ARMISCOM: Self-Healing Service Composition

Juan Vizcarrondo, José Aguilar, Ernesto Expósito, Audine Subias

The Component of Knowledge Representation of ARMISCOM for the Self-healing in Web Services Composition

Juan Vizcarrondo, Jose Aguilar, Ernesto Expósito, Audine Subias

The Component of Knowledge Representation of ARMISCOM for the Self-healing in Web Services Composition

J. Vizcarrondo, J. Aguilar, E. Exposito, A. Subias

***Abstract*— A previous work has proposed a reflective middleware Architecture for the management of service-oriented applications. Our middleware is designed to be fully distributed through all services of the SOA Application. The architecture uses the model of Autonomic Computing which allow the adaptation of our system, in order to self-healing. Particularly, one of the main aspects of this architecture is the representation of the knowledge. Our architecture uses different paradigms for the representation of the knowledge. For the diagnosis task, it uses chronicles, and for the reparation task it uses ontologies. In this paper, we present the knowledge representation framework, which represents the knowledge needed to perform the different operations of the middleware. Specifically, we design a distributed knowledge based on distributed chronicles, ontologies and other data structures.**

***Index Terms*—Web service fault tolerance, service composition, fault-repair ontology, Distributed Pattern Recognition, Reflective middleware**

I. Introduction

The SOA applications (Service Oriented Architecture) are flexible distributed applications, with loose coupling between these components, based on a software development model composed of small units, called services, which operate in heterogeneous distributed environments. This approach encourages a programming style based on the composition and reuse of services (new applications based on existing services).

The Services are inherently dynamics [1] because they can evolve (their internal calculation, interfaces, among others) and alter its results. Now, in the service composition, a failure of a single service generates an error propagation in the other services, and in this way, the failure of the system. Such failures are very hard to be detected and located, so it is necessary to develop new approaches to enable the diagnosis and correction of the fails, locals (in a service) or global (in the composition)

One of the main aspects to solve in SOA applications is their fault tolerance. For that is required a reparation procedure (self-healing). Repair is to restore the broken functionality, and to return the system at the normal execution [20, 21]. Correction of faults in web services always depends of the type of fault. Web service faults can be classified at three levels [2]: physical, development and interactions; additionally, each fault type has a different repair mechanism.

A previous work has proposed a distributed architecture for the self-healing of faults in the services composition, called ARMISCOM (Autonomic Reflective MIddleware for management of Services COMposition). In ARMISCOM, the fault diagnosis is carried out between the diagnosers present in each service [17]. Similarly, repair strategies are developed through consensus among distributed repair services. In this paper, we present *the* knowledge *representation* component of ARMISCOM, which represents the knowledge needed to perform the different operations of the middleware; specifically, it is the knowledge required by the analyzer and planner components of ARMISCOM.

II. Relate Works

There are two types of failures in web services, the faults in a service, and the faults in the sequence of calls in a composition of services. In [2] is proposed a taxonomy of failures in web services, and describes the perceived effects. In addition, they propose a correlation of the failures and the reparation mechanisms. In [9, 10] propose other classification of Fault types, and define the Recovery action of each one.

At the level of architectures for fault management and recovery of the web services composition, [3] proposes a reflective middleware, called SOAR, which is designed as a centralized structure, in order to monitor and adapt the web application. The middleware has two levels: the first describes the basic characteristics of a SOA system (base level), and the second monitors and adapts the SOA system (meta level). The reflective part of the middleware executes the dynamic binding of web services composition, connecting or disconnecting the services of the SOA application.

In [5] is defined a decentralized architecture that has 2 levels. The first level defines a local diagnoser for each service of the composition. The second level is composed of a global diagnoser, which coordinates the local diagnosers to analyze the

Dr Aguilar has been partially supported by the Prometeo Project of the Ministry of Higher Education, Science, Technology and Innovation of the Republic of Ecuador.

J. Vizcarrondo is with Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (CENDITEL), Mérida – Venezuela. (email: jvizcarrondo@cenditel.gob.ve)

J.L. Aguilar is with CEMISID, Universidad de Los Andes, Mérida, Venezuela. Additionally, it is Prometeo Researcher at the Escuela Politécnica Nacional, Quito, and the Universidad Técnica Particular de Loja, Ecuador (aguilar@ula.ve)

E. Exposito and A. Subias are with CNRS, LAAS, 7, avenue du Colonel Roche, F-31400, Univ de Toulouse, INSA, F-31400, Toulouse - France (email: {ernesto.exposito,subias}@laas.fr})

failures. The global diagnoser also implements the mechanisms for the composition recovery. Each local diagnoser has chronicles that describe the failure patterns, and communicates their instantiations to the global diagnoser. The global diagnoser calculates the sequence of events in the service, to find the occurrence of an error, according to the chronicles instanced by the local diagnosers.

[4] proposes a centralized architecture for web services reparation. The architecture is composed of three modules: a module for monitoring and measuring (it determines the QoS parameters that are relevant), a module of diagnosis and definition of strategies (it detects the degradation of the system and builds the reparation plans), and a module of reconfiguration (it executes the reparation plan). Also, in [6] is proposed other centralized architecture, based on QoS monitoring. Furthermore, in [22] is proposed a structure composed of local diagnosers, which are coordinated by a global diagnoser that executes the repair tasks.

In the context of autonomic computing, MAPE has been used to manage failures in web services [11] providing the ability to self-healing in its invocation (alone web services), but not considering failures derived in its composition with other services. Also, other architecture based on MAPE has been proposed to study the faults on the services composition [12], but this architecture is completely centralized.

