# Building Distributed Chronicles for Fault Diagnostic in Distributed Systems using Continuous Query Language (CQL)

Juan Vizcarrondo, Jose Aguilar, Ernesto Expósito, Audine Subias

# Building Distributed Chronicles for Fault Diagnostic in Distributed Systems using Continuous Query Language (CQL)

[1] Vizcarrondo Juan, [2] Aguilar José, [3]Exposito Ernesto, [3]Subias Audine
[1]Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (CENDITEL),
[1]Mérida, Venezuela
[2]CEMISID, Dpto. de Computación, Facultad de Ingeniería, Universidad de Los Andes,
[2]Mérida, Venezuela
[3]CNRS, LAAS, 7, avenue du Colonel Roche, F-31400,
[3]Toulouse, France

***Abstract* - The chronicles paradigm has been used to determine fault in dynamic systems, allows modeling the temporal relationships between observable events and describing the patterns of behavior of the system. The mechanisms used until now usually use semi-centralized approaches, which consist of a central component, that is responsible for making the final inference about the fault diagnosis of the system based on the information collected from the local diagnosers. This model has problems when is implemented for monitoring very large systems, due to the bottleneck representing the central component. In this paper we define a recognition mechanism for a recognition fully distributed of chronicle using Continuous Query Language (CQL). This approach is tested in a classical Web Service Application.**

***Index Terms* - Distributed Pattern Recognition, Temporal patterns recognition, Chronicles, Distributed Fault Diagnostic, Web service composition, fault tolerance**

## I. INTRODUCTION

The chronicles have been used like diagnosis mechanisms for recognizing fault situations [2]. Dousson shown in earlier works on chronicles as a set of patterns, each characterized by observable events and temporal constraints among themselves and with respect to the context, represents an interpretation of what is happening in the dynamics of the system under study at a given time [5]. Thus, each chronicle represents a situation or scenario of normal or abnormal behavior of the system.

Some studies have used the formalism of chronicles for fault diagnosis [1, 2, 3, 4], but little has been carried out in a distributed context. The closest approach to our proposal provides a mechanism for semi-centralized recognition [1]. In this work, they propose an architecture composed of local diagnosers that monitor a set of events in its scope, and in case of any problem, it invokes a global diagnoser, who is in charge of making the recognition of the Chronicle (recognizes the failure). A tool for chronicles recognizing, called CRS (Chronicle Recognition System), has been developed by Dousson [11, 12]. Additionally, another tool, called Intelligent Event Processor (IEP), has been developed as a JBI1 service engine based on the language CQL [17] coupled as a module into openESB2, used for a events complex processing, allowing great power of inference about the detection of events.

In addition, SOA (Service Oriented Architecture) is a software development model in which an application is broken down into small units, logical or functional, called services. SOA allows the deployment of distributed applications very flexible, with loose coupling among software components, which operate in heterogeneous distributed environments [20], [21]. The services are inherently dynamics and cannot be assumed to be always stables [19], because the natural evolution of a service (changes in its interfaces, misbehavior during its operation, among others) can alter its results. Additionally, in the case of service composition the failure of a single service leads to error propagation in the others services involved, and therefore, the failure of the system. Such failures often cannot be detected and corrected locally (single service), thus it is necessary to develop architectures for enable diagnosis and correction of faults, both at individual (service) as global (composition) levels.

In previous work, we proposed a distributed architecture for faults diagnosis in the services composition, in which the faults diagnosis is performed through the interaction of diagnosers present in each service [10]. Similarly, repair strategies are developed through consensus among distributed repair services. This paper present our formalism of distributed chronicles previously developed [23], as mechanism for modeling distributed failure patterns, and particularly, its implementation using CQL and the IEP component, for the recognition distributed of a distributed chronicle.

1 Java Business Integration (JBI) is approach developed under the Java Community Process (JCP) to implement SOA
2 OpenESB is a Java-based open source enterprise service bus, http://www.open-esb.net/

## II. RELATE WORKS

Chronicles have been used to determine the faults present in the system [1, 2, 3, 4], commonly using a centralized or global diagnoser. On the other hand, few studies have proposed a distributed scheme of fault diagnosis using chronicles, in which local diagnosers are assigned to different components of the system, and synchronized to obtain a global diagnosis [6, 9]. In particular, [6] proposes a distributed verification of constraints using local and global events, which is enhanced by introducing delay between the communications present in the different diagnosers. Additionally, [5] studied the chronicles using petri nets, enriching the chronicles with pre and post information about event conditions. Also, [9] propose another mechanism for distributed failure diagnosis using chronicles, for which the chronicles are decomposed into many sub-chronicles as components are in the system, and observations are communicated between diagnosers, to thereby obtain the necessary information that is not available locally. The recognition system allows, on one level, recognizing each sub-chronicle, and the result is communicated to a global diagnoser to build the global diagnosis based on a story that contains the diagnosis of the union of the subchronicles that compose the system under study.

Another chronicle approach used in service composition is presented in [1]. This work has already been discussed above and basically proposes a decentralized architecture for fault diagnosis in composition services using chronicles. This architecture is comprised of two levels, in the first level local diagnosers are placed for each service that is part of the composition, which communicates with an global diagnoser (central) for the diagnosis of the entire composition. The global diagnoser is responsible for coordinating local diagnosers, using the message exchange to find the service and the activity responsible for the failure. In [14] propose a fully distributed architecture among local diagnosers, each diagnoser sends the received events to neighboring diagnosers, to enable the recognition of the global chronicle.

A tool for chronicles recognizing, called CRS (Chronicle Recognition System), has been developed by Dousson [11, 12]. It is responsible for analyzing the flow of events and recognizing, in real time, any pattern matching with a situation described by a chronicle. When a new event is logged in the system, new instances of chronicles are generated in the set of hypotheses. CarDeCRS (chronicle-based distributed diagnosis platform) is an extension of CRS, and allows semi-distributed diagnosis of chronicles [9]. Both tools CRS and CarDeCRS use the CRS language, which does not allow mathematical operators in the constraints of atemporal variables [18], and its implementation as a component of an SOA platform has not yet been developed, reducing its portability.

Furthermore, CQL is a declarative language used to perform queries on flow streams over long periods. Its syntax is very similar to SQL language, allowing great power of inference about events received, by having a large number of operators own of the SQL language as selection, aggregation, joining, grouping and other operators. Intelligent Event Processor (IEP) is an approach for Complex Event Processing (CEP) based on the Continuous Query Language (CQL) [17], whose implementation is automatically performed as SOA component-based.

Recently, in [10] we proposed reflexive middleware architecture for faults management in services composition, in which each service is overseen by a Local diagnoser using chronicles. To complete the proposal, this paper proposes the fault diagnosis system in web service composition, based on a chronicles recognition mechanism fully distributed using the Continuous Query Language, whose implementation is carried out in a SOA environment.

## III. OUR EXTENSION TO THE PARADIGM OF CHRONICLES: DISTRIBUTED CHRONICLES

A chronicle is a set of events with time constraints between them, which represents an interpretation of what is happening in the dynamics of the system under study at a given time [11]. Each chronicle represents a normal or abnormal situation or scenario of the system, which can be seen as a pattern of behavior of the system in this context. It is composed of a group of observable events, with temporal restrictions between them. A chronicle could generate new events and actions at the time of recognition of its occurrence, which could be used as input for other chronicles (it is a process of inference between chronicles [1]). Thus, in [13] defines a chronicle C as a "pair (E, T), where E is the set of events and T a set of constraints between their times of occurrence. When their variables and times of occurrence are instantiated is called an instance of the chronicle".

