exemple de e-commerce simplifié pour illustrer nos propos.

Mots clés : diagnostic en-ligne, systèmes distribués, reconnaissance de chroniques

inria-00282294, version 1 - 27 May 2008

Contents

1 Introduction 5

2 A chronicle recognition approach 6
2.1 Algorithms for matching chronicle variables and input events 6
2.2 Acquisition of chronicles 7
2.3 Distributed Detection using a chronicle based approach 8

3 Representation of distributed chronicles 8
3.1 Example for the formalism 8
3.2 Formalism of chronicles *à la* Dousson 9
3.2.1 Event 9
3.2.2 Chronicle 10
3.3 Extension of the formalism of chronicles 11
3.3.1 Event 11
3.3.2 Synchronization point: 11
3.3.3 Distributed chronicle 12

4 Architecture 14
4.1 General architecture 14
4.2 A local diagnoser service 14
4.2.1 The *CRS* engine 15
4.2.2 The *LocalDiagnoserCRS* module 16
4.2.3 The *LocalDiagnoser* module 16
4.2.4 The *LocalDiagnoserServ* module 16
4.3 The broker service (global diagnoser) 17
4.3.1 The *BrokerServ* module 17
4.3.2 The *Broker* module 18
4.3.3 The *BrokerAlgorithm* module 18
4.3.4 The *ConversionTable* module 18
4.3.5 The *DiagnosisTree* module 19

5 Algorithms 19
5.1 Broker triggered by local diagnosers 20
5.2 Conversion table 20
5.3 Diagnosis tree 22
5.4 General mechanism 24

6 Illustration on an example: the foodshop 24
6.1 Presentation of the example 24
6.1.1 The SHOP service 24
6.1.2 The SUPPlier service 26

6.1.3 The WareHouse service 27
6.2 Chronicles 27
6.2.1 Chronicles of the SHOP 27
6.2.2 Chronicles of the SUPP 29
6.2.3 Chronicles of the WH 30
6.3 Diagnosis process 30
6.3.1 Normal execution 30
6.3.2 Erroneous execution 30

7 Conclusion and prospects 33

A Chronicle files grammars 35
A.1 BNF of the chronicle language 35
A.2 BNF of the distributed chronicle language 39

B Detailed algorithms 40

C Data description 43
C.1 Chronicle 43
C.2 Synchro 43
C.3 ConversionTable 44
C.4 DiagnosisTree 44

inria-00282294, version 1 - 27 May 2008

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 [14], has been widely used and extended [7, 10, 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, 22, 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 [3, 9, 1, 21]. The idea is to consider the system as a set of interacting components instead of a unique 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 [1] 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. A diagnosis platform called CarDeCRS, which stands for *Chronicles Applied to error-Recognition in Distributed Environments, through CRS*, has been developed. It relies on a diagnosis tool developed by France Telecom R&D [15] (CRS, *Chronicle Recognition System*), hence the name.

In section 2, we recall the principles of chronicle recognition approaches. Then, in section 3, we show how to extend the chronicle-based approach to distributed systems. Section 4 details the architecture of each part of the platform. A high-level description of the main algorithms used in CarDeCRS is proposed in section 5. We illustrate those algorithms on a simplified example of an e-foodshop presented in section 6 before concluding in section 7.
.

[1] Web Services' DIAgnosability, MONitoring and Diagnosis

inria-00282294, version 1 - 27 May 2008

inria-00282294, version 1 - 27 May 2008

2 A chronicle recognition approach

Chronicle recognition approaches are challenging techniques for alarm correlation and diagnosis when run-time efficiency is required and/or when time is relevant for aggregating alarms. They rely on a set of patterns, named chronicles, which constitute the base of chronicles. A chronicle is a set of observable events which are time constrained and is characteristic of a situation. In an alarm driven monitoring context, each abnormal situation corresponds to one or more chronicle(s). The events of a chronicle are the alarms and the constraints refer to their occurrence date. The chronicle recognition system is in charge of analyzing the alarm input stream and of identifying, on the fly, any pattern matching with a situation described by a chronicle. The main positive point with these approaches is their high efficiency due to the symptom-to-fault knowledge they rely on. The counterpart is the difficulty of acquiring and updating the base of chronicles. Learning techniques have been proposed to remedy this problem. For instance, given positive and negative examples for each fault, supervised learning techniques provide directly significant chronicles [23, 20].

Most of the studies on chronicle recognition are French [14, 24, 12] and are based on C. Dousson's thesis [11]. A prototype, CRS (which stands for Chronicle Recognition System) has been developed. Information about CRS can be found on the Internet at `http://crs.elibel.tm.fr`.

The chronicle-based approach has been experimented in many domains. The advantages which are usually put forward are twofold. The first one is the high-level formalism allowing to describe, with legibility and modularity, the observable patterns corresponding to interesting behaviours one wants to track and to detect. The second one is the efficiency of the recognition which makes it appropriate for real-time monitoring. The application domains have been the monitoring of gas turbines with the TIGER project [2], the monitoring of telecommunication systems with the AUSTRAL platform developed to support the operators of the French medium voltage system at EDF [19]. It is also used in medical domains. Let us cite the tracking of hepatitis symptoms [17] and the CALICOT system concerned by ECG interpretation and cardiac arrhythmia detection [5]. It has also been used for video surveillance [25] to infer suspect human behaviour from image sequences and inform human security operators.

Let us remark that there are very few work (see 2.3) dealing with a *distributed* chronicle recognition approach.

2.1 Algorithms for matching chronicle variables and input events

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

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

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

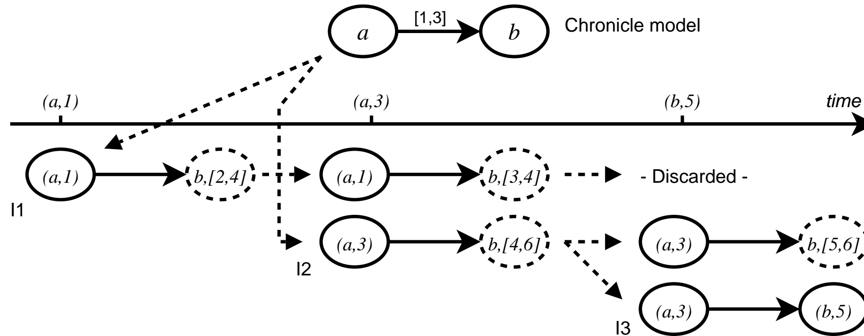


Figure 1: Principle of chronicle recognition

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.

2.2 Acquisition of chronicles

As said before, the main positive point with these approaches is their high efficiency due to the symptom-to-fault knowledge they rely on. Instead of tracking the observable (possibly faulty) behaviour of a system, the chronicle recognition approach recognizes predefined discriminant patterns which are sufficient to ensure the occurrence of a given situation. The counterpart is the difficulty of acquiring and updating the chronicle base. The ways to automatically acquire these patterns are twofold. The first idea is to build them from the fault model of the system to diagnose. For instance, [18] proposes a method based on the idea/concept of Petri net unfolding. Another way is to use learning techniques to remedy this problem. For instance, in [20, 5], examples are first collected for each fault, then supervised learning techniques analyse these positive and negative examples and provide

inria-00282294, version 1 - 27 May 2008

significant chronicles. Another way is based on analyzing alarm logs and extracting the significant patterns by data mining techniques [13]. Another related issue is the completeness of the set of chronicles. It is clear that the more behavioural patterns the base of chronicles specifies, the more diagnosable the system is. This issue is thus directly linked with the diagnosability study and is one of the focus of WP5 (see Deliverable 5.2).

We assume in our work that these chronicles are given. In the case study we acquire these chronicles by hand.

2.3 Distributed Detection using a chronicle based approach

Even if the chronicle based approach seems a good candidate for dealing with distributed systems, few works exist in this domain. Clearly, the concept of "tile" is close to that of "chronicle" and a distributed approach in order to find the most probable trajectory has been proposed in [16]. However, tiles describe exactly the observable behaviour (trace) of a system while a chronicle describes a discriminant pattern for recognizing a (faulty) situation.

The work of [4] proposes a distributed failure detection approach based on chronicles associated to the normal behaviour of the system. The global system is decomposed into a set of monitoring sites. The global constraints describe the synchronization between the local sites. As local clocks are not synchronized, the main problem addressed in this work is the constraint verification with uncertainty on the delay. The authors suppose that the communication delays are bounded and propose a fuzzy quantification with a possibility measure assigned to each global constraint for the constraint verification. The proposed technique is suitable for any system in which the uncertainty on communication delays and the tolerance on the time constraints are similar.

3 Representation of distributed chronicles

The nature of the events we have to process on the web services made us choose to base our local diagnosers on chronicle recognition, the formalism of which we recall here. But as a fault occurring on a service unfortunately often propagates to other services, we enrich the initial formalism with synchronization elements that allow the global diagnoser to spot homologous chronicles and merge them.

3.1 Example for the formalism

As an example for this paper, we consider an orchestration of web services providing e-shopping capabilities to users. The full presentation of this example is proposed in section 6 but in order to introduce the formalism of distributed chronicles, in this section, we only consider one web service out of three. Figure 2 presents the minimalist workflow of the web service that is used for illustrating the formalism.

With this workflow, we can analyze the basic behaviour of the service. A customer orders some items on a web service. This service checks the availability of those items and sends

inria-00282294, version 1 - 27 May 2008

the bill to the customer who can confirm his order if everything is alright or cancel it if there is any problem.

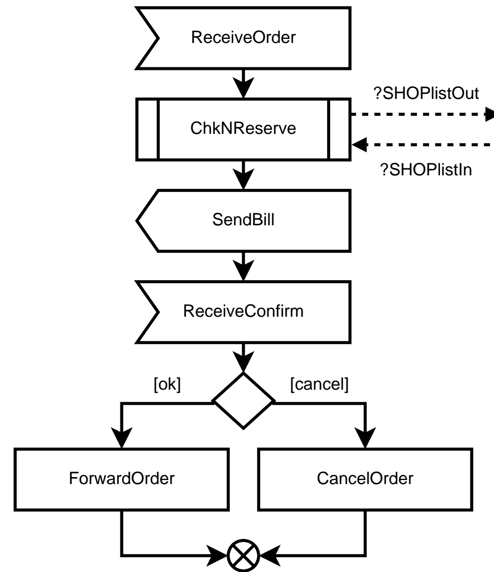


Figure 2: Reduced workflow of the SHOP service

3.2 Formalism of chronicles *à la* Dousson

In this section we present the formalism of the chronicles, as defined by Dousson [14] and define **Event** and **Chronicle**.

3.2.1 Event

An **event type** in a workflow, depending on what is observed in the system, may be of different forms.