Recently, in [17] we proposed a reflective middleware architecture for fault management in service composition, called ARMISCOM, in which each service is overseen by a Local diagnoser using chronicles. To complete the proposal, this paper proposes the knowledge representation component of ARMISCOM. The knowledge representation component is responsible for the management of the knowledge base required by our middleware to carry out its Self-healing task.

The Knowledge representation component of ARMISCOM is composed by distributed chronicles, an ontology to correlate faults and repair methods, and a metadata for storing repair methods available for services within the composition. In previous works, we have designed the distributed chronicles and implemented a mechanism for the recognition of the distributed chronicles using the IEP component in OpenESB and the CQL language [19]. In this paper, we present in detail the design of a distributed ontology in order to correlate the fault type in services with the repair methods, based on [2], which can be used to make inferences about the functional and non-functional properties of the flows in the composition. Additionally, because various repair methods can be applied to solve a given failure, not all can be applied in a given moment because they are not available, is why, in this paper, we also define a distributed data structure for storing the possible repair methods that can be applied at any given time. In this way, this paper presents the design of the component of the distributed Knowledge representation of ARMISCOM for its operation, in order to be used in the self-healing of the web service composition, which contrast with the commonly used mechanisms based on centralized architectures.

III. ARMISCOM ARCHITECTURE

ARMISCOM is a reflective middleware architecture for faults management in the services composition [17]. Reflection is the ability of our middleware to monitor and modify their own behavior, as well aspects of its implementation (syntax, semantics, etc.), allowing the ability to be sensitive to their environment. Thus, ARMISCOM has a dynamic and adaptive behavior, fully distributed, in order to have a closer view of the occurrence of the events that occur in the application. ARMISCOM is divided into two levels, like classic reflective middlewares (see Fig. 1) [17]:

- **Base Level:** A services composition is defined as a set of calculations and interactions of the services that compose a SOA application, with a set of rules that determines these interactions. The base level knows the interactions and its rules in the choreography. In addition, the base level observes both the SOA system and the SOA application. In specific, it monitors the WSDL, UDDI, OWL-S and SCA elements of a SOA system, and uses FraSCAti platform for the intersection process of the services choreography.
- **Meta Level:** it provides the capacity of reflection. It analyses the message exchange between the services that are part of the composition and the components of the SOA system, in order to carry out the introspection. There is a meta level in each service of the choreography.

The implementation of ARMISCON has been designed based on the autonomic computing paradigm. The Autonomic Computing is a computing model inspired on the self-management in the autonomic nervous system of the human [7]. This system is capable of self-administer, for which defines an architecture consisting of 6 levels [7]:

- **Managed Resource:** is any resource of hardware or software.
- **Touch Point:** has the sensor and/or actuator mechanisms.
- **Autonomic Manager:** has the intelligent control loop, with the tasks automate the self-regulation of the applications. The autonomic control loop executes four phases, known as MAPE (Monitoring, Analysis, Planning and Execution). The monitoring phase gets events/data from the sensor interface, the analysis phase is executed by the diagnosers, the planning phase determines how to repair a fault detected, and the execution phase sends the commands to the components via the Touch Point.
- **Orchestrating autonomic managers:** coordinates the Local Autonomic Managers.
- **Manual Manager:** creates the human-computer interface for the autonomic managers.
- **Knowledge Sources:** provides access to the knowledge of the middleware.

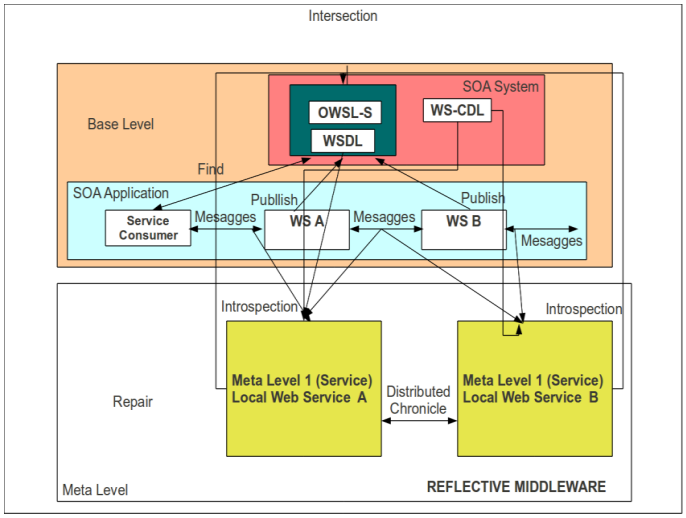

Figure 1. ARMISCOM Architecture

In our case, the *managed resources* and the *touch point* are at the base level; the *autonomic manager*, *the knowledge sources* and *the choreography autonomic manager* are of the meta level (see Fig. 2) [17]. Furthermore, the autonomic manager is composed of three components (diagnoser, repairer and knowledge framework), which are equivalent to the structure MAPE of an autonomic computing architecture. In particular, the diagnoser observes the system and analyzes the failures, and the repairer defines the reparation plans and orders the execution of repair actions.

In ARMISCON each Autonomic Manager works locally (for each service), and through the interaction between autonomic managers is built the diagnosis of failures in the services composition. In particular, the three meta-level modules that composed each autonomic manager are [17]:

- **Diagnoser (Monitor and Analyze):** it inspects the communication services and performs diagnosis. It is invoked by the communication analysis services and has a diagnoser module distributed among the services, to identify the faults (this module is based on chronicles fault patterns).
- **Repairer (Plan and Execute the reparation):** it has mechanisms for the resolution of the fault problems present in the composition of services.