Thus, the representation of a chronicle is carried out by specifying [14]:

- A set of time points.
- A set of constraints between time points.
- A set of atemporal propositions representing activities that occur in the chronicle.
- A set of Reifying predicates representing the context of occurrence of the atemporal propositions.
- A set of external actions to execute when the chronicle is recognized.

Then, a chronicle model can be written as:

**Chronicle Model {**
**Events{**
event($e_1$, $T_1$), event($e_2$, $T_2$), event($e_3$, $T_3$)
**}**
**Constrains{**
$T_2$-$T_1$ < $C_1$
$T_3$-$T_2$ < $C_2$
**}**
**When recognized{**
$action_1$
$action_2$

}
}
Where:

[1] $e_i$ represents the set of atemporal propositions representing activities (events) in the chronicle.
[2] $T_i$ is the time points of occurrence of the events.
[3] Constrains: they are the set of constraints between $T_i$.
[4] $C_i$ they are constants representing the difference among the time points of occurrence of two events.
[5] Actions: they are the set of actions to execute when the chronicle is recognized.

Because the implementation of chronicles using centralized mechanisms requires a large number of components, it is costly. Therefore, the uses of distributed recognition mechanisms are suitable to achieve the diagnosis of failures in large systems. In the next section, we present our extension to the paradigm of Chronicles.

***Definition of distributed chronicles***

To begin to define our extension, it is essential to clarify that the detection of distributed chronicles is carried out by detecting a set of events $E=\{E_1, \ldots, E_2, \ldots, E_p\}$, distributed among the different n processes of the system [6]. In general, such events can be grouped into n sub-chronicles distributed. The recognition of the n sub-chronicles results in the recognition of a full chronicle. So, we can say that:

**Definition 1:** "A chronicle can be decomposed into n-sub-chronicles, in which each sub-chronicle SCi is assigned to a site/process Pi of the system under study, and describes a sub-set of events $Eac_i$, with $Tac_i$ temporal restrictions, that must occur and respect in that site i in order to recognize the chronicle".

$$C(E,T) = UNION_{i=1,\,n}(SC_i(Eac_i, Tac_i)) \quad (1)$$

where,

- $Eac_i$ and $Tac_i$ correspond to a set of events and temporal restrictions of the chronicle assigned to each component i, where $Eac_i = \{E_k,,\ldots, E_l\}$, $Tac_i = \{T_k,,\ldots, T_l\}$ and $E_{k,\ldots,} E_l \in E$, $T_k, \ldots, T_l \in T$, and these events $E_{k,}, \ldots, E_l$ occur in site i}.
- UNION is a predicate defined by the union of the set of events $(\bigcup_{i=1}^{n} Eac_i)$ and of the set of temporal constraints $(\bigcup_{i=1}^{n} Tac_i)$ distributed in the n sub-chronicles.

**Definition 2:** because a chronicle C can be decomposed into n sub-chronicles (SC), the recognition of the global chronicle can be carried out in a sub-chronicle $SC_i$, recognizing its events ($Eac_i$, $Tac_i$), with the union of the partial recognition of the other sub-chronicles ($SC_j \; \forall \; j=1,n \mid j \neq i$) of the other events (in this case, the other sub-chronicles must send it a message to inform it the recognition of their events). In this way. the sub-chronicle $SC_i$ recognizes the chronicle C(E, T):

$$C(E, T) = \{(Eac_i, Tac_i), UNION_{j=1,\,n \mid j \neq i}(SC_j(Eac_j, Tac_j))\} \quad (6)$$

Particularly, because a chronicle is decomposed into n sub-chronicles, it is necessary to extend the representation of sub-chronicles to correlate the events recognized between different sub-chronicles, and be able to recognize the global chronicle. For this, we extend the formalism of chronicles by adding several aspects to manage the synchronization process between the sub-chronicles, as follows (see Fig 1):

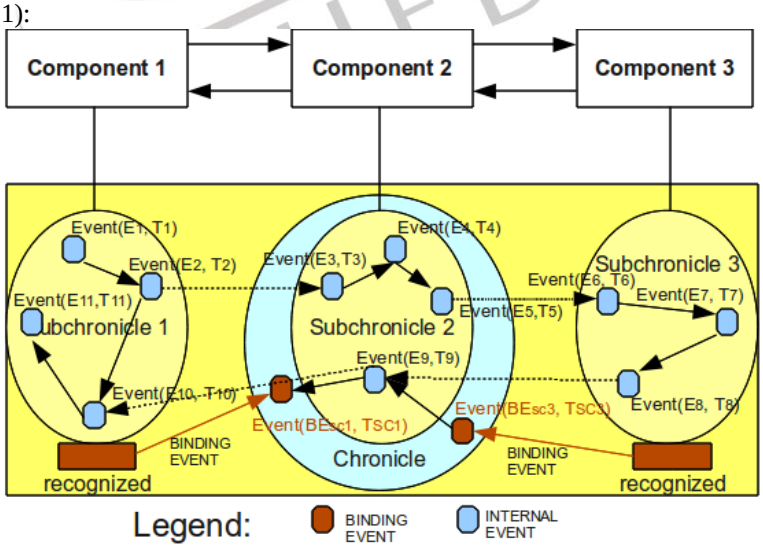



Figure 1. Example of chronicle decomposed in sub-chronicles

**Definition 3 Variables State:** It shows if the value of a variable of an event in the chronicle is normal (¬Err) or abnormal (Err). Usually, we can denote it using a Boolean value (0 and 1), but the abnormal state can be expanded to enrich the classification of this behavior.

**Definition 4 Binding Events (BE)**: Are events from sub-chronicles, to connect them to other sub-chronicles, and thus to represent the communication between sub-chronicles. A binding event $BE_j$ is instanced when a neighbor sub-chronicle is recognized and then is propagated to the other sub-chronicles. Thus, the recognition (output event) of a sub-chronicle $SC_i$ can be linked to the $BE_i$ events belonging to a sub-chronicle $SC_j$

**Definition 5:** we define a Distributed Chronicle as "a classical chronicle C decomposed into a set of sub-chronicles $SC_i$, which are linked together via Binding Events".

***Distributed Chronicles Recognition***

The recognition process of distributed chronicles must be fully distributed. For this, it is placed a recognition system in each component of the system, which will be responsible to recognize the sub-chronicle in this place. As mentioned above, each sub-chronicle is linked to other events through binding events, which will allow to the local recognition system infers information from their neighboring components. Using this information, the recognition system can recognize local sub-chronicles, generate events to neighboring sites, and in general, spread the inferred. The inclusion of binding events allows each site to have a global vision of the entire system.

The architecture of each site is shown in Fig. 2, and consists of a local chronicle recognition module, which receives events from the local monitor, and events generated by other neighboring sites (BE). For example, we see in Figure 2 as the site 2, in a given time, receives events from the monitor 2, and those generated by the sub-chronicles at sites 3 ($BE_3$) and 1 ($BE_1$).