1. The name of an activity *act*, *e.g.* *ReceiveOrder*, *ChkNReserve*, etc.
2. The name of an activity augmented with the fact that the activity is starting (namely act^-) or ending (namely act^+), *e.g.* $ReceiveOrder^-$, $ChkNReserve^+$, etc.
3. The name of an activity enriched with some observable parameters, following the form $act(?var_1, \ldots, ?var_n)$, *e.g.* $ChkNReserve^-(?SHOPlistOut)$, $ChkNReserve^+(?SHOPlistIn)$, etc. Note that we use the question mark ? to denote variables, and then $?var$ means that *var* is a variable.

inria-00282294, version 1 - 27 May 2008

4. A combination of 2 and 3, namely:
   - $act^-(?var_1, \ldots, ?var_n)$ which means *"The activity act is starting with the parameters $?var_1, \ldots ?var_n$"*;
   - $act^+(?var_1, \ldots, ?var_n)$ which means *"The activity act is ending with the return values $?var_1, \ldots ?var_n$"*.

An **event** is a pair $(e, ?t)$ where e is an **event type** and $?t$ the occurrence date of the event, *e.g.* $(ChkNReserve^-(?SHOPlistOut), ?t)$.
An **event instance** is an event, which variables and date have been instantiated, *e.g.* $(ChkNReserve^-(?SHOPlistOut = \{biscuit, tea\}), ?t = 2)$.
An **event log** $\mathcal{L}$ is a temporally ordered list of event instances.
Example: $(ReceiveOrder, ?t_1 = 1)$
$(ChkNReserve^-(?SHOPlistOut = \{biscuits, tea\}), ?t_2 = 2)$
$(ChkNReserve^+(?SHOPlistOut = \{biscuits, tea\}), ?t_3 = 5)$
...

3.2.2 Chronicle

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.
Example: $\mathcal{C} = (\mathcal{S}, \mathcal{T})$ with
$\mathcal{S} = \{$ $(ReceiveOrder, ?t_1)$,
$(ChkNReserve^-(?SHOPlistOut), ?t_2)$,
$(ChkNReserve^+(?SHOPlistIn), ?t_3)$,
$(SendBill, ?t_4)$,
$(ReceiveConfirm, ?t_5)$,
$(ForwardOrder, ?t_6)$
$\}$ and
$\mathcal{T} = \{?t_1 < ?t_2, ?t_2 < ?t_3, ?t_3 < ?t_4, ?t_4 < ?t_5, ?t_5 < ?t_6\}$
A **partially instantiated chronicle** c from a model $\mathcal{C}$ is a set of event instances of $\mathcal{C}$ consistent with the temporal constraints of $\mathcal{T}$.
Example:
$c = \{$ $(ReceiveOrder, ?t_1 = 1)$,
$(ChkNReserve^-(?SHOPlistOut = \{biscuits, tea\}), ?t_2 = 2)$
$(ChkNReserve^+(?SHOPlistOut = \{biscuits, tea\}), ?t_3 = 5)$
$\}$
A **fully instantiated chronicle** c from a model $\mathcal{C}$ is a set containing an instance of each event of $\mathcal{C}$ and which is consistent with the temporal constraints of $\mathcal{T}$.

inria-00282294, version 1 - 27 May 2008

Example:

$c = \{ \quad (ReceiveOrder, ?t_1 = 1),$
$\quad (ChkNReserve^{-}(?SHOPlistOut = \{biscuits, tea\}), ?t_2 = 2)$
$\quad (ChkNReserve^{+}(?SHOPlistOut = \{biscuits, tea\}), ?t_3 = 5)$
$\quad (SendBill, ?t_4 = 6),$
$\quad (ReceiveConfirm, ?t_5 = 8),$
$\quad (ForwardOrder, ?t_6 = 10)$
$\}$

3.3 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.

3.3.1 Event

We first enrich the notion of event, in order to allow some events of a chronicle to trigger a global diagnosis stage without having to wait for the full recognition of this chronicle.

A **brokering event** is an event triggering the global diagnoser via a push call before a full recognition of the chronicle . This concept has not been implemented yet, but will come with the implementation of repair plans into CarDeCRS.

An **event of a distributed chronicle** is a tuple (E, t, b) representing an event enriched with a Boolean b denoting if the event is a **brokering event** (bro) or not $(\neg bro)$.

3.3.2 Synchronization point:

Errors occurring on remote services may propagate to other services *via* parameters or return values of remote procedure calls. It is thus important to check whether a local behaviour, proposed as diagnosis by a local diagnoser, may correspond to what has been observed by the other services. It is the role of synchronization elements to express these constraints.

Then, we extend the formalism of chronicles, adding synchronization elements to the formalism of chronicles.

The **set of chronicle variables** of $\mathcal{C}$ is the set of non temporal variables that belong to the events of this chronicle.

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.
Example: $(?SHOPlistOut, \neg err)$

inria-00282294, version 1 - 27 May 2008

A **synchronization point** is a tuple $(event, \{vars_c\}, serv_{type})$ where $event$ is an event, $\{vars_c\}$ a set of synchronization variables linked with this event and $serv_{type}$ a type of remote service the local service communicates with.
Example: $((ChkNReserve^-(?SHOPlistOut), ?t_2, \neg bro),$
$\{(?SHOPlistOut \neg err)\}, supplier)$

An **instance of synchronization point** is a synchronization point in which $event$ is instantiated, and $serv_{type}$ is instantiated as $serv_{id}$, *i.e.* the effective address of the remote service.
Example: $((ChkNReserve^-(?SHOPlistOut = \{biscuits, tea\}), ?t_2 = 2, \neg bro),$
$\{(?SHOPlistOut, \neg err)\}, supplier = www.mysupplier.com)$

An **incoming/outgoing synchronization point** is an oriented synchronization point. Incoming stands for a $serv_{remote} \rightarrow serv_{local}$ communication, outgoing for the contrary. By extension, an **incoming/outgoing infection point** is an oriented synchronization point having at least one erroneous (err) variable in $\{vars_c\}$ (see $\mathcal{I}_2$ in the SHOP::ExternalOrder chronicle below).

3.3.3 Distributed chronicle

Finally, we introduce the concept of distributed chronicle.

A **distributed chronicle** is a classical chronicle enriched with a "synchronization" part, so that we could merge it with chronicles from adjacent services. A distributed chronicle model $\mathcal{C}_\mathcal{D}$ is a tuple $(\mathcal{S}, \mathcal{T}, \mathcal{O}, \mathcal{I})$ where $\mathcal{S}$ is a set of events, $\mathcal{T}$ a set of constraints between their occurrence dates, and $\mathcal{O}$ and $\mathcal{I}$ are respectively two sets of outgoing and incoming synchronization points.

Examples:
Example1: The SHOP::NormalOrder chronicle.

Considering the case of a client who manages to reserve all his items (Figure 3.a), we get the *SHOP::NormalOrder* distributed chronicle described as $\mathcal{C}_{\mathcal{D}1} = (\mathcal{S}_1, \mathcal{T}_1, \mathcal{O}_1, \mathcal{I}_1)$, with

$\mathcal{S}_1 = \{$ $(ReceiveOrder, ?t_1, \neg bro),$
$(ChkNReserve^-(?SHOPlistOut), ?t_2, \neg bro),$
$(ChkNReserve^+(?SHOPlistIn), ?t_3, \neg bro),$
$(SendBill, ?t_4, \neg bro),$
$(ReceiveConfirm, ?t_5, \neg bro),$
$(ForwardOrder, ?t_6, \neg bro)$
$\}$
$\mathcal{T}_1 = \{?t_1 < ?t_2, ?t_2 < ?t_3, ?t_3 < ?t_4, ?t_4 < ?t_5, ?t_5 < ?t_6\}$
$\mathcal{O}_1 = \{((ChkNReserve^-(?SHOPlistOut), ?t_2, \neg bro),$
$\{(?SHOPlistOut, \neg err)\}, supplier)\}$
$\mathcal{I}_1 = \{((ChkNReserve^+(?SHOPlistIn), ?t_3, \neg bro),$
$\{(?SHOPlistIn, \neg err)\}, supplier)\}$

Example2: The SHOP::ExternalOrder chronicle.

inria-00282294, version 1 - 27 May 2008

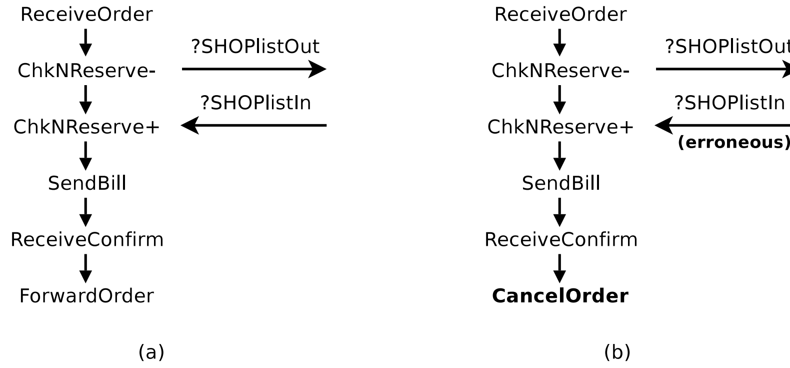


Figure 3: (a) Normal chronicle of the Shop and (b) ExternalOrder chronicle of the Shop

Considering the case where some items are out of stock (Figure 3.b), we get the *SHOP::ExternalError* distributed chronicle described as $\mathcal{C}_{\mathcal{D}2} = (\mathcal{S}_2, \mathcal{T}_2, \mathcal{O}_2, \mathcal{I}_2)$, with

$$\begin{aligned}\mathcal{S}_2 = \{ \quad & (ReceiveOrder, ?t_1, \neg bro),\\ & (ChkNReserve^-(?SHOPlistOut), ?t_2, \neg bro),\\ & (ChkNReserve^+(?SHOPlistIn), ?t_3, \neg bro),\\ & (SendBill, ?t_4, \neg bro),\\ & (ReceiveConfirm, ?t_5, \neg bro),\\ & (\boldsymbol{CancelOrder}, ?t_6, bro)\\ \} & \end{aligned}$$

$$\mathcal{T}_2 = \{?t_1 <?t_2, ?t_2 <?t_3, ?t_3 <?t_4, ?t_4 <?t_5, ?t_5 <?t_6\}$$

$$\begin{aligned}\mathcal{O}_2 = \{ & ((ChkNReserve^-(?SHOPlistOut), ?t_2, \neg bro),\\ & \{(?SHOPlistOut, \neg err)\}, supplier)\}\end{aligned}$$

$$\begin{aligned}\mathcal{I}_2 = \{ & ((ChkNReserve^+(?SHOPlistIn), ?t_3, \neg bro),\\ & \{(?SHOPlistIn, err)\}, supplier)\}\end{aligned}$$

Colored distributed chronicle: by extension, a coloured distributed chronicle $\mathcal{C}_{\mathcal{DC}}$ is a tuple $(\mathcal{S}, \mathcal{T}, \mathcal{O}, \mathcal{I}, \mathcal{K})$ where $\mathcal{K}$ is a "colour". $\mathcal{K}$ represents the degree of importance of the chronicle. Using two colours (green and red), we have the following strategy.