IV. KNOWLEDGE FRAMEWORK COMPONENT

The Knowledge Framework provides the interface to allow the management of knowledge in our middleware. It is composed by (see Fig. 3):

- The SOA System:
  - **Web Services Description Language (WSDL):** It describes how the services can be called, what parameters are expected, and what functionalities are offered.
  - **Web Services Choreography Description Language (WS-CDL):** It describes the Web Services Choreography.
  - **Semantic Markup for Web Services (OWL-S):** It describes semantically the web services using ontologies [8], in order to automate tasks of discovering, invoking, composing, and monitoring of web services.

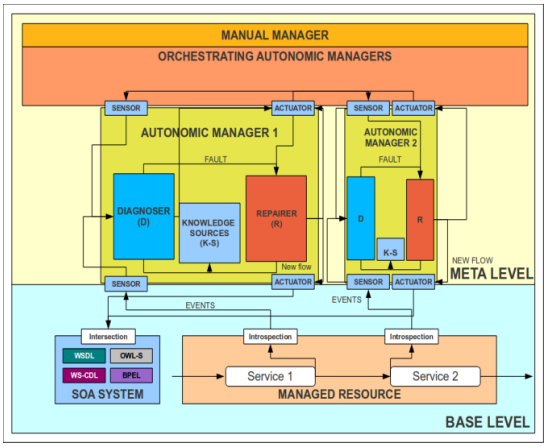

Figure 2. ARMISCON autonomic structure

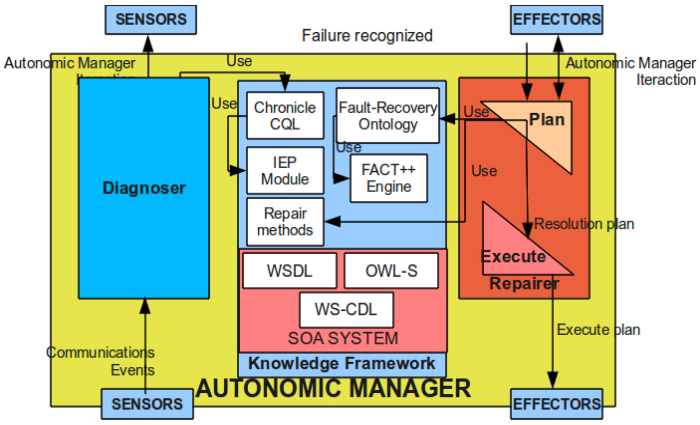

Figure 3. Knowledge Framework components

- **Distributed Chronicles:** It is used mainly by the Diagnoser component. In previous works, we have presented how to represent distributed chronicles, which define the faults, using CQL language [19].
- **Fault-Recovery Ontology:** It is used mainly by the repairer component, in order to define the relationships between the faults and repair methods.
- **Service repair methods:** It is used to store the methods available for the repair service.

A. Distributed Chronicles

In previous work [18, 19], we have designed distributed chronicles in order to specify the different patterns of the faults of the web services. For this, in [18] we have extended the formalism of chronicles, introducing the notion of sub-chronicles, binding events, etc. Furthermore, we have described the process of recognition of our chronicles fully distributed.

Specifically, in [18, 19] we have designed a set of event patterns for recognizing distributed chronicles based on the fault types proposals in [2]. To implement the chronicles we have used the IEP component in OpenESB and the language CQL to define the restrictions between events, in contrast with the tools normally used for recognizing chronicles, as CRS and CarDeCRS. The language CQL allows more expressive by introducing constraints on non-temporal variables [19].

The chronicles are the knowledge about the pattern of behavior of a SOA application when it has a fault. Each chronicle defines a fault type, and it is the knowledge that requires the diagnoser component to detect and diagnose a fault in the application. In [18, 19] are defined the generic patterns (chronicles) for each type of fault defined in [2]. The generic chronicles defined for each fault are:

Physical:

- Unavailable Service Fault

Development:

- Parameters Incompatibility Fault
- Fault due to Interface Might Have Changed
- Fault due to Non-deterministic Actions
- Workflow Inconsistency Fault

Interactions:

- Misunderstood Behaviour Fault (Incorrect Service).
- Response Faults.
- Time-out.
- Misbehaving Execution Flow Fault.
- Incorrect Order.
- Violation of the Service Level Agreement (SLA) and Quality of Service (QoS).

In this way, the knowledge about the behavior of a SOA application with fault is defined using chronicles. Our middleware customizes these generic chronicles according to the specific characteristics of the SOA application supervised

B. Fault-Recovery Ontology

The Fault-Recovery ontology allows correlating faults in the composition of services with available methods for correcting faults in the SOA application. The ontology is the main element of the repairer component, because using it the repairer analyzes the methods of correction of the fault diagnosis by the diagnoser component. The repairer component reasons about the possible methods of corrections of a fault, using the knowledge about that describes in the ontology.

This ontology about the methods the reparation of each fault type in a SOA application is based on the work [2], where they carried out a survey over this topic. The ontology is implemented as a web service that can be accessed by all repairers in our middleware.

Now, we describe the concepts and relationships among them of our ontology. We start describing the concepts of the fault types, then the concepts of the reparation methods, and finally, the generic structure of our ontology where we describe the relationships among the concepts.