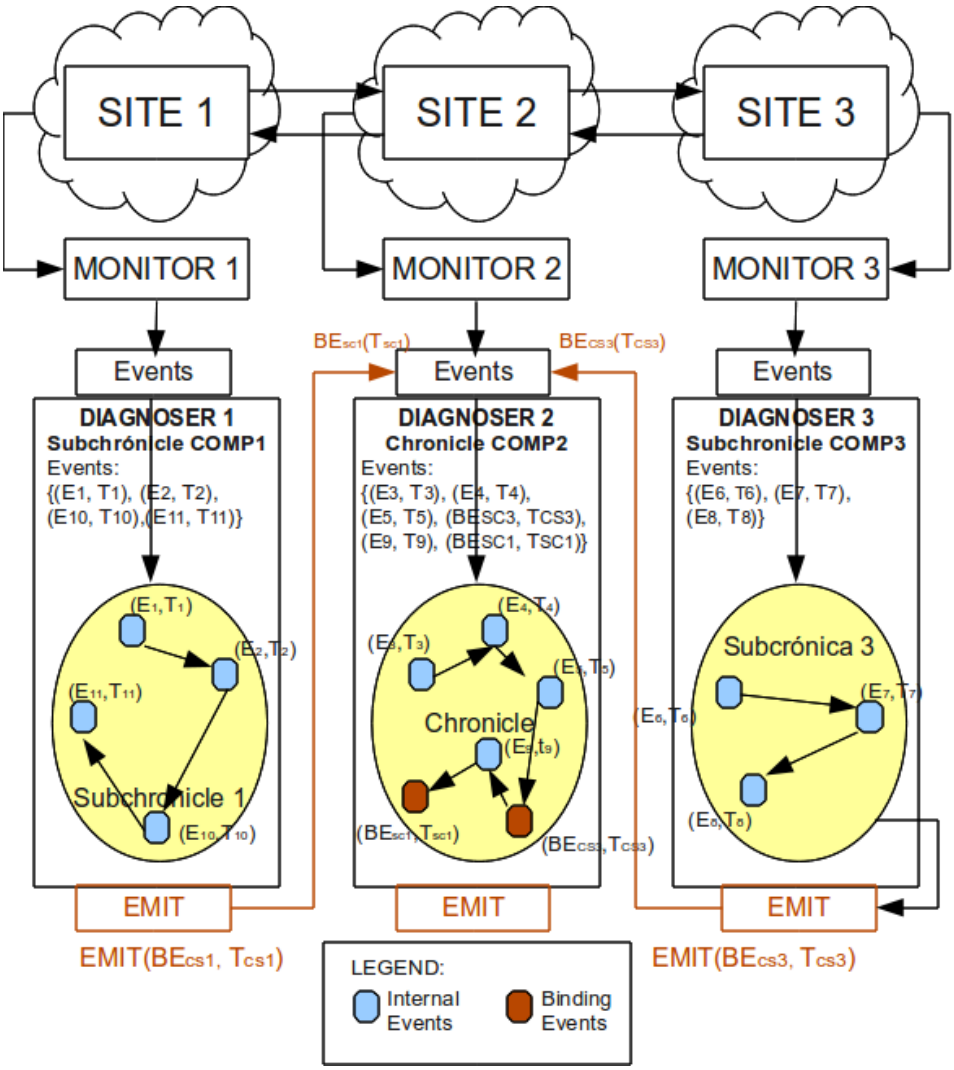



Figure 2. Distributed CRS

The Chronicle Model of the Fig. 2 can be written as:

| Chronicle Subchronicle 1 { | Chronicle Subchronicle 2 { | Chronicle Subchronicle 3 { |
|---|---|---|
| **Events{**<br>event($E_1$, $T_1$), event($E_2$, $T_2$), event($E_{10}$, $T_{10}$), event($E_{11}$, $T_{11}$)<br>**}**<br>**Constrains{**<br>$T_2$-$T_1 \leq C_1$<br>$T_{10}$-$T_2 \leq C_2$<br>$T_{11}$-$T_{10} \leq C_3$<br>**}**<br>**When recognized{**<br>Emit event($BE_{SC1}$,$T_{SC1}$) to Diagnoser 2<br>**}**<br>**}** | **Events{**<br>event($E_3$, $T_3$), event($E_4$, $T_4$), event($E_5$, $T_5$), event($BE_{SC3}$, $T_{SC3}$), event($E_9$, $T_9$), event($BE_{SC1}$, $T_{SC1}$)<br>**}**<br>**Constrains{**<br>$T_4$-$T_3 \leq C_4$<br>$T_5$-$T_4 \leq C_5$<br>$T_{SC3}$-$T_5 \leq C_6$<br>$T_9$ -$T_{SC3} \leq C_7$<br>$T_{SC1}$-$T_9 \leq C_8$<br>**}**<br>**When recognized{**<br>Create log(Fault 1)<br>**}**<br>**}** | **Events{**<br>event($E_6$, $T_6$),event($E_7$, $T_7$), event($E_8$, $T_8$)<br>**}**<br>**Constrains{**<br>$T_7$-$T_6 \leq C_9$<br>$T_8$-$T_7 \leq C_{10}$<br>**}**<br>**When recognized{**<br>Emit event($BE_{SC3}$,$T_{SC3}$) to Diagnoser 2<br>**}**<br>**}** |

Figure 3. Chronicle Model example

***Representation of Distributed Chronicles using CQL***

CQL is a declarative language for manage continuous queries in a flow streams[3]. Syntactically, CQL is very similar to the SELECT statement of SQL, but queries execution are different from conventional database queries in SQL, where queries are executed on demand and run until all requested data is completed. Conversely, in CQL queries are continuous on streams running indefinitely (infinite stream of tuples), or until they are terminated.

The CQL syntax contains many operators found in SQL as projection, selection, aggregation, joining, grouping, etc., but CQL has other operators to convert streams in relations[4] [15, 16]:

- **Stream to Relation Operators:** Are based on the concept of a sliding window over a stream. These are divided into:
  - **Time-based:** converts a stream to a relation based on a specified period of time, EG: Event 1 [Range 30 Seconds].
  - **Tuple-based:** converts a stream to a relation based on a specified number of events, EG: Event 1 [Rows 3].
  - **Attribute based:** converts a stream to a relation based on a specified attribute and a size that defines the range of values for the attribute, Eg: Event 1 [Partition By Status = TRUE] .
  - **Partitioned:** convert a stream to a relation based on a specified attribute and a specified number of events, EG: Event 1 [Partition By Status = TRUE Rows 2] .
- **Relation-to-Relation Operator:** maps semantic relations of temporal variables, using similar queries used in SQL statements:

**SELECT** event1 .id
**FROM** event1 [Range 30 Seconds]
**WHERE** event1.id > 10

- **Relation to Stream Operators:** Are used when is required convert query relations to stream.:
  - **ISTREAM:** converts to stream a relation that is at time T but was not at time T-1.
  - **DSTREAM:** converts to stream a relation that was at time T-1 but was not at time T.
  - **RSTREAM:** converts to stream a relation that is at time T.