- A "normal" execution is represented by a green chronicle which, even fully recognized, does not trigger the global diagnoser. The chronicle on Figure 3.a could be a green one. Those chronicles are used by the broker during stages of information gleaning, when there is a need to account for the normal execution of one service.

- An "abnormal" execution is represented by a red chronicle which has to be fully recognized to trigger the broker. The chronicle on Figure 3.b could be a red one.

inria-00282294, version 1 - 27 May 2008

This strategy can be made more complex, allowing a set of colours and partially recognized chronicles triggering the broker before complete recognition in order to get the diagnosis as early as possible.

4 Architecture

4.1 General architecture

The main idea of decentralized diagnosis is to perform, in a single system, both local and global diagnoses. Local diagnosers, each one including a CRS module, are used in order to monitor local erroneous behaviours. But when local diagnoses are ambiguous, a global diagnoser, also called *broker*, performs a global diagnosis process, in order to discard the local diagnoses which cannot be synchronized, merges the remaining ones and, if needed, takes a global decision.

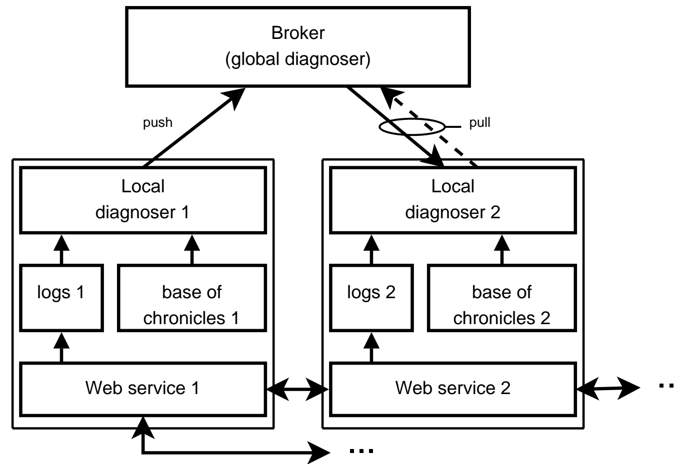


Figure 4: General architecture of CarDeCRS

Figure 4 illustrates this general operating. On this figure, we observe that Web Services, which are communicating together, generate execution logs. Those logs instantiate predefined chronicle models which, once recognized, may trigger a global diagnosis process, according to the diagnosis strategy applied by the local diagnoser and the broker.

4.2 A local diagnoser service

Figure 5 shows the detailed architecture of a local diagnoser service. Each local diagnoser service is composed of several modules:

inria-00282294, version 1 - 27 May 2008

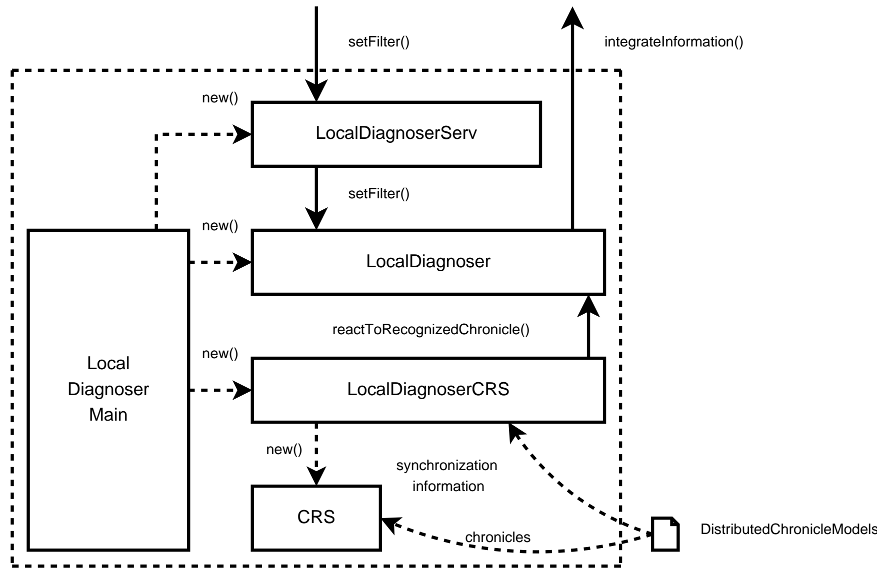


Figure 5: Detailed architecture of a local diagnoser service

- a standard *CRS* engine (by FT R&D);
- an API providing high-level access to CRS, called *LocalDiagnoserCRS*;
- the *LocalDiagnoser* itself;
- a Java-RMI server, called *LocalDiagnoserServ*.

In the following, we describe in detail each module mentioned above.

4.2.1 The *CRS* engine

The chronicle recognition system used in CarDeCRS is a standard CRS engine (for more information, see `http://crs.elibel.tm.fr`).

This engine uses two external files:

- a description file of chronicle models (`.rg`) which contains the exhaustive list and descriptions of the chronicles that can be recognized by the system (it corresponds to the *base of chronicles* mentioned in Figure 4);
- an event file (`.evts`) which contains the logs generated by the web server (it corresponds to the set of *logs* mentioned in Figure 4). Of course, this file can be written on-line.

inria-00282294, version 1 - 27 May 2008

4.2.2 The *LocalDiagnoserCRS* module

The *LocalDiagnoserCRS* module is both an API and an extension for *CRS*. Indeed, our approach is based on distributed chronicle models, that is to say chronicle models enriched with information:

- synchronization information, representing the different variables exchanged between services;
- "colour" information, representing the degree of importance of the distributed chronicle, in our case red for critical chronicles and green for non critical ones.

When a distributed chronicle model is fully instantiated, the *LocalDiagnoserCRS* notifies the *LocalDiagnoser* via a `reactToRecognizedChronicle()` call. The continuation depends on the diagnosis policy applied by the *LocalDiagnoser* module.

This module uses one external file:

- a description file of distributed chronicle models (`.dst`) which contains the synchronization descriptions of the chronicles that can be recognized by the system, and the colour of each distributed chronicle model.

4.2.3 The *LocalDiagnoser* module

The *LocalDiagnoser* module is the heart of the local diagnoser. Its functioning is based on a diagnosis policy, symbolized by a "filter mode" set for each running process:

- in *filter* mode, only critical (red) chronicles recognized by the *LocalDiagnoserCRS* trigger the broker, via an `integrateInformation()` call, and green chronicles are stored in a chronicle buffer which can be flushed by a `setFilter()` call performed by the broker;
- in *open* mode, each recognized chronicle (either red or green) performs an `integrateInformation()` call.

Only the broker may change the filter mode on-line, via a `setFilter()` call. At present, the diagnosis policy is quite simple, but it would be made more complex in a next future.

4.2.4 The *LocalDiagnoserServ* module

The *LocalDiagnoserServ* is an RMI-based module which acts as a gateway between the broker and the local diagnoser. It allows the broker to change on-line the mode of the filter used by the local diagnoser, via a `setFilter()` call.

Other methods may be implemented on this remote method interface, depending on the needs of our future policy algorithms.

inria-00282294, version 1 - 27 May 2008

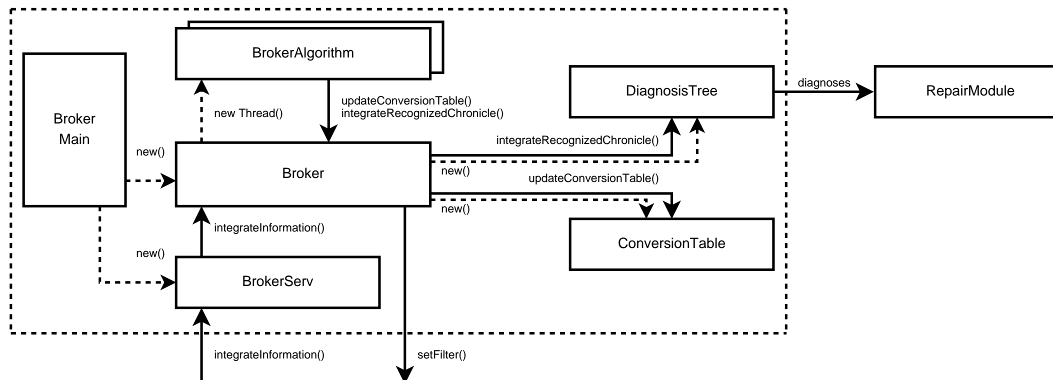


Figure 6: Detailed architecture of the broker service

4.3 The broker service (global diagnoser)

Figure 6 shows the detailed architecture of the broker service (broker is the name given to the service in charge of the global diagnosis). Just like the local diagnoser, it is composed of several modules:

- a Java-RMI server, called *BrokerServ*;
- the *Broker* itself;
- a *BrokerAlgorithm*, run on another thread;
- a *ConversionTable* that manages links between services;
- a *DiagnosisTree* that elaborates the global diagnosis.

Each module mentioned above is presented in detail in the following.

4.3.1 The *BrokerServ* module

As each local diagnoser owns an RMI-based module dedicated to communications with the broker, the broker owns an RMI-based module dedicated to communications with all the local diagnosers. This gateway allows the local diagnosers to notify the broker of a chronicle recognition, via an `integrateInformation()` call.

Other methods may be implemented, depending on the needs of both our future policy algorithms and our management of choreographies.

inria-00282294, version 1 - 27 May 2008

4.3.2 The *Broker* module

The *Broker* module acts as a sore point between all the local diagnosers. It receives all the diagnosis requests forwarded by the *BrokerServ* and runs a *BrokerAlgorithm* on a new thread for each diagnosis request.

During the execution of the *BrokerAlgorithm*, it can be asked to enrich the global *ConversionTable* and to make the *DiagnosisTree* grow, via `updateConversionTable()` and `integrateRecognizedChronicle()` calls, requested by the *BrokerAlgorithms*.

This module is also responsible for two other tasks. First, during the global diagnosis process, it is responsible for interrogating local diagnosers via `setFilter()` calls. Then, at the end of this process, it is responsible for transmitting the computed diagnosis to a repair module (not implemented yet).

4.3.3 The *BrokerAlgorithm* module

The *BrokerAlgorithm* is in charge of processing the information sent by a local diagnoser, with two goals in mind.

- First, it has to forward the distributed chronicle models sent by the local diagnoser to the *ConversionTable* module, in order to be able to link synchronization variables between services.
  This is done via an `updateConversionTable()` call.
- Then, it has to forward the synchronization information sent by the local diagnoser to the *DiagnosisTree* module, in order to merge the current broker knowledge with the recently received one.
  This is done via an `integrateRecognizedChronicle()` call.