C. Concepts about Fault Types in the Web Services Composition

In [2] have described a taxonomy, which classifies the failures in the services composition at three levels: physical, development and interactions. This is the base of our ontology to define the concepts about the fault types.

Physical Faults: Failures are due to the environment where the service (infrastructure) operates and are unrelated to the functionality of the service, causing the service to be considered unavailable (Service or network connection to the service is down). The symptom is that it is not possible to invoke the service (fault due to Unavailable Service).

Development Faults: At the time of conception or development of services and/r composition, may emerge faults that are not considered by the developer of the services and their composition, these types of fault are:

- Parameters Incompatibility Fault: This failure arises when a service is invoked with incorrect values and/or data types of the arguments, with respect to the types and restrictions defined in the WSDL[1] document.
- Fault due to Interface Might Have Changed: The type of data in the interface of some service S_i, which is part of the composition, is modified, so that an incompatibly of parameters is originated when Service S_i is newly invoked. The difference of this fault with respect to parameters incompatibility fault is that the S_i service was previously invoked without failure with the original parameters.
- Fault due to Non-deterministic Actions: This failure occurs when the value of the response of a service is not consistent with the value that should produce the service in the choreography. This kind of failure is extremely rare and is usually because to generate a correct response, the service must previously to invoke another operation in the same service.
- Workflow Inconsistency Fault: In this type of fault the logic in the flow is not correct (Workflow Inconsistency), a service cannot be invoked because its interface does not match the description in the composition. The diagnosis of this type of failure is very complicated, because it is confused with a physical fault (fault due to Unavailable Service).

Interaction Faults: In service composition, interactions occur between services, which can cause faults. In these cases, the types of faults are:

- Misunderstood Behaviour Fault (Incorrect Service): One of the services in the flow of the composition does not produce the expected results. That is not due to that the service does not work properly (it could perform its operations the best possible), but the result is not as expected. To show an example of this, assume that when a service is invoked is expected to return the temperature measured hourly, and the service returns the temperature measured every two hours.
- Response Fault: When the invocation of a service is performed produces a failure in its operation, this may be due to infrastructure problems, authentication or internal logic of the service.
- Time-out: When is described the invocation of a service in the composition, a time period is specified for the response, otherwise a timeout event is generated that allows abort the services composition and avoid other faults in the composition.
- Misbehaving Execution Flow Fault: This fault occurs when a service group or individual service in the composition not yield the expected results in its implementation. They work correctly, but they are not coupled with the other services in the composition, or the result that generate is erroneous within the composition.
- Incorrect Order: Incorrect order failure is because the messages used to interact with the services in the composition arrive in a different order of time than expected.
- Violation of the Service Level Agreement (SLA) and Quality of Service (QoS): Non-functional properties of the services are expressed in terms of QoS and SLA. SLAs are used to describe that capabilities should have the service, and QoS is used to measure the quality of the service based on the response time and quality of the information generated. This fault is generated when the SLA and/or QoS are violated.

D. Concepts about Repair methods in the Services Composition

Once a problem is identified in a services composition, it is necessary to perform a set of actions for the services and/or composition in order to return the system to normal behavior. Thus, different repair methods have been proposed to repair the faults in the composition [22, 23], which are applied depending on the level at which the failure occurs:

Service: These repair methods are applied only at the service level. Some methods of repair of this type are.

- **Retry:** It is applied when a service is temporarily unavailable. In this case, it is suspending the current service execution and the service invocation is retried with known parameters until it becomes available.
- **Substitute a Service:** Is to replace the current service by an equivalent. The compatibility assessment is performed by comparing the interface functionality (WSDL), quality parameters (QoS) and service contracts (SLA).
- **Modify parameters incompatible:** At the time invoking or receiving a service, the message exchanged is incompatible with the definitions of WSDL. The repair involves placing an intermediate service, which is responsible for modifying the input or output messages among the services.
- **Reassign:** This repair method is used when the service does not meet the QoS and/or SLA parameters, the action to take is to reassign the service to a new server to solve the problem. Unlike the substitution of service, this repair method does not seek a new equivalent service, it invokes the same service in a new location.
- **Skip a Service:** Is to jump a service that is part of the composition, which can be running or has not yet been invoked, to continue the execution flow of the composition.

Flow: Is to change the execution flow of the composition.

- **Substitute a Flow:** It is used to faults in some level of the composition. This method consists in replacing part of the flow for equivalent flows, adding new or subtracting services.
- **Redo:** Consists of repeating the invocation of a piece of the flow of the composition, using different parameters taken from previous executions that have worked properly.
- **Alternative behaviors:** is to define an alternative flow to follow the composition in the case of a failure.
- **Skip a flow:** Is to jump a part of the flow in the composition, which can be running or has not yet been invoked, to continue the execution flow of the composition.
- **Change Settings:** Is to change the value of a process variable. This method is used when needed to re- execute a portion of the flow, but using different values of the process variables.

E. Relationships in our Fault-Recovery Ontology

The design of our ontology contains two classes, called Fault and Repair Strategies (see Fig. 4), which represent the concepts of failure and repair methods described in sections 4.A and 4.B. Thus, the Fault class has a property called Has_repair_method, which allows us to assign elements of the class Repair Strategies to each type of fault. In this way are matched the failures in the services composition with the mechanisms to solve the faults. It is a superclass of the classes Physical, Development and Interaction. Also, the class Repair Strategies has a property, called Solve_fault, which performs the inverse operation to Has_repair_method, and it is a superclass of the classes Service and Flow.