Particularly, reified temporal logic can be used on the representation of chronicles, allowing propositional terms with temporal objects (reifying predicates). The Reifying predicates used in chronicles are described in [14]. Below are presented the necessary structure to represent the reifying predicates used in chronicles in CQL language:

Table 1. Reifying predicates in CQL language.

| Temporal Predicate of the Chronicles | Description | Temporal Predicate of the Chronicles in CQL | Query CQL description |
|---|---|---|---|
| hold(P : v, (t1 , t2 )) | The domain attribute P must keep the value y on the interval [t1 , t2]. | **SELECT**<br>ISTREAM('newevent')<br>**FROM**<br>Event0[t2 - t1],<br>Event1[now]<br>**WHERE**<br>Event0.P = 'v' AND<br>NOT(Event0.P <> 'v') AND<br>Event1.P = 'v' | To sure that the attribute P keeps the value v on the interval [t1 , t2], the attribute of the incoming stream Event1 (time = NOW) must have the value v. Additionally, at least one previous value of the attribute of the relation Event0(time = [t2 – t1] has been y. Note that this guarantees the interval keeps previous value of |

3 Streams are a series of time stamped events that have the same schema.
4 Relations are collection of events that match a condition at a point in time.

| | | | |
|---|---|---|---|
| | | | Event0 at time NOW - t2 + t1 equal to v. |
| event(P : (v1 , v2 ), t) | The attribute P changes its value from v1 to v2 at t. | **SELECT**<br>ISTREAM('newevent') **FROM**<br>Event0[ΔT],<br>Event1[NOW]<br>**WHERE**<br>Event0.P = 'v1' AND<br>NOT(Event0.P = 'v2')<br>AND Event1.P = 'v2' | The current stream Event1 has attribute P (time = NOW) as value v2. Additionally there is at least one relation with the attribute P equal to v1 at time ΔT (ΔT has a value which sures that the attribute P can have the value v1. |
| event(P, t) | Message P occurs at t. | **SELECT**<br>ISTREAM('newevent')<br>**FROM**<br>event[NOW]<br>**WHERE**<br>event.msg = 'P' | The current stream Event has Message P at time NOW. |
| noevent(P, (t1 , t2 )) | The chronicle would not be recognized if any change of the value of the domain attribute P occurs between t1 and t2. | **SELECT**<br>ISTREAM('newevent')<br>**FROM**<br>Event0[t2 - t1],<br>Event1[NOW]<br>**WHERE**<br>NOT (event0.msg = 'P') AND<br>NOT (event1.msg = 'P') | The current stream Event has not Message P at time NOW and do not exist previous relation with Message P from time: NOW - t2 + t1. |
| occurs((n1 , n2 ), P, (t1 , t2 )) | The event that matches the pattern P occurred exactly N times between the two time points t1 and t2 . The value ∞ can be used for n2. | **SELECT**<br>ISTREAM('newevent')<br>**FROM**<br>event[NOW]<br>**WHERE**<br>event.msg = 'P' AND<br>(***SELECT***<br>count(event)<br>***FROM***<br>event[t2 - t1]<br>***WHERE***<br>event.msg = 'P'<br>) >= n1 - 1 AND<br>(***SELECT***<br>count(event)<br>***FROM***<br>event[t2 - t1]<br>***WHERE***<br>event.msg = 'P'<br>) <= n2 - 1 | The current stream Event has Message P at time NOW and occurred a number N of time between n1 - 1 and n2 - 1 at time t2 - t1. |

Then, a chronicle model can be written in CQL using Relation-to-Stream Operators as:

**SELECT** ISTREAM($e_3$.id, e3.time, 'Situations of interest')
**FROM** $e_1$ [Range $C_1$ Seconds], $e_2$ [Range $C_2$ Seconds], $e_3$ [now]
**WHERE** Constrains

Where:

- ***SELECT*** defines the outcome of a recognized chronicle (**When is recognized**), generating with this a relation that can be used by another chronicle or component system
  - *ISTREAM* describes the content of the relation generated by the chronicle, which may have a mixture of different attributes of the relations and streams present in chronicle. In this case the operator ISTREAM is used, but could be used any other. Additionally, id and time attributes are used in the generation of the stream:
    - $e_i$.id represents the identifier events in the chronicle.
    - $e_i$.time is the time points of occurrence of the events.
    - The last attribute is the assignment of the name of the event that is being generated. In this case is established a generic name called 'Situations of interest".
- ***FROM*** represents the set of streams and relations that are used in the chronicles to extract the events. This section has all events that are part of a chronicle, expressed as streams and relations:
  - $C_i$ they are constants representing the difference among the time points of occurrence of two events.
  - $e_i$ *[Range* $C_i$ *Seconds]* describes a relation which were converted from the stream $e_i$ using the time-based Operators (Stream to Relation) and are stored by $C_i$ seconds.
  - $e_j$ *[now]* defines the stream $e_j$ is happening at this time (now).
- ***WHERE:*** are the set of constraints between Ti (temporal variables) and other event attributes (atemporal variables).

In this way, when chronicle is recognized, it emits a stream with some attributes and other information using the select statement. Thus, the distributed chronicle which was shown in Fig. 3 can be implemented in CQL with the following code:

```
Chronicle Subchronicle 1 {
 SELECT
   ISTREAM(
    'BESC1',
    'Diagnoser 2'
   )
FROM
   E1[C1 + C2 +C3],
   E2[C2 + C3],
   E10[C3],
   E11[now]
WHERE
   E2.time > E1.time AND
   E10.time > E2.time AND
   E11.time > E10.time
}
```

```
Chronicle Subchronicle 2 {
 SELECT
   ISTREAM(
    'log',
    'Fault 1'
   )
FROM
   E3[C4 + C5 + C8],
   E4[C5 + C8],
   E5[C8]
   BESC3[C8 - C7]
   BESC1[now]
WHERE
   E4.time > E3.time AND
   E5.time > E4.time AND
   BESC3.time > E5.time AND
   BESC1.time > BESC3.time
}
```

```
Chronicle Subchronicle 3 {
 SELECT
   ISTREAM(
    'BESC3',
    'Diagnoser 2'
   )
FROM
   E6[C9 + C10],
   E7[C10],
   E8[now]
WHERE
   E7.time > E6.time AND
   E8.time > E7.time
}
```

Figure 4. Chronicle Model example in CQL

As is shown in figure 4, we can express the chronicle of Figure 3 in CQL statement, for this we became the set of events in streams and relations. The explanation of the conversion for sub-chronicle 1 is shown below:

- **SELECT:** At moment when sub-chronicle 1 is recognized, it is necessary to emit the event $BE_{SC1}$ to Diagnoser 2, for this we define the production of a stream using the converter ISTREAM with event name $BE_{SC1}$ and destination Diagnoser 2.
- **FROM:** The last event which reaches sub-chronicle is $E_{11}$, it is a stream that occurs at this time (now). Additionally, $E_1$, $E_2$ and $E_{10}$ events are converted in relations using the operator Time-based with constant period of time $C_3$ to $E_{10}$, $C_2 + C_3$ to $E_2$ (time of occurrence of $E_2$, which must occur before $E_{10}$, so the event $E_2$ must consider the period of $E_{10}$ plus period of $E_2$: $C_2 + C_3$) and $C_1 + C_2 + C_3$ to $E_1$ (time of occurrence of $E_1$ must occur before $E_2$, so the event $E_1$ must consider the period of $E_2$ plus period of $E_1$: $C_1 + C_2 + C_3$) .
- **WHERE:** the single definition of events as relations does not guarantee the correct order of arrival of events (p.e. $E_1$ event could occur after $E_2$). Thus, it is necessary to establish restrictions on the order in which events should arrive in the sub-chronicle, in this way, is established that the event E1 must occur before $E_2$, then the event $E_{10}$, and finally the event $E_{11}$.