4.3.4 The *ConversionTable* module

The role of the *ConversionTable* module is to build a table linking synchronization variables between services. In other words, if outgoing variable x on $servA$ corresponds with incoming variable y on $servB$, this module has to add two lines to the table (one suffices but we use two for implementation reasons):

- $(servA, servB, x, y)$;
- $(servB, servA, y, x)$.

This is done in two steps. First, when $servA$ triggers the *Broker*, it sends, with each recognized chronicle, an ordered list of variables that are exchanged between $servA$ and its remote partners. If $servA$ only exchanges one variable x with a remote service $servB$, then the *Broker* builds the following incomplete conversion table:

- $(servA, servB, x, ?)$;

inria-00282294, version 1 - 27 May 2008

- $(servB, servA, ?, x)$.

In order to replace the question marks with real variable names, the *Broker* interrogates $servB$ that sends it its own recognized chronicles with their ordered variables, which leads to, if $servB$ only communicated with $servA$:

- $(servA, servB, x, y)$;
- $(servB, servA, y, x)$.

This table is used by the *Broker* module in order to merge chronicles from different services.

4.3.5 The *DiagnosisTree* module

The *DiagnosisTree* module has to combine the knowledge (i.e. the recognized chronicles) sent by all the local diagnosers, taking into account constraints expressed on synchronization variables. Each node of the tree is composed of:

- a set of chronicles that do not violate synchronization constraints;
- a set of constraints that remain to be checked.

When a new chronicle is sent to the diagnosis tree, it makes all the compatible branches grow, deleting the newly checked constraints from the "set of constraints that remain to be checked". The remote services used by this new chronicle are then put into a set referencing all the services that may be questioned by the *Broker* via `setFilter()` calls.

5 Algorithms

This section gives a high level representation of the main algorithms implemented in CarDeCRS. Some symbols are common to several algorithms.
On each *LocalDiagnoser*:

- $\mathcal{F}$ represents the "filter mode" of the current local diagnoser;
- $\mathcal{G}_c$, is a temporary buffer of recognized green chronicles which is flushed when $\mathcal{F}$ is set to *open*.

On the *Broker*:

- $\mathcal{T}_c$, is the conversion table used to determine the corresponding variables between services;
- $\mathcal{D}_t$, is the diagnosis tree built on-line, according to the received chronicles.

inria-00282294, version 1 - 27 May 2008

inria-00282294, version 1 - 27 May 2008

5.1 Broker triggered by local diagnosers

A *LocalDiagnoser* module triggers the broker when a chronicle that passes through the filter is recognized. This means that if the filter is set to *filter*, only red chronicles are sent and if the filter is set to *open*, both red and green chronicles are sent. Algorithm 1 presents this mechanism.

```
initialization: filter F = filter, chronicle set G_c = ∅;
on event chronicle c recognized do
    if F == filter and c.colour == red then
        call Broker.integrateInformation(c);
    else if F == open then
        call Broker.integrateInformation(c);
    else
        G_c = G_c ∪ {c};
    end
end
```

Algorithm 1: Broker triggering on a local diagnoser

*LocalDiagnoser*s can also send information to the broker "on demand", during a diagnosis process. In this case, the broker sets the *LocalDiagnoser*'s filter to *open* and, as a consequence, the previously unsent recognized green chronicles (see Algorithm 2) are sent to the broker.

```
on event LocalDiagnoser.setFilter(mode k) call do
    F = k;
    if F == open then
        foreach c ∈ G_c do
            call Broker.integrateInformation(c);
        end
        G_c = ∅
    end
end
```

Algorithm 2: Filter management on a local diagnoser (setFilter call)

5.2 Conversion table

In order to establish the links between the variables of the various services, a conversion table has to be built. Indeed, the variables of a source service and the corresponding variables on a remote service are not similarly named. Therefore, to be able to see if variables and states shared by chronicles match or not, the broker needs to know what are the correspondences. Without this conversion table, establishing the diagnosis is impossible.

Such a conversion table can be built with no *a priori* knowledge on the homologous variables if either a methodology for writing distributed chronicles has been pre-established

or web services generate logs in which the correspondence between local and remote variables is written.

Here is an example of an hypothetical conversion table:

	source service	remote service	source variables	remote variables
1	$servA$	$servB$	$[x, y]$	$[z, t]$
2	$servB$	$servA$	$[z, t]$	$[x, y]$
3	$servA$	$servC$	$[i, j]$	$[\,]$
4	$servC$	$servA$	$[\,]$	$[i, j]$
...	...	...	...	...
...	...	...	...	...

In this example, the broker has received information from both $servA$ and $servB$ but not from $servC$. So, the broker has been able to fill the first two lines completely, but not the two following ones, for which some pieces of information are still missing.

Then, if the broker receives information from $servA$ about variable y, e.g. a chronicle saying that y is erroneous, it now knows that this chronicle can only merge with chronicles of $servB$ with t having the same erroneous status.

Algorithm 3 presents the global construction algorithm of the conversion table.

```
Data: a recognized chronicle c
initialization: s_s = c.sourceService;
foreach remote service s_r of c do
  | if T_c has (s_s, s_r) entries then
  |   | if those entries are incomplete then
  |   |   | update the table;
  |   | end
  | else
  |   | create partial (s_s, s_r) entries in T_c;
  | end
end
```

Algorithm 3: Construction of the conversion table, main algorithm

In this algorithm, "incomplete information" means that we already got information from a service but not from the corresponding one. In the correspondence table given in example, this case happens when the tuple $(servC, servA, \{k, l\})$ is received, which causes the update of lines 3 and 4 in the table. $\{k, l\}$ are the variables of $servC$ homologous with the variables $\{i, j\}$ of $servA$.

inria-00282294, version 1 - 27 May 2008

inria-00282294, version 1 - 27 May 2008

5.3 Diagnosis tree

Establishing a diagnosis requires the use of a diagnosis tree that merges the information sent to the broker by local diagnosers (on their own request or on the broker request). Figure 7 shows an example of a possible diagnosis tree.

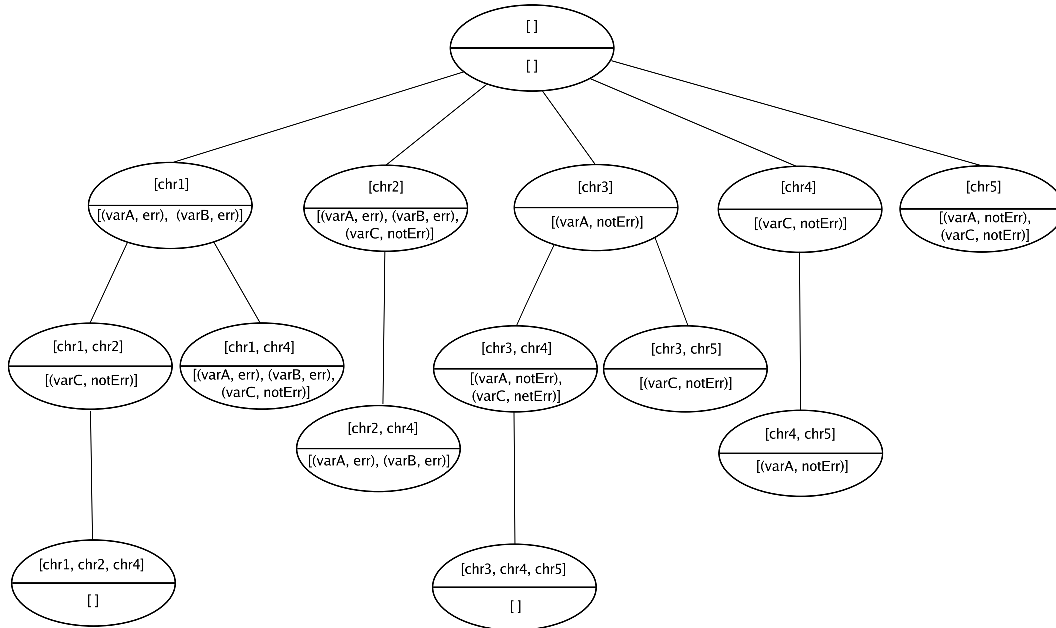


Figure 7: Diagnosis tree example

As shown in this example tree, each node is composed of two sets of information:

- a list of chronicles which contains the chronicles that have led to the current diagnosis step;
- a list of $(variable, status)$ pairs which contains the variables that still have to be checked in order to lead to a diagnosis.

The list of chronicles contains as many elements as the depth of the node in the tree. Indeed, each node has the chronicle list of its parent node plus the newly recognized chronicle.

The cardinality of the list of variables can either increase or decrease. It increases when the newly recognized chronicle does not eliminate already listed constraints. On the contrary, it decreases when already listed constraints are eliminated by the new chronicle (and if this new chronicle does not add too many constraints on variables). Getting this list empty

means that all the listed chronicles have led to the diagnosis by checking that the status of all their variables were compatible: the diagnosis if relevant.

In this example, we have two possible diagnoses, *i.e.* two branches leading to an empty set of constraints:

- $\{chr1, chr2, chr4\}$
- $\{chr3, chr4, chr5\}$

This means that the first possible explanation to the fault is that $chr1$, $chr2$ and $chr4$ occurred, and the second one is that $chr3$, $chr4$ and $chr5$ have occurred[3].

```
Data: a recognized chronicle c
initialization: L = {D_t.rootNode};
foreach node n of L do
    if c is compatible with n then
        L = L ∪ n.children;
        create a child node for n in D_t;
    else
        do nothing;
        /* if n is not compatible, its children won't be */
    end
end
```

Algorithm 4: Diagnosis tree construction, main algorithm

The main algorithm, given in Algorithm 4, has been simplified in order to clarify it. Algorithm 5 explains the compatibility between a node and a chronicle. Algorithms 8 and 9 (see Appendix B) presents in detail what has been abstracted in the previous ones.

```
Data: a node n, a recognized chronicle c
Result: True if the data are compatible, false else.
foreach synchro s of c do
    if s is not compatible with any synchro constraint of n then
        return false;
    else
        return
    end
    true;
end
```

Algorithm 5: Diagnosis tree construction, node and chronicle compatibility

[3]This is more talkative with real chronicle names like “nullCompterStock” or “DataAcquisitionError”, etc.

inria-00282294, version 1 - 27 May 2008

inria-00282294, version 1 - 27 May 2008

5.4 General mechanism

The general mechanism of the system brings into play the *LocalDiagnoser* and the *Broker* modules. As the *LocalDiagnoser* algorithm has already been explained, we concentrate on the broker algorithm, in Algorithm 6.

on event *Broker.integrateInformation(chronicle c)* **do**
  update $\mathcal{T}_c$ with c;
  update $\mathcal{D}_t$ with c;
  foreach *service s mentioned in c* **do**
    call (LocalDiagnoser)s.setFilter($green$);
  end
end

Algorithm 6: Broker mechanism (integrateInformation call)

6 Illustration on an example: the foodshop

6.1 Presentation of the example

In this example, we consider an orchestration of web services that provide e-shopping capabilities to users. Three web services take part in this orchestration:

- the SHOP service, which is the web interface used by the customer;
- the SUPPlier service, which processes the orders;
- the WareHouse service, which manages the goods.