The individual instances developed in our ontology are shown in Table I. Repair methods for each failure shown in Table I should be taken as a possible set of actions to run to solve the fault, this selection should be done sequentially among the methods available on site. That is, the repair component must try to solve the fault with the first action available (best case), and if with this one is not possible to solve the problem, it continues sequentially with the next action, until repair the fault or reach the last option (worst case). For example, for the failure of unavailable service, the first action is to try to place the service again available (redo service (best case)), in case it cannot be performed, the second action is to try reassign service on another site, if it cannot solve the problem, it is necessary to try the service substitution by an equivalent, and so until repair the fault or test the last action (skipFlow service (worst case)).

We have implemented our ontology using the protégé[1] tool, which is based on the Web Ontology Language (OWL). Subsequently, the repair component of ARMISCOM invokes a service, which reasons and makes inferences about the repair mechanisms according to the failures present in the composition, using our ontology and the inference motor FACT++[2] of protegé.

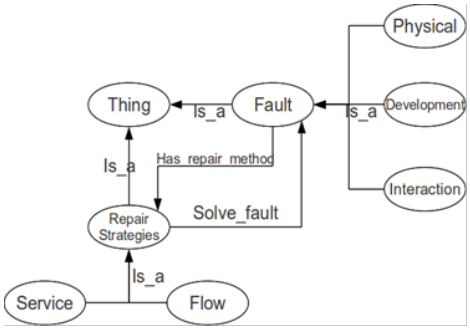

Figure 4. Fault-Recovery ontology structure

Thus, this part of the Knowledge component is implemented by a web service which uses our ontology and the reasoner FACT++. The relationships between the concepts of faults and repair methods in the ontology generated in protegé are shown in Fig. 5

F. Setting the Services of repair methods (Metadata about repair methods)

As shown in the previously proposed ontology, a fault in the composition may have different repair methods. Although some resolution mechanisms can be setting in real time as redo, parametersUpdate, skip service, etc., others need to be previously setting. For example, the method "substitute a service" needs previously to identify the equivalent services, using like knowledge base the SOA system (UDDI, WSDL, OWL-S), because search equivalent services takes some time (it cannot be implemented in real time). Additionally, not all correction mechanisms may be used in some cases/sites, then it is necessary to define a knowledge base that allows ARMISCOM chooses the reparation mechanisms for each case/site.

In these cases, it is necessary to define a mechanism that allows the middleware has stored alternative flows for its repair mechanisms, in order to provide a consistent and quick reparation of a SOA application. Distributed repairs in ARMISCOM are continually looking for equivalent services to replace the service that is responsible when there is a malfunction. Get equivalent services often is not an easy task, and in many cases it is necessary to modify the execution flow of the SOA application (add or remove services). Each repair component continuously updates the metadata with new services and equivalent flows. Because in ARMISCOM the component responsible for performing failure analysis conceives the composition as a stream of events, it is necessary to expand the representation of sub-flows as a sequence of events. In this way, a SOA application can be viewed as a sequence of events E, which can be decomposed into sub-regions or sub-flows R_i of events Eac_i:

[1] Protégé is a free, open source ontology editor. It provides a graphic user interface to define ontologies. This application is written in Java.

2 FaCT++ is a tableaux-based reasoner for expressive Description Logics (DL) developed by the University of Manchester, It covers OWL and OWL2 languages.

TABLE I
INDIVIDUAL INSTANCES IN OUR FAULT-RECOVERY ONTOLOGY

SubClass	individual instance	Has_repair_method
physical	unavailable	• redo • reassign • substitute • substituteflow • skipService • skipFlow
development	parameterIncompatibility	• CompleteMissingParameters • substitute • substituteflow • skipService • skipFlow
	interfaceMightHaveChanged	• CompleteMissingParameters • substitute • substituteflow • skipService • skipFlow
	DueToNonDeterministicActions	• parametersUpdate • substitute • substituteflow • skipService • skipFlow
	workflowinconsistency	• substituteflow • skipFlow
interaction	misunderstoodBehaviourFault	• parametersUpdate • substituteflow • skipFlow
	responsefault	• substitute • substituteflow • skipService • skipFlow
	timeout	• reasign • retry • substitute • substituteflow • skipService • skipFlow
	misbehavingExecutionFlow	• redo • substituteflow • skipFlow
	IncorrectOrder	• substituteflow • skipFlow
	QualityOfService	• reasign • substitute • substituteflow
	ServiceLevelAgreement	• parametersUpdate • reasign • substitute • substituteflow

$$Application(E) = UNION_{i=1,\,n}(R_i(Eac_i)) \quad (1)$$

Where,

- Eac_i are a set of events, such that $Eac_i = \{E_k,,..., E_l\}$ occur in the region i.
- UNION is a predicate that defines the union of distributed events (Eac_i) in the n regions.

Suppose the SOA application shown in Fig 6. This application can be decomposed into regions associated with event services, such that:

$$Application = UNION\{R_1(E_1), R_2(E_2, E_3, E_9), R_3(E_4, E_5), R_4(E_6, E_7), R_5(E_{10}, E_{11}), R_6(E_8, E_{12}, E_{13})\} \mid \forall k,m < 13 \; y \; \forall i,j < 6, \; i \neq j, \; E_k \in R_i \; y \; E_m \in R_j, \text{ then, } R_i(E_k) \cap R_j(E_m) = \emptyset \quad (2)$$

Based on regions R_1, R_2, R_3, R_4, R_5, R_6, it is possible to find equivalent regions R'_1, R'_2, R'_3, R'_4, R'_5, R'_6. Thus, to manage the equivalent regions of the SOA application within ARMISCOM, we need to define a metadata to store repair mechanisms in each case. Repairing a flow of the composition is to find an equivalent region that allows mapping the initial event E_0 and final E_F, that is, one must know the stored equivalent regions related to each repair mechanism.