How the chronicles 2 and 3 are performed similarly, in the case of sub-chronicle 2 is assumed that the event $E_8$ occurs after $E_7$ ($C_8 > C_7$). Additionally, we added the attribute time to events stream to facilitate inference about the sequence of occurrence of these and express their exact occurrence in the real system, which is not necessarily the arrivals time in the chronicle recognizer (communication times can affect the arrivals time from real system to the recognizer). Finally, the 3 sub-chronicles have emitted the corresponding events $BE_{SC1}$, $BE_{SC3}$ and log, which should be routed to theirs destination (for example, diagnoser 2 from sub-chronicles 1 and 3).

## IV. Case Study

In order to test our proposal, we will use a common example of e-commerce SOA implementation (see Fig 5), which comprises three business processes (which will constitute our services):

- **Shop:** the shop where users purchase products.
- **Supplier** offers products to the store, it needs to check their availability before making a response to the store.
- **Warehouse:** where the products are stored in the providers. This process has a service level agreement (SLA)[5] with Supplier, which is that at least one product from the list should be returned[6]. It can invoke to other warehouse to search products in other warehouses of the company. This property allows to answer at least one product when the required amount is not in the local warehouse.

5 SLA is a contract between the service consumer and service provider and define the level of service

6 This SLA define how message delivery is guaranteed, the Warehouse delivery messages in the proper order (least one product in order)

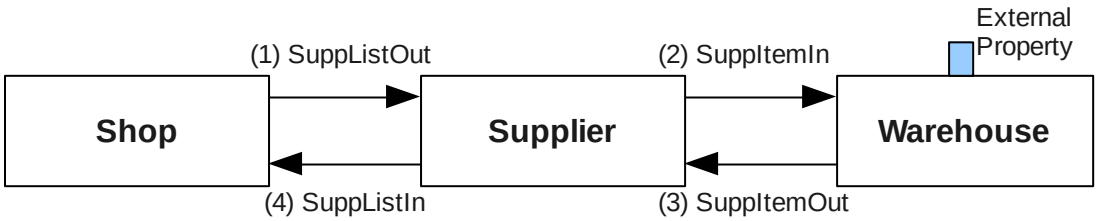



Figure 5. Example of choreography (e-commerce).

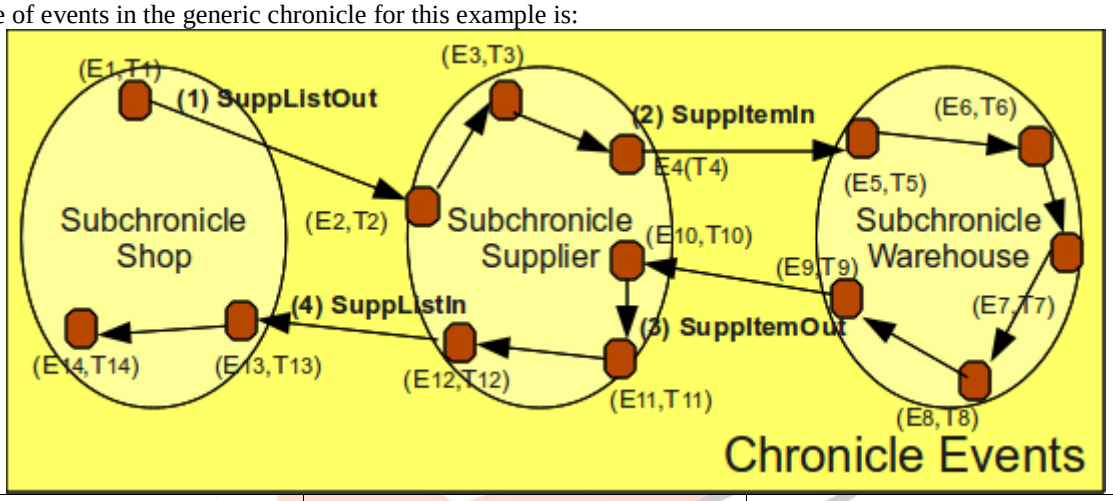
Now, we describe a classical behavior of this application:

(1) **SuppListOut:** Shop provides the list of products required to the supplier.
(2) **SuppItemIn:** Supplier checks its deposit invoking the Warehouse process.
(3) **SuppItemOut:** Warehouse provides the answer about the list of products in the deposit to the Supplier, which must contain at least one product.
(4) **SuppListIn:** The Supplier notifies the Shop which products can provide.

***Design of Chronicles***

Let us now characterize the distribution of events among different diagnosers (sites) that are part of the composition, which will allow us to build a *generic chronicle* for that particular application (connecting all events that may occur). With this generic chronicle we can derive each specific chronicle to detected abnormal situations. For practical reasons, we consider that the time is measure in seconds, and delay in communications and the recognition time of chronicles are negligible.

The sequence of events in the generic chronicle for this example is:




| Shop | Supplier | Warehouse |
|---|---|---|
| • **$E_1$:** Shop sends product order to Supplier.<br>• **$E_{13}$:** Shop receives the list of products.<br>• **$E_{14}$:** Shop makes products payment. | • **$E_2$:** Supplier receives product order<br>• **$E_3$:** Supplier checks the products in the catalog.<br>• **$E_4$:** Supplier provides product order to Warehouse for the products that it has not.<br>• **$E_{10}$:** Supplier receives the response of the products.<br>• **$E_{11}$:** Supplier makes the invoice.<br>• **$E_{12}$:** Supplier responds to shop with products shipped. | • **$E_5$:** Warehouse receives the request of the Supplier.<br>• **$E_6$:** Warehouse searches products (maybe it invokes other warehouses).<br>• **$E_7$:** Warehouse updates inventory.<br>• **$E_8$:** Warehouse packs and ships products to the buyer.<br>• **$E_9$:** Warehouse provides the answer about the list of products in the deposit to the Supplier . |

Figure 6. Events in each service in the E-Commerce case.

Now, we can define the specific chronicle for each failure that we like diagnose. We will consider the following failure scenario:

- A failure because there is a violation of a Warehouse Service Agreement (SLA violation).

From current events in the choreography, we build specific chronicles for each fault. In general, each specified chronicle is decomposed into the same three sub-chronicles defined above. Sometimes, for a given situation maybe we can need less sub-chronicles, as in the case of the failures studied (see Fig. 7, which shows the structure of the specified chronicle for the Warehouse SLA Violation).

To design the Chronicle Model for SLA Violation Warehouse, we analyze events in the Warehouse: the problem of SLA violation occurs when the Warehouse Service performs the products search in the event $E_6$ (Warehouse searches products), and it does not get anything, or it does not invoke another Warehouse ($E_7$), or Warehouse fails in packs and ships some products to the buyer ($E_8$). In all cases, it emits the response to Supplier with a list empty of products. Then, the Shop service detects a fault in the event $E_{10}$ (Supplier receives the response of the products). For this reason, the sub-chronicle in Supplier sends the event $BE_{SLA}$ to Warehouse diagnoser (really, the CRS in supply diagnoser sends the message), and then this diagnoser can recognize the fault SLA Violation. When the Warehouse Diagnoser recognizes the chronicle, it invokes the repair with the fault found.