When a customer wants to place an order, he selects items on the SHOP. This list of items is transferred to the SUPP which sends a reservation request to the WH, for each item of the list.

The WH returns an acknowledgement to the SUPP for each item request and, at the end of the item list, the SUPP sends a list of the available items to the SHOP which forwards it to the customer. The customer agreement terminates the process.

A detailed "per slice" presentation of the services and their chronicles is given below.

6.1.1 The SHOP service

A high-level workflow of the SHOP service is given in Figure 8. From this figure, we can analyse the whole execution process.

First, the service receives an itemlist from the customer. As the customer is not considered as a workflow execution, there is no "data exchange" represented between him and the SHOP.

The service stores the order and forwards it to the SUPP service, in the *?SHOPlistOut* variable transmitted via the `Supplier::CNR()` call. As a return to this call, the SHOP receives the *?SHOPlistIn* variable, containing a list of all the available items.

The service computes the cost of those available goods, and sends a summary of the order to the customer, in order to request his agreement. If the customer agrees, the order proceeds. If not, the order is cancelled. The ensuing part of the process is not represented in this example.

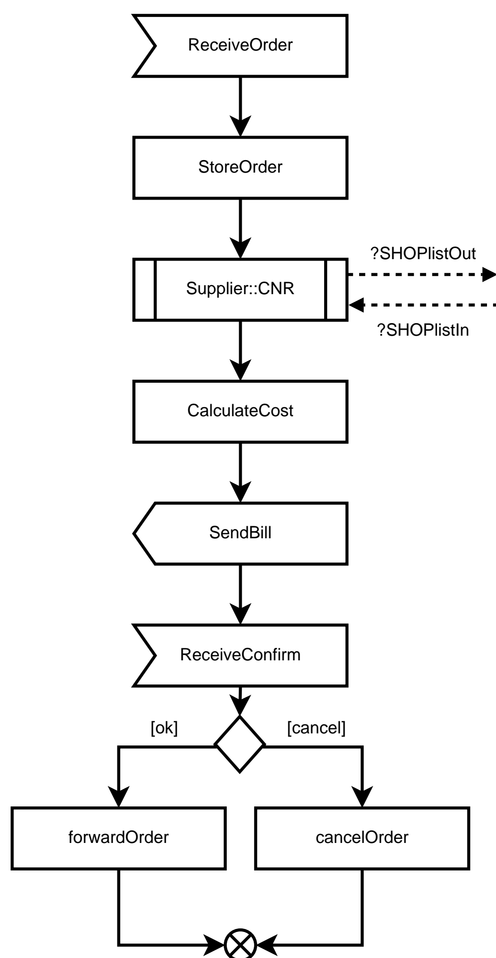


Figure 8: Workflow of the SHOP service

inria-00282294, version 1 - 27 May 2008

6.1.2 The SUPPlier service

A high-level workflow of the SUPP service is given in Figure 9. This figure represents the inner working of the SUPPlier which acts as an interface between the online SHOP and the WareHouse.

As soon as it receives the customer order in the *?SUPPlistIn* variable, the SUPP iterates on this list in the aim of booking each item separately via `WHouse::CNR()` calls. To each outgoing *?SUPPitemOut* request corresponds an incoming *?SUPPitemIn* answer, specifying if the item was successfully booked or not.

If the booking phase was successful, the "to-return" item list is updated and the process goes on. If not, the process goes on, keeping the item list unchanged. At the end of the process, the item list containing only the booked items is returned in the *?SUPPlistOut* variable.

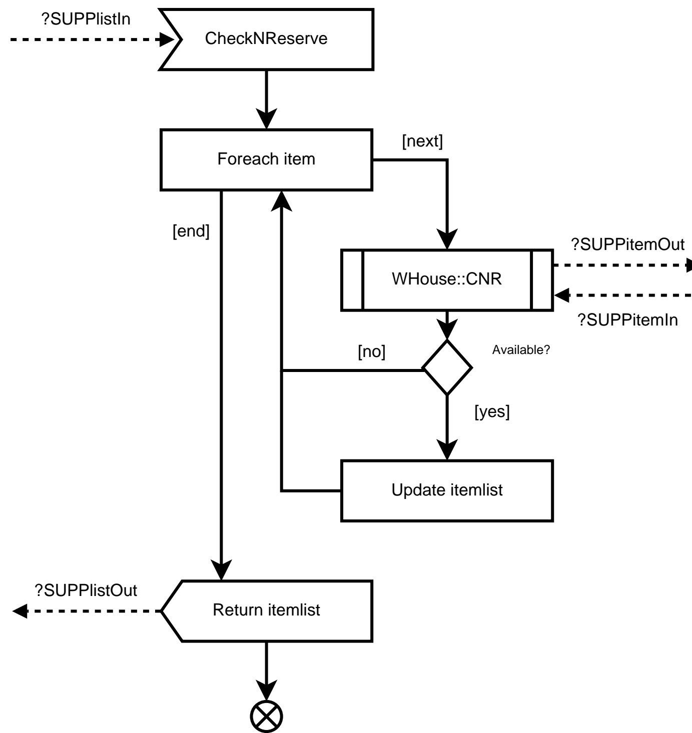


Figure 9: Workflow of the SUPPlier service

inria-00282294, version 1 - 27 May 2008

6.1.3 The WareHouse service

A high-level workflow of the WH service is given in Figure 10. This figure represents the simple booking protocol used by the warehouse.

The WH receives item requests in the *?WHitemIn* variable. If the requested item is available, the service performs the booking operation, updates its stock and replies positively. If it is not available, the service directly replies negatively.

Both positive and negative replies are returned in the *?WHitemOut* variable.

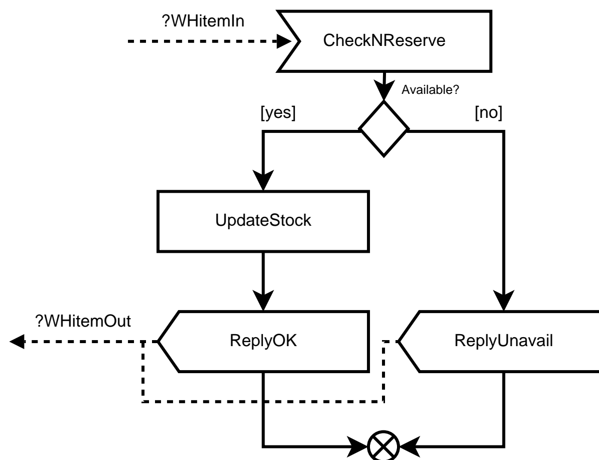


Figure 10: Workflow of the WareHouse service

6.2 Chronicles

In order to optimize the clarity of this example, we chose to consider only single faults in this document. That means only one fault may happen in the orchestration and no other fault can compensate it. Of course, CarDeCRS can deal with multiple and compensated faults. In those cases, the quality of the resulting diagnosis only depends on the quality of the written chronicles.

The chronicles given below use the CRS formalism, which is different from the formalism given in section 3.

6.2.1 Chronicles of the SHOP

Three execution cases may be foreseen on the SHOP. One of them ends with the *forwardOrder* activity, two of them with the *cancelOrder* activity.

inria-00282294, version 1 - 27 May 2008

The first case to consider is the normal execution case: the customer orders a list of items and receives a list with all the items available. As everything is alright, he confirms his order. Using the grammar described in Appendix A that relies on [8], we can write the first chronicle. First, the local part, describing the flow of events that constitute the chronicle:

```
chronicle shopNormalOrder[?PID,?shop,?supp]() {
  event (ReceiveOrder[?PID],t0)
  event (StoreOrder[?PID],t1)
  event (Supplier::CheckNReserveCall[?PID,?shop,?supp],t2)
  event (Supplier::CheckNReserveRet[?PID,?shop,?supp],t3)
  event (CalculateCost[?PID],t4)
  event (SendBill[?PID],t5)
  event (ReceiveConfirm[?PID],t6)
  event (ForwardOrder[?PID],t7)
  t7>t6>t5>t4>t3>t2>t1>t0
}
```

then, the communication part, that describes the status of the variables exchanged between the SHOP and its partner, the SUPP:

```
shopNormalOrder[?PID,?shop,?supp] = { normal, GREEN,
  (Supplier::CheckNReserveCall,((?SHOPlistOut,notErr)),out,supp),
  (Supplier::CheckNReserveRet,((?SHOPlistIn,notErr)),in,supp) }
```

As a reminder, in this "communication part", the `GREEN` colour means that this chronicle shall not be sent to the broker, unless the filter of the local diagnoser is set to *open*.

The second case to consider is linked with a data acquisition error: the customer does a mistake when placing his order, which results in either the billing of a wrong item or the billing of a wrong number of a certain item. Both lead to cancellation of the order:

```
chronicle shopDataAcquisitionError[?PID,?shop,?supp]() {
  //same as shopNormalOrder
  ...
  event (ReceiveConfirm[?PID],t6)
  event (CancelOrder[?PID],t7)
  t7>t6>t5>t4>t3>t2>t1>t0
}
```

This time, the outgoing variable *?SHOPlistOut* is erroneous, due to the error of the customer. As a result, the incoming variable *?SHOPlistIn* is also erroneous. This chronicle has to trigger the broker, its colour is set to red:

```
shopDataAcquisitionError[?PID,?shop,?supp] = { dataAcqError, RED,
  (Supplier::CNRcall,((?SHOPlistOut,err)),out,supp),
  (Supplier::CNRreturn,((?SHOPlistIn,err)),in,supp) }
```


inria-00282294, version 1 - 27 May 2008

The third case to consider is linked with an external error: the customer places his order right, but an external problem makes an item disappear on the list returned by SUPP. The customer cancels his order. This chronicle has exactly the same "local chronicle" as the data acquisition error one, but the *?SHOPlistOut* variable is not erroneous in this case:

```
shopExternalError[?PID,?shop,?supp] = { normal, RED,
  (Supplier::CNRcall,((?SHOPlistOut,notErr)),out,supp),
  (Supplier::CNRreturn,((?SHOPlistIn,err)),in,supp) }
```

As all this information is quite verbose, we sum it up in a table representing only the status of the synchronization variables of each chronicle. Red chronicles are represented in bold case. The table for service SHOP is shown below.

SHOP	*?SHOPlistOut*	*?SHOPlistIn*
normalBehav	$\neg err$	$\neg err$
dataAcqErr	err	err
externalErr	$\neg err$	err

6.2.2 Chronicles of the SUPP

There are also three basic execution cases on the SUPP service, each one leading to a chronicle. To preserve the legibility of this document, only the table summing up the status of the synchronization variables of each chronicle is detailed.