For that, in ARMISCOM is defined a metadata for each service with the repair methods that can be used in equivalent region (see Table II). In Table II, each attribute is defined as:

- **Weight:** Represents the order in which methods should be extracted, it can be defined based on some kind of optimization.
- **RepairMethod:** Represents the method of reparation available.
- **Flow:** defines the sequence of events (flow) which are affected during the reparation.
- **Flow_init:** Represents the first event on the services composition, in the which should begin the reparation.
- **Flow_end:** Represents the last event on the services composition, in the which should be completed the reparation.

With this metadata, ARMISCOM can define the repair methods available for each case/site. The metadata works as follows: suppose that is necessary to implement the repair method "substitute flow" from event 5 until event 9, then we need to perform the next search: WHERE RepairMethod = "substitute flow" AND Flow_init = 5 AND Flow_end = 9, return data BY Weight. Additionally, because the query could not find any method for the desired flow to modify, the repair component could perform a new search based on a new flow (Eg: Flow_init = 4 and Flow_end = 9 and the same method "substitute flow" describe.

TABLE II
METADATA FOR EACH SERVICE WITH THE REPAIR METHODS

Reparation methods available in a site (service)				
Weight	**RepairMethod**	**Flow**	**Flow_init**	**Flow_end**

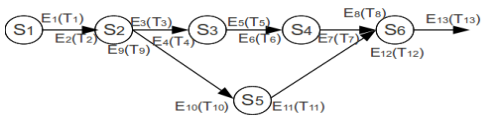

Figure 6. SOA application decomposed into events region

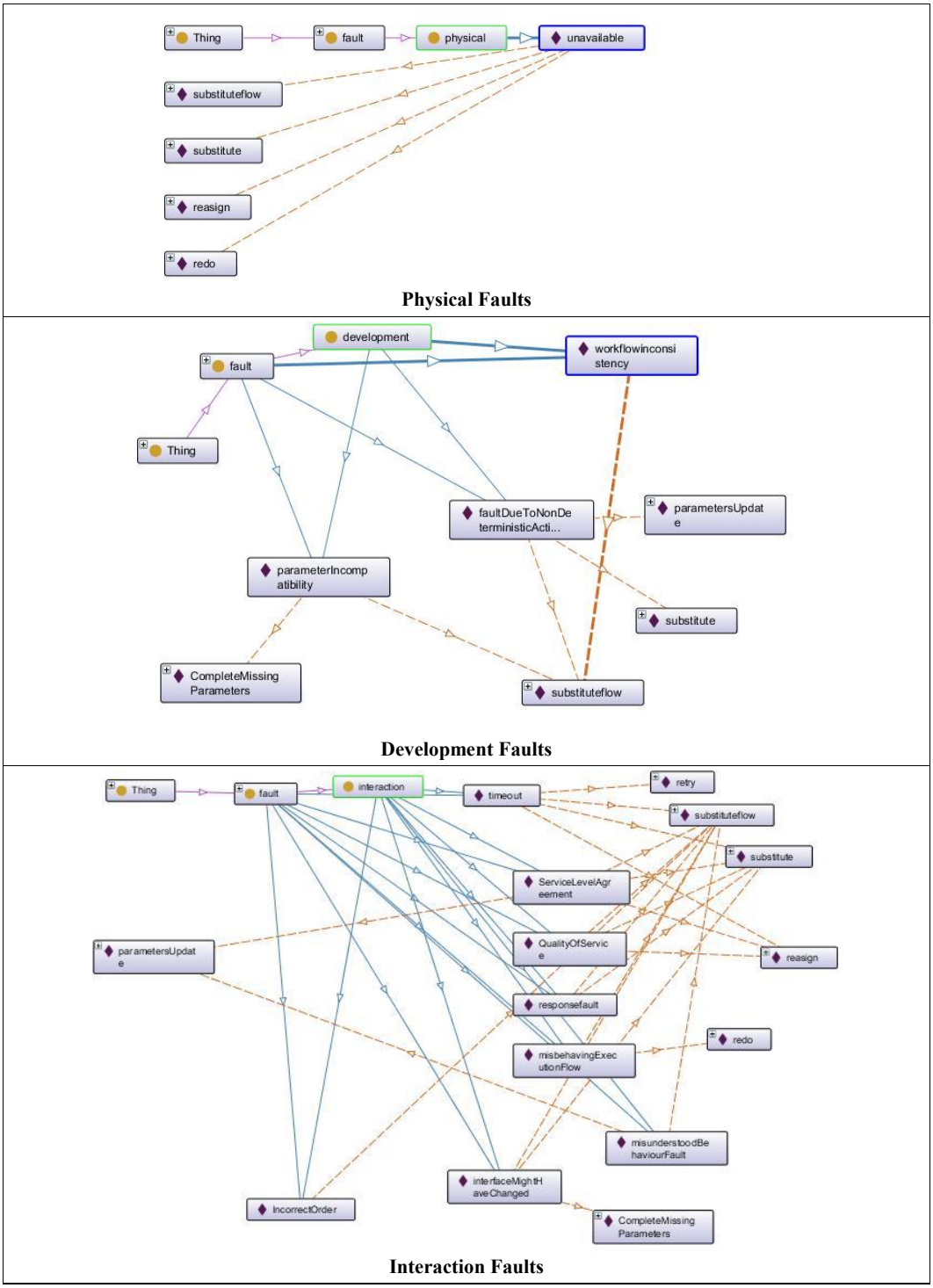

Figure 5. Relationships among the concepts in our Fault-Recovery ontology.