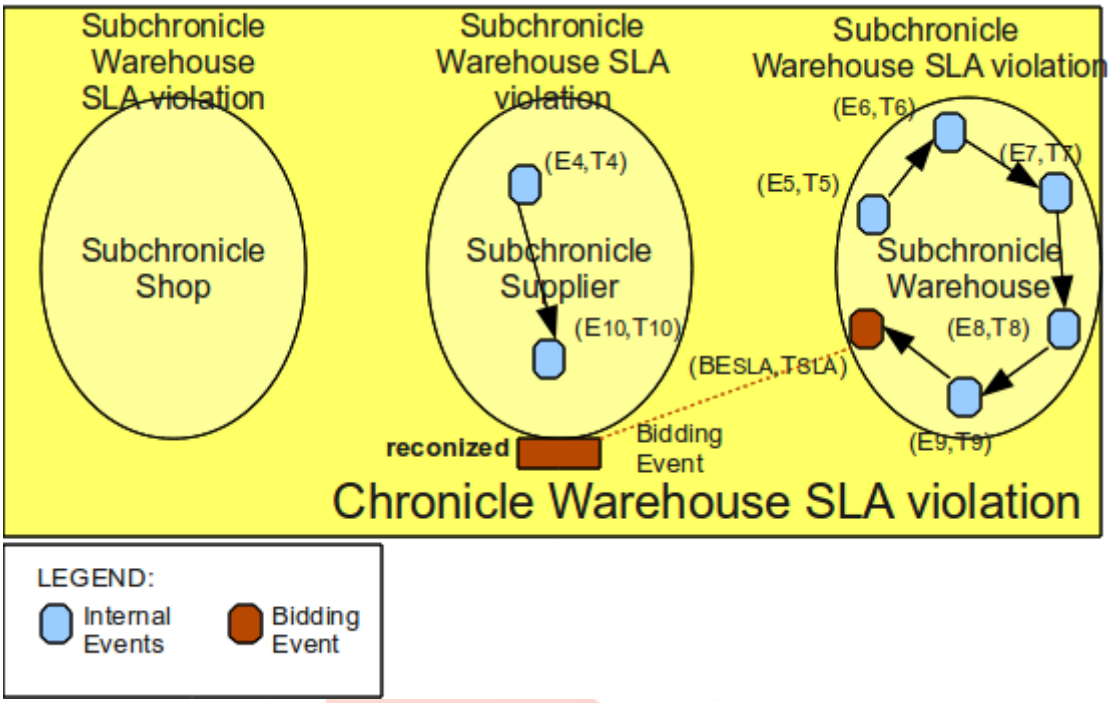



Figure 7. Distribution of events in sub-chronicles on SLA Violation Warehouse and Warehouse Service Delay faults.

For the case of SLA Violation fault (case of products search), the chronicle would be (at the beginning in classical Chronicles notation and then in CQL):

| **Chronicle SLA Violation fault (case of products search) in Reifying predicates (CRS and CarDeCRS)** | | |
|---|---|---|
| **Subchronicle Shop SLA {**<br>**}** | **Subchronicle Supplier SLA {**<br>**Events{**<br>event(E4, T4, lp = TRUE), event(E10, T10, lp = FALSE)<br>**}**<br>**Constrains{**<br>T10-T4 ≤ 10<br>**}**<br>**When recognized{**<br>Emit event(BESLA,TBESLA) to Diagnoser Warehouse<br>**}**<br>**}** | **Subchronicle Warehouse SLA Search {**<br>**Events{**<br>event(E5, T5, lp = TRUE), event(E6, T6, lp = FALSE), event(EBESLA, TBESLA, lp = FALSE)<br>**}**<br>**Constrains{**<br>T6-T5 ≤ 2<br>TBESLA-T6 ≤ 8<br>**}**<br>**When recognized{**<br>Emit event('ServiceLevelAgreement','SearchProducts') to Repair Warehouse<br>**}**<br>**}** |
| **Chronicle SLA Violation fault (case of products search) in CQL language** | | |
| **Subchronicle Shop SLA {**<br>**}** | **Subchronicle Supplier SLA {**<br>**SELECT**<br>ISTREAM(<br>id => E4.id,<br>event => 'BESLA',<br>time => E10.time,<br>lpsupplier => E10.lp,<br>lp => E4.lp,<br>to => 'Diagnoser 3',<br>)<br>**FROM**<br>E4[10],<br>E10[now]<br>**WHERE** | **Subchronicle Warehouse SLA Search {**<br>**SELECT**<br>ISTREAM(<br>id => E5.id,<br>fault => 'ServiceLevelAgreement',<br>faulttype => 'SearchProducts',<br>time => EBESLA.time,<br>lp => E6.lp,<br>to => 'Repair 3',<br>)<br>**FROM**<br>E5[10],<br>E6[8],<br>EBESLA[now] |

| | | |
|---|---|---|
| | E10.time > E4.time AND<br>E10.id = E4.id AND<br>E4.lp - E10.lp > 0<br>**}** | **WHERE**<br>E6.time > E5.time AND<br>EBESLA.time > E6.time AND<br>E5.lp > 0 AND<br>E6.lp = 0 AND<br>E6.id > E5.id AND<br>EBESLA.id > E6.id AND<br>**}** |

Figure 8. Chronicle Model for SLA Violation Warehouse (case of Products search)

where pl is the product list.

It is important to note the difference in how the value of attribute lp in sub-chronicle Supplier SLA is represented in temporal predicates and in CQL. CQL can represent the difference of the attribute lp for the events $E_4$ and $E_{10}$ very easy ($E_4$.lp – $E_{10}$.lp). To CRS and CarDeCRS do not allow mathematical operators in the constraints of atemporal variables (p.e. attributes), making very complex the management of the values of the lp attribute in different events (for that, we need to determine when the value of lp is correct (TRUE) and when it is wrong (FALSE)). Using the CQL, it enriches the way as the chronicles are represented, because it allows mathematical operators in the constraints of atemporal variables.

The Chronicle model for SLA Violation fault (case of fails due to pack and ship) is very similar, but pl=0 in event $E_8$ (Warehouse fails due to pack and ship) is minor that event $E_7$ (Warehouse updates inventory);

| | | |
|---|---|---|
| **Subchronicle Shop SLA {**<br>**}** | **Subchronicle Supplier SLA {**<br>**SELECT**<br>ISTREAM(<br>id => E4.id,<br>event => 'BESLA',<br>time => E10.time,<br>lpsupplier => E10.lp,<br>lp => E4.lp,<br>to => 'Diagnoser 3',<br>)<br>**FROM**<br>E4[10],<br>E10[now]<br>**WHERE**<br>E10.time > E4.time AND<br>E10.id = E4.id AND<br>E10.lp < E4.lp<br>**}** | **Subchronicle Warehouse SLA** Packs and ships **{**<br>**SELECT**<br>ISTREAM(<br>id => E5.id,<br>fault => 'ServiceLevelAgreement',<br>faulttype => 'Ship and Packed',<br>time => EBESLA.time,<br>lp => E8.lp,<br>to => 'Repair 3',<br>)<br>**FROM**<br>E5[10],<br>E6[8],<br>E7[7],<br>E8[5],<br>EBESLA[now]<br>**WHERE**<br>E6.time > E5.time AND<br>E7.time > E6.time AND<br>E8.time > E7.time AND<br>EBESLA.time > E8.time AND<br>E5.lp > 0 AND<br>E5.lp = E6.lp AND<br>E6.lp = E7.lp AND<br>E8.lp < E7.lp AND<br>E6.id = E5.id AND<br>E7.id = E6.id AND<br>E8.id > E7.id AND<br>EBESLA.id > E7.id<br>**}** |

Figure 9. Chronicle Model for SLA Violation Warehouse (case of Fails packs and ships)

In general, according to the chronicles, there is an agreement that list of products must be equal to the number of products ordered, otherwise there is a violation of the agreement ($E_8$.lp < $E_7$.lp).