In the first two cases, the SUPP receives an order, transmits it to the WH, gets the booking confirmation for all the items and returns the new item list to the SHOP. But the SUPP does not know if the list contained in *?SUPPlistIn* was erroneous or not. That is why we have to foresee the two cases in two chronicles having the same local part (chronicles *normalBehaviour*, representing the ideal case, and *forwardError*, representing an erroneous *?SUPPlistIn* variable transmitted from the SHOP to the SUPP). Those two chronicles are green and will be sent to the broker only when the filter of the local diagnoser is set to *open*.

In the last case, the WH is not able to provide the SUPP with all the required items. The service does not update its itemlist before looping again, which differentiates the *externalError* chronicle from the two previous ones. This chronicle represents exclusively a faulty behaviour of one component of the system, its colour is set to red.

SUPP	*?SUPPlistIn*	*?SUPPitemOut*	*?SUPPitemIn*	*?SUPPlistOut*
normalBehav	$\neg err$	$\neg err$	$\neg err$	$\neg err$
forwardErr	err	err	err	err
externalErr	$\neg err$	$\neg err$	err	err

inria-00282294, version 1 - 27 May 2008

inria-00282294, version 1 - 27 May 2008

6.2.3 Chronicles of the WH

In the WH, like in the SUPP, we can consider a normal behaviour and a forwarded error that generate the same sequence of activities: check, update and reply. The difference between those two chronicles consists in the status of *?WHitemIn* and *WHitemOut*: $\neg err$ for the normal case and err for the erroneous one.

But the WH can also generate another sequence of events when an item is not available: check and reply. Let us suppose that the WH does not consider being out of stock as an error. This chronicle is green.

WH	*?WHitemIn*	*?WHitemOut*
normalBehav	$\neg err$	$\neg err$
forwardErr	err	err
nullCptStock	$\neg err$	err

6.3 Diagnosis process

Considering all the chronicles defined for the three web services mentioned in the previous section, we simulate an orchestrated execution and its corresponding diagnosis process.

At the beginning of the diagnosis process, the filter of each local diagnoser is set to *filter*, which means that only red chronicles are able to trigger the broker. As regards the broker, it has no knowledge of what is happening: it does not even know what services are running.

6.3.1 Normal execution

A customer places an order via the SHOP web service which stores it before transmitting it to the SUPPlier. For each product of this itemlist, the SUPPlier calls the WareHouse so as to book the corresponding product (the products are supposed to be sent only after the customer confirmation).

Fortunately, the WareHouse has enough goods to fulfil the needs of the customer who, after having received his bill, confirms his order.

In this case, the shopping process goes well and only green chronicles are recognized: as the filter of each local diagnoser is set to *filter*, the broker is not triggered.

6.3.2 Erroneous execution

The beginning of this second case is the same as the first one: a customer places an order and this order is forwarded to the SUPPlier. For each product of this itemlist, the SUPPlier calls the WareHouse so as to book the corresponding product.

Unfortunately, this time, one product is missing which provokes the recognition of the *nullComputerStock* chronicle. As mentioned in section 6.2.3, this chronicle is classified as a green chronicle because the WH does not consider being out of stock as an error. The broker is not triggered and the execution goes on.

inria-00282294, version 1 - 27 May 2008

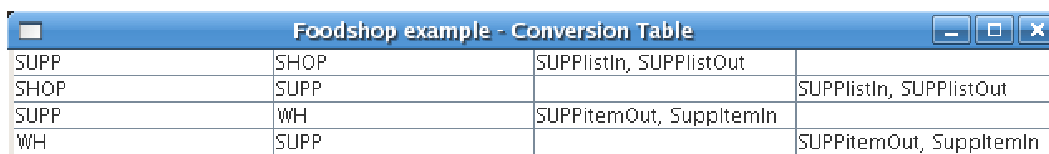
But when the SUPP receives the negative answer of the WH, the red chronicle called *externalError* is recognized and the SUPP forwards this chronicle to the broker, triggering a global diagnosis process.

```
externalError :
shop->supplier : SUPPlistIn (notErr)
supplier->warehouse : SUPPitemOut (notErr)
warehouse->supplier : SUPPitemIn (err)
supplier->shop : SUPPlistOut (err)

Broker knowledge :
(none)

New broker knowledge :
shop->supplier : SUPPlistIn (notErr)
supplier->warehouse : SUPPitemOut (notErr)
warehouse->supplier : SUPPitemIn (err)
supplier->shop : SUPPlistOut (err)
```

The broker integrates this knowledge, updating its conversion table and its diagnosis tree (Figures 11 and 12).



Foodshop example - Conversion Table

SUPP	SHOP	SUPPlistIn, SUPPlistOut	
SHOP	SUPP		SUPPlistIn, SUPPlistOut
SUPP	WH	SUPPitemOut, SuppItemIn	
WH	SUPP		SUPPitemOut, SuppItemIn

Figure 11: Conversion table after integration of the first chronicle

The recognized chronicle references two other services that the broker has to interrogate (according to the adopted diagnosis strategy). In this prospect, the broker sets to *open* the filter of the first service it interrogates, the WareHouse. This action has two consequences:

- all the previously recognized green chronicles are sent from the Warehouse to the broker;
- all the chronicles recognized in future will be sent to the broker.

After having received chronicle *nullCptStock* from the WH's local diagnoser, the broker integrates its new information. The new states of the conversion table and the diagnosis tree after this integration are shown respectively in Figures 13 and 14.

```
nullComputerStock :
```

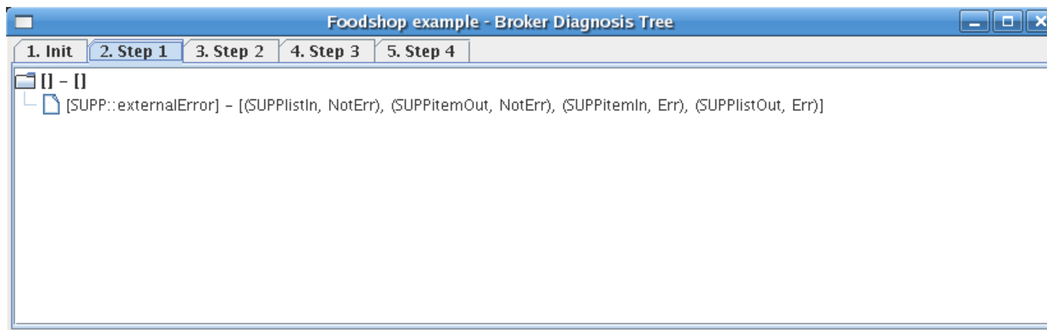


Figure 12: Diagnosis tree after integration of the first chronicle

```
supplier->warehouse : WHitemIn (notErr)
warehouse->supplier : WHitemOut (err)

Broker knowledge :
shop->supplier : SUPPlistIn (notErr)
supplier->warehouse : SUPPitemOut (notErr)
warehouse->supplier : SUPPitemIn (err)
supplier->shop : SUPPlistOut (err)

New broker knowledge :
shop->supplier : SUPPlistIn (notErr)
supplier->warehouse : SUPPitemOut (notErr)
supplier->warehouse : WHitemIn (notErr)
warehouse->supplier : SUPPitemIn (err)
warehouse->supplier : WHitemOut (err)
supplier->shop : SUPPlistOut (err)
```

Foodshop example - Conversion Table

SUPP	SHOP	SUPPlistIn, SUPPlistOut	
SHOP	SUPP		SUPPlistIn, SUPPlistOut
SUPP	WH	SUPPitemOut, SuppItemIn	WHitemIn, WHitemOut
WH	SUPP	WHitemIn, WHitemOut	SUPPitemOut, SuppItemIn

Figure 13: Conversion table after integration of the second chronicle

inria-00282294, version 1 - 27 May 2008

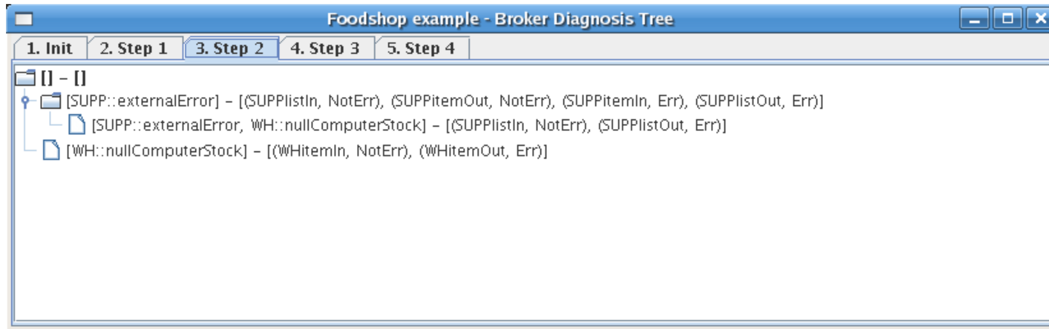


Figure 14: Diagnosis tree after integration of the second chronicle

While services implied in the faulty execution have not been invoked, the broker proceeds to interrogate the remaining local diagnosers. That way, the broker sets the SHOP's filter to *open* and waits for its recognized chronicles.

The diagnosis computation goes on, even if a possible diagnosis (empty node in the diagnosis tree) has been found (Figure 15).

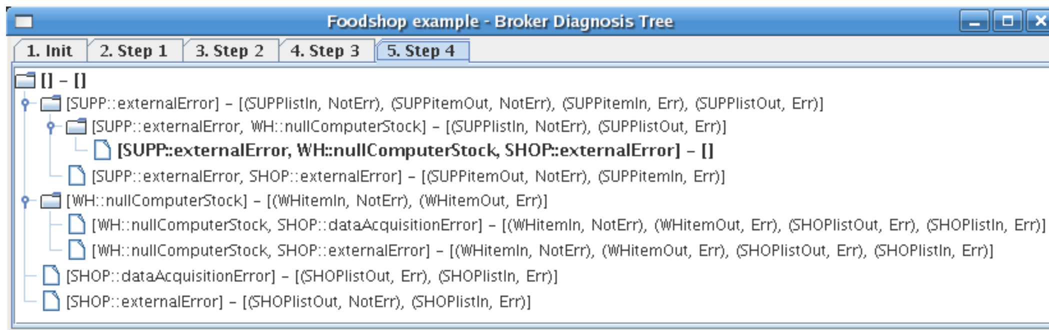


Figure 15: Diagnosis tree after integration of all the chronicles

7 Conclusion and prospects

Our contribution 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

inria-00282294, version 1 - 27 May 2008

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. Our platform, CarDeCRS allows us to make experiments on orchestrations of web services 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 cover the case of choreographies of web services, which do not rely on a “conductor” contrary to orchestrations. The second one is to couple the diagnosis service with a repair service (developed by a partner of ours), the goal being to ensure a good QoS, even in case of fault occurrences.

inria-00282294, version 1 - 27 May 2008

A Chronicle files grammars

A.1 BNF of the chronicle language

The up-to-date BNF of the chronicle language is available on the CRS website[4], with other useful documentation, such as the CRS API documentation.