V. CASE STUDY

In this test case we will use a common example of e-commerce SOA implementation (see Fig. 7), which comprises three business processes (which will constitute our services):

- **Shop:** it is the place where users purchase products.
- **Supplier:** it offers products to the shop, but needs to verify their availability before to response.
- **Warehouse:** it is the place where the products are stored by the providers. This service has a service level agreement (SLA)[3] with Supplier, which is that at least one product from the list should be returned[4]. It can interact with other warehouses of the company, in order to search products. In this way, it can answer with at least one product, when it has not in the local warehouse.

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 list of products in the deposit to the Supplier.
(4) **SuppListIn:** The Supplier informs the products that can provide to the Shop.

A. Some elements of the knowledge component of ARMISCOM in this case

In the case of chronicles, Fig. 8 and Table III define the distribution of the events among the diagnosers (sites) of the composition, which is a generic chronicle for this application (connecting all events that may occur in it). With this generic chronicle, can be built the specific chronicles to detect each abnormal situation.

Based on the patterns of the generic chronicles for the different types of faults of a SOA application proposed in [18, 19], the knowledge component builds the specific distributed chronicle for each fault: Quality of Service, Timeout, etc. One example of one of these chronicles is shown in Table IV in the cases of Timeout and Quality of Service.

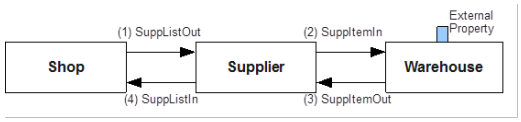

Figure 7. Example of choreography (e-commerce).

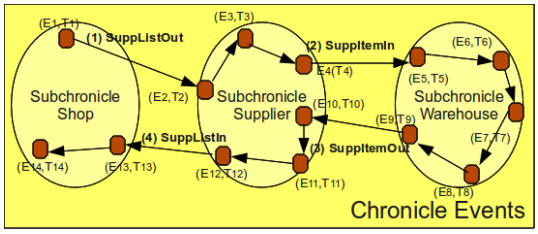

Figure 8. Sequence event divided by diagnoser in E-Commerce case.

TABLE III
EVENT DESCRIPTION DIVIDED BY DIAGNOSER IN E-COMMERCE CASE

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

Tmeout:

- **Subchronicle Supplier:**
  - **Input:**
    - ***E_4*** is an event that is maintained by ***15000 ms*** and ***$E_{NOEVENT}$:*** is a stream produced by the no response from the warehouse. Both **E_4** as **$E_{NOEVENT}$** have no temporal attributes ***id*** (is an identifier used to ensure that the events corresponding to the invocation of the application itself), ***time*** (generated when the event occurs) and ***lp*** (products list) .
  - **Constraint:**
    - The events should have the same id, and the time difference between ***$E_{NOEVENT}$*** and ***E_4*** must be ***5000 ms***.
  - **Output**
    - Emit a bidding event call ***$EB_{Timeout}$*** to Warehouse diagnoser**:**
  - **Subchronicle warehouse:**
    - **Input:**
      - ***E_5***, ***E_6*** and ***E_7*** are events maintained by ***15010***, ***15008*** and ***15006*** respectively; ***$EB_{Timeout}$*** is a stream. All have the same attributes id, time and lp, as in Subchronicle Supplier.
    - **Constraint:**
      - The events must be the same ***id*** and the arrival sequence of the events is established.
    - **Output:**
      - Emit an event to repair, with fault information. To this, we have added additional information to the event, to tell the repairer the name and type (timeout) of the fault, and the affected flow (flow_init = 5 and flow_end = 9, the affected flow are a five services).

[3] SLA is a contract between the service consumer and service provider and define the level of service

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

TABLE IV
DISTRIBUTED CHRONICLES FOR TIMEOUT FAULT AND QUALITYOFSERVICE IN CQL

Distributed Chronicle: Timeout

Subchronicle Supplier Timeout {

```
SELECT
    ISTREAM(
     id => E4.id,
     event => 'EBTimeout',
     time => E10.time,
     lpsupplier => E10.lp,
     lp => E4.lp,
     to => 'Diagnoser warehouse',
    )
FROM
    E4[15000],
    ENOEVENT[now]
WHERE
    ENOEVENT.time >= E4.time + 5000 AND
    ENOEVENT.id = E4.id
}
```

Subchronicle Warehouse Timeout {

```
SELECT
    ISTREAM(
     id => E5.id,
     fault => 'timeout',
     faulttype => 'N/A',
     time => E4.time,
     lp => E4.lp,
     flow_init => 5,
     flow_end => 9,
     to => 'Repair warehouse',
    )
FROM
    E5[15010],
    E6[15008],
    E7[15006],
    EBTimeout[now]
WHERE
    E6.time > E5.time AND
    E7.time – E6.time > 4 AND
    E6.id = E5.id AND
    E7.id = E6.id AND
    EBTimeout..id = E7.id
}
```