We must emphasize that the sub-chronicle Supplier SLA in both cases is identical: "Products search" and "Failure due to pack and ship".

***OpenESB and Intelligent Event Processor (IEP Component)***

OpenESB is a services oriented architecture used to build SOA applications and support their executions; it is a Java-based open source[7] Enterprise Service Bus [22]. The bus provides simplicity, efficiency, to SOA applications, based on the standards JBI

7 Open source are software that can be used, changed, and shared (in modified or unmodified form) freely by anyone

(Java Business Integration), XML, XML Schema, WSDL, BPEL and Composite application. OpenESB defines 2 classes of components: Binding Component (BC)[8] and Services Engine (SE)[9]. Among the services is the IEP component which allows collects, processes, and send events in real time.

IEP uses the CQL language for the event processing [17], providing the combination of streams from different sources, to infer event or pattern to identify situations of interest. The IEP component allows implementing all CQL operators for the events processing. By default, the IEP receives events from a SOAP service interface, and each recognized sub-chronicle is stored in a file called NameServicediagnoser.xml (eg: Supplierdiagnoser.xml ), that can be readed by other components (produces new events), as diagnosers or repair.

To verify the recognition of our distributed chronicles, we implement the application of E-commerce in OpenESB and our chronicle model in IEP module. At the Warehouse service we have added two additional operations to easily induce both SLA faults and to verify the recognition of our chronicles:

- **tuneSearchBehavior:** used to inject the fault *Search product operation (E5).* If it is activated (*tuneSearchBehavior = TRUE*) the operation *"Search product"* does not produce results in search (*return lp = 0*), *otherwise* S*earch product operation* works normally. Thus, an example of this operation is if *tuneSearchBehavior = FALSE* and receives the requirement to search 20 products (*lpin = 20*), *"Search product"* operation returns all products found (*lpout = 20*), conversely if *tuneSearchBehavior = TRUE* and receives the same requirement to search 20 products *(lpin = 20)*, the *"Search product"* operation not find any product (*lpout = 0*).
- **tuneShip:** used to induce the fault in *ship products operation (E8).* If it is activated (*tuneShip = TRUE*) the operation *leaves unpackaged a product* of all required. To show the performance of this operation supposes that *tuneShip = FALSE* and receives the request to pack 20 products (*lpin = 20*), operation returns the packaged of the 20 products (*lpout = 20*), otherwise, supposes that *tuneShip* is changed to *TRUE* and receives the same request to pack 20 products (*lpin = 20*), operation returns only 19 products, leaving a product unpacked (*lpout = 19*).

We run the e-commerce application 9 times, adding an id attribute for each invocation. In Table 2 the results of invocations are shown with the different values used for tuneSearchBehavior and tuneShip.

Table 2. the result of invocations in e-commerce application

| id | tuneSearchBehavior | tuneShip | Operation result (lp) | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | E11 | E12 | E13 | E14 |
| 1 | FALSE | FALSE | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 2 | FALSE | FALSE | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 3 | FALSE | FALSE | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| ***4*** | ***TRUE*** | ***FALSE*** | ***8*** | ***8*** | ***8*** | ***8*** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| 5 | FALSE | FALSE | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 6 | FALSE | FALSE | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 7 | FALSE | FALSE | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| ***8*** | ***FALSE*** | ***TRUE*** | ***22*** | ***22*** | ***22*** | ***22*** | ***22*** | ***22*** | ***22*** | **21** | ***21*** | ***21*** | ***21*** | ***21*** | ***21*** | ***21*** |
| 9 | FALSE | FALSE | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

In table 2 we can see that the values of lp are very well managed with the CQL sentences, in order to recognize the chronicles. In this way, our chronicles are recognized using the CQL sentences. When each sub-chronicle is recognized the event generated is stored in a file (in the case of SLA Supplier the event generated BESLA is stored in Supplierdiagnoser.xml (this is identical in both cases of SLA faults)), and emits event BESLA to sub-chronicle W*arehouse SLA Search* and sub-chronicle *Warehouse SLA Pack and ship*. Sub-chronicle *Warehouse SLA Search* stores generated events in Warehousediagnoser.xml, and emits a fault event to warehouse repair. Sub-chronicle *Warehouse SLA Packs and ships* stores generated events in Warehousediagnoser1.xml, and emits a fault event to o repairer in Warehouse. The contents of the files derived from each sub-chronicle recognition are shown in Fig. 10:

8 Binding Component are used for performing the Communication between applications in a service-oriented architecture (SOA).

9 Services Engine allow to add functionality to facilitate and monitor the processes executed within the BUS


Figure 10. Distributed Chronicle detection to Service Level Agreement (case Search Products and Ship and Packed) in E-commerce application.

As is shown in the Fig. 10, when the operation of the application of e-commerce is normal (id = 1, 2, 3, 5, 6, 7 9) any sub-chronicle is activated. Otherwise the invocation in id = 4 generates the recognition of the failure "search SLA", and the results are stored in Supplierdiagnoser.xml. Additionally, it emits BESLA event, which is readed by the diagnoser warehouse in sub-chronicle Warehouse SLA Search. This last sub-chronicle emits the fault event to repairer in Warehouse (file Warehousediagnoser.xml). Similarly, for the SLA Packs and ships (id = 8) the failure is partially recognized by the supplier and then it emits the BESLA event to warehouse diagnoser, and with this event is possible to recognize the fault (see Warehousediagnoser1.xml file).

***Analysis of results***

The implementation of chronicles using CQL language has provided greater expressiveness to add constraints of atemporal variables. In our case of study could distinguish two faults of Service Level Agreement at the level of the warehouse service in two distinct activities (Search Products and Ship and Packed) using the atemporal variables lp. To express these constraints in CRS and CarDeCRS tools, we should implement additional components (E6 as E6.lp = FALSE in Search Products), and E8.lp = FALSE in ship and pack)). Although we need more implementations of Chronicles in CQL to verify the recognitions of other chronicles, the results are promising because add more expressiveness in the event processing.

The distributed chronicles implemented in CQL Statement detect the faults in the composition of Service Level Agreement violations in both cases (Search Products and Ship and Pack). It is important to mention in this section that the CQL language allows adding mathematical operators in the constraints of atemporal variables in the case of the products amount verification (lp), which is not possible in CRS and CarDeCRS tools without implementing a previous component to introduce this situation as Reifying predicate, such is the case of the 2 SLA chronicles in our case study. To implement the difference in product list (lp) between events using Reifying predicate is necessary to implement a translator indicating whether lp is correct, however the language CQL naturally implements and deploys mathematical operators on attributes constraints (atemporal constraints).

The extension of the formalism of chronicles and the implementation of our CRS (model based on CQL) facilitate the interaction between local Diagnosers, making the recognition of the global chronicle without need a coordinator to manage their interactions. At communication level, this represents a remarkable improvement over the mechanisms shown in other studies [1, 2, 3 , 4, 11 ]. Additionally, the implementation of the mechanism in the cases studied is natural (a recognizer by service). Also, being a distributed approach, the scalability problem of the distributed applications can be handled properly by our approach.