<table>
<tr><th>Rule</th><th></th><th>Definition</th></tr>
<tr><td>ChronicleFile</td><td>::=</td><td>("include" <STRING> | DomainDefinition | AttributeOrMessage | TimeConstraintGraph | ConstraintModel | Priority | Chronicle)∗ <EOF></td></tr>
<tr><td>Priority</td><td>::=</td><td>"priority" <ID> "{" (TimePoint SymbolicInstantSequence)∗ "}"</td></tr>
<tr><td>AttributeOrMessage</td><td>::=</td><td>("deduced")? (Attribute | Message)</td></tr>
<tr><td>Chronicle</td><td>::=</td><td>"chronicle" Signature (TemporalParameters)? (":" "priority" <ID>)? "{" ((ChronicleStatement)∗) "}"</td></tr>
<tr><td>ChronicleStatement</td><td>::=</td><td>Predicate</td></tr>
<tr><td></td><td>|</td><td>Constraint</td></tr>
<tr><td></td><td>|</td><td>TimeConstraint</td></tr>
<tr><td></td><td>|</td><td>(<ID> | <HTYPE>) Parameters</td></tr>
<tr><td></td><td>|</td><td>"chronicle" Signature (TemporalLabeledIndexes)?</td></tr>
<tr><td></td><td>|</td><td>"when" "recognized" (WhenRecognizedActions | "{" (WhenRecognizedActions)∗ "}")</td></tr>
<tr><td>WhenRecognizedActions</td><td>::=</td><td>"print" <STRING></td></tr>
<tr><td></td><td>|</td><td>"emit" "event" "(" Signature "," TimePoint ")"</td></tr>
<tr><td>TimeConstraintGraph</td><td>::=</td><td>"time" "constraint" <ID> (TemporalParameters)? "{" (TimeConstraint)∗ "}"</td></tr>
<tr><td>ConstraintModel</td><td>::=</td><td>"constraint" Signature "{" (Constraint)∗ "}"</td></tr>
<tr><td>Attribute</td><td>::=</td><td>("attribute" Signature (":" <VAR>)? | "boolean" ("attribute")? Signature (":" <VAR>)? | "numerical" ("attribute")? Signature (":" <VAR>)? | "symbolic" ("attribute")? Signature (":" <VAR>)?) ("{" (Constraint)∗ "}")?</td></tr>
<tr><td colspan="3">*continued on next page...*</td></tr>
</table>

[4] http://crs.elibel.tm.fr/docs/language/grammar/index.html

inria-00282294, version 1 - 27 May 2008

continued from previous page

<table>
<tr><th>Rule</th><th></th><th>Definition</th></tr>
<tr><td>Message</td><td>::=</td><td>"message" Signature ("{" (Constraint)* "}")?</td></tr>
<tr><td>EventLog</td><td>::=</td><td>(TimedEvent)* <EOF></td></tr>
<tr><td>TimedEvent</td><td>::=</td><td>Date Signature (":" "(" ConstantParameter "," ConstantParameter ")")?</td></tr>
<tr><td></td><td>|</td><td>"(" Signature (":" "(" ConstantParameter "," ConstantParameter ")")? "," Date ")"</td></tr>
<tr><td>Date</td><td>::=</td><td><INTEGER></td></tr>
<tr><td>Predicate</td><td>::=</td><td>"noevent" "(" Signature (":" "(" Parameter "," Parameter ")")? "," "(" TimePoint "," TimePoint ")" ")"</td></tr>
<tr><td></td><td>|</td><td>"hold" "(" Signature ":" Parameter "," "(" TimePoint "," TimePoint ")" ")"</td></tr>
<tr><td></td><td>|</td><td>("context")? "event" "(" Signature (":" "(" Parameter "," Parameter ")")? "," TimePoint ")"</td></tr>
<tr><td>Signature</td><td>::=</td><td>(<ID> | <HTYPE>) (Parameters)?</td></tr>
<tr><td>Parameters</td><td>::=</td><td>"[" (Parameter ("," Parameter)*)? "]"</td></tr>
<tr><td>Parameter</td><td>::=</td><td>(<VAR> | "?" | "*")</td></tr>
<tr><td></td><td>|</td><td>ConstantParameter</td></tr>
<tr><td>ConstantParameter</td><td>::=</td><td><ID></td></tr>
<tr><td></td><td>|</td><td><TRUE></td></tr>
<tr><td></td><td>|</td><td><FALSE></td></tr>
<tr><td></td><td>|</td><td>("+")? <INTEGER></td></tr>
<tr><td></td><td>|</td><td>"-" <INTEGER></td></tr>
<tr><td>TemporalParameters</td><td>::=</td><td>"(" (<ID> ("," <ID>)*)? ")"</td></tr>
<tr><td>TemporalLabeledIndexes</td><td>::=</td><td>"(" (TimePoint ("," TimePoint)*)? ")"</td></tr>
<tr><td>TimeConstraint</td><td>::=</td><td>TimePoint SymbolicInstantSequence</td></tr>
<tr><td></td><td>|</td><td><ID> "-" <ID> ("in" TimeInterval | RightUpperBound | RightLowerBound)</td></tr>
<tr><td></td><td>|</td><td><ID> TemporalLabeledIndexes</td></tr>
<tr><td>SymbolicInstantSequence</td><td>::=</td><td>"=" TimePoint (SymbolicInstantSequence)?</td></tr>
<tr><td></td><td>|</td><td>"<" TimePoint (IncreasingInstantSequence)?</td></tr>
<tr><td></td><td>|</td><td>">" TimePoint (DecreasingInstantSequence)?</td></tr>
<tr><td></td><td>|</td><td>"<=" TimePoint (IncreasingInstantSequence)?</td></tr>
<tr><td></td><td>|</td><td>">=" TimePoint (DecreasingInstantSequence)?</td></tr>
<tr><td>IncreasingInstantSequence</td><td>::=</td><td>("=" TimePoint | "<" TimePoint | "<=" TimePoint) (IncreasingInstantSequence)?</td></tr>
</table>

continued on next page...

inria-00282294, version 1 - 27 May 2008

continued from previous page

<table>
<tr><th>Rule</th><th></th><th>Definition</th></tr>
<tr><td>DecreasingInstantSequence</td><td>::=</td><td>("=" TimePoint | ">" TimePoint | ">=" TimePoint) (DecreasingInstantSequence)?</td></tr>
<tr><td>TimePoint</td><td>::=</td><td><ID> ("+" (<INTEGER> | TimeInterval) | "-" (<INTEGER> | TimeInterval))?</td></tr>
<tr><td>Constraint</td><td>::=</td><td>ConstraintDisjunction ((<IMPLIES> ConstraintDisjunction | <EQUIV> ConstraintDisjunction))?</td></tr>
<tr><td></td><td>|</td><td>"if" ConstraintDisjunction "then" ConstraintDisjunction ("else" ConstraintDisjunction)?</td></tr>
<tr><td>ConstraintDisjunction</td><td>::=</td><td>ConstraintConjunction (<OR> ConstraintDisjunction)?</td></tr>
<tr><td>ConstraintConjunction</td><td>::=</td><td>AtomicConstraint (<AND> ConstraintConjunction)?</td></tr>
<tr><td>AtomicConstraint</td><td>::=</td><td><VAR> (InDomainConstraint | Equality Parameter)</td></tr>
<tr><td></td><td>|</td><td><NOT> AtomicConstraint</td></tr>
<tr><td></td><td>|</td><td>"(" ConstraintDisjunction ")"</td></tr>
<tr><td>InDomainConstraint</td><td>::=</td><td>("in" Domain | "not" "in" Domain | RightUpperBound | RightLowerBound)</td></tr>
<tr><td>DomainDefinition</td><td>::=</td><td>"domain" <ID> "=" Domain</td></tr>
<tr><td>Domain</td><td>::=</td><td>IntersectedDomain (("”— Domain | "\\" Domain))?</td></tr>
<tr><td>IntersectedDomain</td><td>::=</td><td>BasicDomain ("&" IntersectedDomain)?</td></tr>
<tr><td>BasicDomain</td><td>::=</td><td>"(" Domain ")"</td></tr>
<tr><td></td><td>|</td><td>"~" BasicDomain</td></tr>
<tr><td></td><td>|</td><td>"{" (SymbolicSequence | NumericalSequence | BooleanSequence)? "}"</td></tr>
<tr><td></td><td>|</td><td>TimeInterval</td></tr>
<tr><td></td><td>|</td><td><ID></td></tr>
<tr><td>SymbolicSequence</td><td>::=</td><td><ID> ("," <ID>)*</td></tr>
<tr><td>BooleanSequence</td><td>::=</td><td><TRUE> ("," <FALSE>)?</td></tr>
<tr><td></td><td>|</td><td><FALSE> ("," <TRUE>)?</td></tr>
<tr><td>NumericalSequence</td><td>::=</td><td>TimeValue ("," TimeValue)*</td></tr>
<tr><td>RightLowerBound</td><td>::=</td><td>">=" TimeValue</td></tr>
<tr><td></td><td>|</td><td>">" TimeValue</td></tr>
<tr><td>RightUpperBound</td><td>::=</td><td>"<=" TimeValue</td></tr>
<tr><td></td><td>|</td><td>"<" TimeValue</td></tr>
<tr><td>TimeValue</td><td>::=</td><td>("+")? <INTEGER></td></tr>
<tr><td></td><td>|</td><td>"-" <INTEGER></td></tr>
<tr><td>TimeInterval</td><td>::=</td><td>AtomicTimeInterval (("+" TimeInterval | "-" TimeInterval))?</td></tr>
</table>

continued on next page...

inria-00282294, version 1 - 27 May 2008

continued from previous page

Rule		**Definition**
AtomicTimeInterval	::=	TimeIntervalLeftBound "," TimeIntervalRightBound
	|	"-" AtomicTimeInterval
TimeIntervalLeftBound	::=	("[" (("+")? <INTEGER> | "-" <INTEGER>) | "]" "-" "oo")
TimeIntervalRightBound	::=	("+" ("oo" "[" | <INTEGER> "]") | ("-" <INTEGER> | <INTEGER>) "]")
Equality	::=	"=="
	|	"!="

inria-00282294, version 1 - 27 May 2008

A.2 BNF of the distributed chronicle language

In order to describe the synchronization part of distributed chronicles, a language was created, the BNF of which is shown below.