## V. CONCLUSION

We have proposed reflective middleware architecture for autonomic management of service-oriented applications [10]. Our middleware is fully distributed through all services that are part of the SOA application. For this, our architecture has placed a diagnoser in each service, allowing both the diagnostic of faults of services and of compositions. In order to support this architecture, in this paper we have implemented our proposed of distributed mechanism based on chronicles, which allows fully distribute the recognition of the possible faults in SOA applications, which favors its implementation in large systems.

We have extended the formalism of chronicles, introducing the notion of sub-chronicles, binding events, etc. Additionally, we have described the process of recognition of our chronicle fully distributed. Our distributed approach contrasts with the semi-centralized and decentralized approaches that have been developed so far. In the case study, we test the distributed recognition mechanism to detect failures in two situations, at the two level of the warehouse service (Search Products and Ship and Packed). Additionally, we have shown in the case study that the proposed recognition mechanism is simple to implement.

In Fig. 11 we compare the benefits of the implementation of our middleware through various instances of our diagnosers using CQL and IEP component in OpenESB vs the CRS and CarDeCRS tools.

| Tool | Interoperatividad | Distributed | Atemporal variables constraints |
|---|---|---|---|
| **CRS** | NO | NO | Temporal Predicates |
| **CarDeCRS** | Internal protocol | Semi-decentralized | Temporal Predicates |
| **CQL** | Implements a variety of input and output protocols | Fully distributed | Temporal Predicates. Atemporal variables |

Figure 11. IEP vs CRS and CarDeCRS

In our middleware all the services exhibit the ability of interoperability with other components due to the CQL language. Additionally, we see that the CRS and CarDeCRS tools do not permit communication with others, CarDeCRS enables interoperability with other diagnosers, implementing an internal protocol based in semi-decentralized architecture. IEP component allows deploying of instances of our diagnosers using different protocols (SOAP, database, email, ftp, among others). Because IEP supports many protocols, it is easy to implement the distributed architecture of the different diagnosers (fully distributed). In the case studies we have implemented the distributed architecture with message exchange using SOAP and xml files. By contrast, the CRS only allows implementation of a monolithic architecture and CarDeCRS a Semi-Decentralized architecture.

Additionally, CRS and CarDeCRS tools do not allow the implementation of restrictions in atemporal variables, which detracts expressiveness when implementing chronicles. On the other hand, CQL allows to implement atemporal variables, as we have shown in the case studies, and all the classical Temporal Predicates commonly used in chronicles (holds, events, no events and occurs, see table 1).

## VI. ACKNOWLEDGMENT

This work has been supported by FP7-ICT IMAGINE research and development project, funded by the European Commission under the Virtual Factories and Enterprises (FoF-ICT-2011.7.3, Grand Agreement No: 285132) and the PCP program “Supervision and maintenance tasks in a shared organizational environment”.

## REFERENCES

[1] WS-Diamond: WS-Diamond, IST-516933, Deliverable D4.3, 2008. Specification of diagnosis algorithms for Web Services – phase 2. Version 0.5.

[2] Cordier, M, Krivine, J., Laborie, P., Thi´baux, S., 1998. Alarm processing and reconfiguration in power distribution systems. In: Proc. of IEA-AIE’98, pp 230-240.

[3] Cordier, M., Dousson, C., 2000. Alarm driven monitoring based on chronicles. In: Proc. of Safeprocess’2000, pp 286-291.

[4] Quiniou, R., Cordier, M., Carrault, G., Wang, F., 2001. Application of ilp to cardiac arrhythmia characterization for chronicle recognition. In: ILP’2001, pp 220-227.

[5] Guerraz, B., Dousson, C., 2004. Chronicles construction starting from the fault model of the system to diagnose. In: Proc. of the 15th Int. Workshop on Principles of Diagnosis (DX’04), pp 51–56.

[6] Boufaied, A., Subias, A., Combaceau, M., 2004. Distributed fault detection with delays consideration. In: Proc. of the 15th Int. Workshop on Principles of Diagnosis (DX’04).

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

[8] Grosclaude, I., 2004. Model-based monitoring of component-based software systems. In: Proc. of the 15th Int. Workshop on Principles of Diagnosis (DX’04), pp 51–56.

[9] Mhalla, A., Jerbi, N., Collart, S., Craye, E., Benrejeb, M., 2010. Distributed Monitoring Based on Chronicles Recognition for Milk Manufacturing Unit. In: Journal. of Aut. & Syst. Eng. (JASE), Vol. 4(1).

[10] Vizcarrondo, J., Aguilar, J., Exposito, E., Subias, A., 2012. ARMISCOM: Autonomic Reflective MIddleware for management Service COMposition. In: Proceedings of the 4th Global Information Infrastructure and Networking Symposium (GIIS 2012), IEEE Communication Society, Choroni, Venezuela.

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

[12] Dousson, C., Gaborit, P., Ghallab, M., 1993. Situation recognition: representation and algorithms. In: Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI’93), pp 166-172.

[13] Le Guillou, X. , Cordier, M.O. , Robin, S. , Rozé, L., 2008. Chronicles for On-line Diagnosis of Distributed Systems. In Proceeding of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence, pp 194-198.

[14] Morin, B., Debar, H., 2003. Correlation on intrusion: an application of chronicles. In 6th International Conference on recent Advances in Intrusion Detection RAID, Pittsburgh, USA.

[15] Boufaied, A., Subias, A., Combacau, M., 2002. Chronicle modeling by Petri nets for distributed detection of process failures. In: Second IEEE International Conference on Systems, Man and Cybernetics (SMC 2002). IEEE Computer Society Press.

[16] Arasu, A., Babu, S., Widom, J., 2006. The CQL continuous query language: semantic foundations and query execution. In Journal The VLDB Journal - The International Journal on Very Large Data Bases, Volume 15 Issue 2.

[17] Kajic, R., 2010. Evaluation of the Stream Query Language CQL. Institutionen för informationsteknologi, Uppsala universitet.

[18] Salter, D., Jennings, F., 2008. Building SOA-Based Composite Applications Using NetBeans IDE 6. Packt Publishing Ltd., ISBN 978-1-847192-62-2.

[19] Artikis, A., Paliouras, G., Portet, F., Skarlatidis A., 2010. Logic-based representation, reasoning and machine learning for event recognition. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems.

[20] Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C., 2001. Grid Information Services for Distributed Resource Sharing. In: Proceeding of the 10th IEEE International Symposium on High Performance Distributed Computing, pp 181-184.

[21] Lin, A., 2007. Conceptual Model for Business-Oriented Management of Web Services. In Proceedings of the 6th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, pp 80-85.

[22] Ismaili, F., Sisediev, B.: Web services research challenges,limitations and opportunities. In: WSEAS Transactions on Information Science and Applications archive Vol. 5, Issue 10, pp. 1460-1469, 2008.

[23] Keen, M., Hopkins, A., Milinski, S., Nott, C., Robinson, R., Adams, J., Verschueren, P., Acharya, A., 2004. Patterns: Implementing an SOA Using an Enterprise Service Bus. IBM Redbooks publication, First Edition.

[24] Vizcarrondo, J., Aguilar, J., Exposito, E., Subias, A., 2014. Distributed Chronicles to Faults Recognition, Technical report, CEMISID, ULA.