<table>
<tr><th>Rule</th><th></th><th>Definition</th></tr>
<tr><td>DistributedChronicleFile</td><td>::=</td><td>ModeList VarList (Chronicle)∗ <EOF></td></tr>
<tr><td>ModeList</td><td>::=</td><td>"MODE" "=" "{" Mode ("," Mode)∗ "}"</td></tr>
<tr><td>VarList</td><td>::=</td><td>"VAR" "=" "{" Var ("," Var)∗ "}"</td></tr>
<tr><td>Chronicle</td><td>::=</td><td>"distributed" "chronicle" Signature "=" Description</td></tr>
<tr><td>Signature</td><td>::=</td><td><IDENT> Parameters</td></tr>
<tr><td>Parameters</td><td>::=</td><td>"[" (Parameter ("," Parameter)∗)? "]"</td></tr>
<tr><td>Parameter</td><td>::=</td><td>"?" <IDENT></td></tr>
<tr><td>Description</td><td>::=</td><td>"{" Mode "," Color "," (Synchro ("," Synchro)∗)? "}"</td></tr>
<tr><td>Synchro</td><td>::=</td><td>"(" <IDENT> "," "(" ("(" Var "," State ")" ("," Var "," State)∗)? ")" "," Direction "," <IDENT> ")"</td></tr>
<tr><td>Direction</td><td>::=</td><td>"in" | "out"</td></tr>
<tr><td>State</td><td>::=</td><td>"err" | "notErr" | "undef"</td></tr>
<tr><td>Colour</td><td>::=</td><td>"GREEN" | "RED" | "ORANGE"</td></tr>
<tr><td>Mode</td><td>::=</td><td><IDENT></td></tr>
<tr><td>Var</td><td>::=</td><td><IDENT></td></tr>
</table>

The *ModeList* rule aims at listing the different working modes of the service, defined by expert knowledge, in order to associate each chronicle with a repair plan, in future works. The *VarList* rule lists all the synchronization variables (see section 3.3.2) of the service.

The first <IDENT> of a *Synchro* is the name of the event which sends or receives the associated synchronization variable to a remote service. The last <IDENT> of a *Synchro* is the type of this remote service.

inria-00282294, version 1 - 27 May 2008

B Detailed algorithms

Algorithm 7 describes the hidden part of algorithm 3, *i.e.* the parsing operation that transforms a *DistributedChronicleModel* into a list of remote services associated with their synchronization variables.

Data: a chronicle c
Result: a list $\mathcal{L}$ of $(remoteServ, var_1, \ldots, var_n)$ tuples
$\mathcal{L} = \emptyset$;
foreach *synchro s of c* **do**
  if $\exists t \in \mathcal{L} | t = (s.remoteService, \ldots)$ **then**
    update t;
  else
    $\mathcal{L} = \mathcal{L} \cup (s.remoteService, s.var_1, \ldots, s.var_n)$;
  end
end
return $\mathcal{L}$;

Algorithm 7: Conversion table construction, chronicle parsing

inria-00282294, version 1 - 27 May 2008

Algorithm 8 completes algorithm 5, detailing the compatibility test between two synchros.

```
Data: two synchros s_1 and s_2
Result: True if they are compatible, false else
if the source services are equal AND the remote services are equal then
 | /* same service sent two chronicles */ do nothing;
else if the source service of the first synchro is the remote service of the second
synchro AND vice versa then
 | /* remote service sends complementary variables */ translate the variable
 | names using the conversion table;
else
 | /* service that doesn't exchange with already known services */
 | return true;
end
foreach variable of the first synchro do
 | foreach variable of the second synchro do
 | | if the variables' names match then
 | | | if the variables' states are not equivalent then
 | | | | return false;
 | | | end
 | | end
 | end
end
return true;
```

Algorithm 8: Diagnosis tree construction, synchro compatibility

inria-00282294, version 1 - 27 May 2008

Algorithm 9 raises the last level of abstraction of algorithm 4. It details how a child node is created into the diagnosis tree.

```
Data: The node, the recognized chronicle
Result: A new child node
/* create the list of synchros */
create a new synchro list with the current node synchro list;
foreach synchro of the recognized chronicle do
    if the synchro is in the created list then
        /* a constraint has been checked */
        remove the synchro from the created list;
    else
        /* there is a new condition to check in order to establish a
        diagnosis */
        add the synchro to the created list;
        notify the broker that a new remote service has been integrated into the
        diagnosis;
    end
end
/* create the new node */
create a new empty node with the previously built list of synchros;
make this node a child of the given node;
/* create the list of chronicles */
copy the chronicle list of the given node into the new node;
add the recognized chronicle to this new list;
```

Algorithm 9: Diagnosis tree construction, creation of a child node

inria-00282294, version 1 - 27 May 2008

C Data description

C.1 Chronicle

DistributedChronicleModel: a *DistributedChronicleModel* is a CRS *ChronicleModel* enriched with some more attributes.

- `private String m_nom`: the name of the distributed chronicle;
- `private Vector<Synchro> m_sync`: the list of synchronization points of the chronicle model;
- `private String m_mode`: the mode of the chronicle;
- `private int m_colour`: the filter mode of the chronicle, used with the filter system.

SimplifiedDistributedChronicleModel: a *SimplifiedDistributedChronicleModel* is equivalent to a *DistributedChronicleModel* excepts that it can be serialized. Indeed, as *DistributedChronicleModel* extends the CRS *ChronicleModel* object, it is not possible to make it serializable. That is the reason why we had to create this new class containing only relevant information.

A *SimplifiedDistributedChronicleModel* has the same attributes as a *DistributedChronicleModel* object.

C.2 Synchro

A *Synchro* object represents a synchronization point.

- `private String m_activity`: the name of the activity linked with the synchronization point;
- `private Vector<StateVar> m_states`: the list of $(variable, status)$ pairs of the synchronization point;
- `private int m_direction`: the direction of the synchronization point (incoming or outgoing);
- `private String m_remoteService`: the remote service of the synchronization point;
- `private String m_sourceService`: the source service of the synchronization point.

StateVar: a *StateVar* is a pair $(variable, status)$.

- `private String m_varName`: the name of the *StateVar*;
- `private int m_varState`: the status (erroneous, not erroneous, undef) of the *StateVar*.

inria-00282294, version 1 - 27 May 2008

C.3 ConversionTable

A *ConversionTable* is just a set of conversion data.

- private Vector<ConversionObject> m_table: the table containing all the conversion data.

ConversionObject: a *ConversionObject* describes one entry of the *ConversionTable*.

- private String m_source: the source service;
- private String m_dest: the remote service;
- private String[ ] m_varSrc: the names of the variables in the source service;
- private String[ ] m_varDest: the names of the variables in the remote service.

C.4 DiagnosisTree

A *DiagnosisTree* is a tree used to diagnose what happens in an orchestration of services.

- private ArrayList<DiagnosisTree> m_childList: the list of children;
- private final ConcurrentLinkedQueue<Synchro> m_synchroList: the list of *Synchro*s of this node;
- private final ConcurrentLinkedQueue<String> m_chronicleList: the list of chronicles;
- private Broker m_broker: the reference to the *Broker*.

inria-00282294, version 1 - 27 May 2008

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] 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.

[4] A. Boufaied, A. Subias, and M. Combaceau. Distributed fault detection with delays consideration. In *Proceedings of the International Workshop on Principles of Diagnosis (DX'04)*, pages 57–62, 2004.

[5] G. Carrault, M.-O. Cordier, R. Quiniou, M. Garreau, J.-J. Bellanger, and A. Bardou. A model-based approach for learning to identify cardiac arrhythmias. In W. Horn, Y. Shahar, G. Lindberg, S. Andreassen, and J. Wyatt, editors, *Proceedings of AIMDM-99 : Artificial Intelligence in Medicine and Medical Decision Making*, volume 1620 of *LNAI*, pages 165–174, Aalborg, Denmark, june 1999. Springer Verlag.

[6] M.-O. Cordier and C. Dousson. Alarm driven monitoring based on chronicles. In *Proc. of Safeprocess'2000*, pages 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*, pages 230–240, 1998.

[8] M.-O. Cordier, X. Le Guillou, S. Robin, L. Rozé, and T. Vidal. Distributed chronicles for on-line diagnosis of web services. In *Proc. of the 18th Int. Workshop on Principles of Diagnosis (DX'07)*, 2007.

[9] 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.

[10] Michel Dojat, Nicolas Ramaux, and Dominique Fontaine. Scenario recognition for temporal reasoning in medical domains. *Artificial Intelligence in Medicine*, 14(1-2):139–155, 1998.

[11] C. Dousson. *Suivi d'évolutions et reconnaissance de chroniques.* Intelligence artificielle, Université Paul Sabatier, Toulouse, France, september 1994.

inria-00282294, version 1 - 27 May 2008

[12] C. Dousson. Extending and unifying chronicle representation with event counters. In *ECAI*, pages 257–261, 2002.

[13] C. Dousson and T. Vu Duong. Discovering chronicles with numerical time constraints from alarm logs for monitoring dynamic systems. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 620–626, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

[15] Christophe Dousson, 1994. Chronicle Recognition System. http://crs.elibel.tm.fr/.

[16] E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *Journal of Discrete Event Dynamic Systems*, 15:33–83, March 2005.

[17] J. Gamper and W. Nejdl. Formalizing reasoning about change: A temporal diagnosis approach. In *AI*IA*, pages 335–346, 1995.

[18] B. Guerraz and C. Dousson. Chronicles construction starting from the fault model of the system to diagnose. In *Proceedings of the International Workshop on Principles of Diagnosis (DX'04)*, pages 51–56, 2004.

[19] P. Laborie and J.-P. Krivine. Automatic generation of chronicles and its application to alarm processing in power distribution systems. 8th *international workshop of diagnosis (DX'97)*, september 1997. Mont Saint-Michel, France.

[20] E. Mayer. Inductive learning of chronicles. In *ECAI*, pages 471–472, 1998.

[21] 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.

[22] Yannick Pencolé, Marie-Odile Cordier, and Laurence Rozé. Incremental decentralized diagnosis approach for the supervision of a telecommunication network. In *IEEE Conf. on Decision and Control (CDC'02)*, pages 435–440, 2002.

[23] R. Quiniou, M.-O. Cordier, G. Carrault, and F. Wang. Application of ILP to cardiac arrhythmia characterization for chronicle recognition. *Lecture Notes in Computer Science*, 2157:220–227, 2001.

[24] N. Ramaux, D. Fontaine, and M. Dojat. Temporal scenario recognition for intelligent patient monitoring. In *AIME '97: Proceedings of the 6th Conference on Artificial Intelligence in Medicine in Europe*, pages 331–342, London, UK, 1997. Springer-Verlag.

inria-00282294, version 1 - 27 May 2008

[25] N. Rota and M. Thonnat. Activity recognition from video sequences using declarative models. In *Proc. of the 14th Int. Workshop on Principles of Diagnosis (DX'00)*, 2000.

inria-00282294, version 1 - 27 May 2008