Distributed Chronicle: QualityOfService: Delay

Subchronicle Supplier Delay0 {

```
SELECT
    ISTREAM(
     id => E4.id,
     event => 'EBDelay',
     time => E10.time,
     lp => E4.lp,
     to => 'Diagnoser supplier',
    )
FROM
    E4[5500],
    E10[now]
WHERE
    E10.time - E4.time >= 2000 AND
    E10.time - E4.time < 5000 AND
    E10.id = E4.id AND
}
```

Subchronicle Supplier Delay1{

```
SELECT
    ISTREAM(
     id => EBDELAY1.id,
     fault => 'QualityOfService',
     faulttype => 'Delay',
     time => E10.time,
     lp => E4.lp,
     flow_init => 8,
     flow_end => 8,
     to => 'Repair supplier',
    )
FROM
    EBDELAY[15500],
    EBDELAY1[now],
WHERE
    count(EBDELAY.id) + 1 > 2 AND
     EBDELAY.id <>  EBDELAY1.id
}
```

Quality of Service (Delay):

- **Subchronicle Supplier 1:**
  - **Input:**
    - ***E_4*** is an event that is maintained by ***55000 ms*** and ***E_{10}*** is a stream. They have attributes ***id***, ***time*** and ***lp***.
  - **Constraints:**
    - The difference in the time of events ***E_4*** and ***E_{10}*** should be between ***2000*** and ***5000 ms***.
  - **Output:**
    - Emit a bidding event, called ***EB_{Delay}***, to Supplier diagnoser.

- **Subchronicle Supplier 2:**
  - **Input:**
    - ***EB_{Delay}*** is an event maintained by ***15500 ms*** and ***EB_{Delay1}*** is a stream, both have attributes ***id***, time and lp.
  - **Constraint:**
    - The amount of received events must be greater than ***2***
  - **Output:**
    - Emit an event to repair in supplier, with fault information. To this, we have added additional information to the event, to tell the repairer the name and type of the fault (name = Quality Of Service type = Delay), and the affected flow (flow_init = 8 and flow_end = 8, the affected flow is a unique service).

Additionally, the service of the repair methods available at each site, and their metadata, are shown in Table V.

Thus, the supplier only has a repair mechanism (substituteflow) affecting the flow from the event 5 until the event 9 (repair E5, E6, E7, E8 and E9 operations), On the contrary, warehouse has four repair mechanisms: parametersUpdate (repair E6 operation), CompleteMissingParameter (repair E5 operation), substituteflow (repair E8 operation) and substituteflow (repair E7 operation).

B. Testing the e-commerce application using the Knowledge Component

To verify the operation of component of knowledge of ARMISCOM, we implement the application of E-commerce in OpenESB and connected the distributed diagnoser and repair modules. At the Warehouse service we have added one additional operation to easily induce delay faults and to verify its full operation:

setTuneDelay: Used to induce delay time in the warehouse service (initial delay is 0 ms, no delay). Thus, three invocations of the application are performed (id = {1, 2, 3}) where TuneDelay is setting with a delay of 3000 ms (induces multiple delay fault). Subsequently is invoked again the warehouse service (id = 4) with a TuneDelay of 6000ms what would cause a timeout in e-commerce application. The results are shown in Table VI.

TABLE V
AVAILABLE METHODS TO REPAIR E-COMMERCE APPLICATION

reparation methods available in Supplier				
Weight	**RepairMethod**	**Flow**	**Flow_init**	**Flow_end**
1	substituteflow	E6, E7, E8, E9	5	9
reparation methods available in Warehouse				
Weight	**RepairMethod**	**Flow**	**Flow_init**	**Flow_end**
1	parametersUpdate	E6	6	6
1	CompleteMissingParameter	E5	5	5
1	substituteflow	E8	8	8
1	substituteflow	E7	7	7

TABLE VI
KNOWLEDGE SOURCE USED IN QUALITY OF SERVICE (DELAY) AND TIMEOUT FAULTS

Fault	**Distributed Chronicle Diagnosis Response**	**Fault-Recovery Ontology Response**	**Service repair Selection Response**
Quality Of Service (Delay)	<?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput4_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>3</id> <fault>QualityOfService</fault> <faulttype>Delay</faulttype> <time>1408573740066</time> <lp>10</lp> <flow_init>8</flow_init> <flow_end>8</flow_end> <Timestamp>2014-08-20T17:59:05.927-04:30</Timestamp> </msgns:StreamOutput4_MsgObj>	<?xml version="1.0" encoding="utf-8"?> <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope"> <S:body> <ns2:getRepairMethodResponse xmlns:ns2="http://ws/"> <return>reassign;substitute;substituteFlow</return> </S:body> </S:Envelope>	<?xml version="1.0" encoding="utf-8"?> <msgrepair:supplier> <methodrepair>substituteflow</methodrepair> <flow_init>8</flow_init> <flow_end>8</flow_end> </msgrepair:supplier>
Timeout	<?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput2_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>4</id> <fault>timeout</fault> <faulttype>N/A</faulttype> <time>1408573625710</time> <lp>2</lp> <flow_init>5</flow_init> <flow_end>9</flow_end> <Timestamp>2014-08-20T17:57:06.025-04:30</Timestamp> </msgns:StreamOutput2_MsgObj>	<?xml version="1.0" encoding="utf-8"?> <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope"> <S:body> <ns2:getRepairMethodResponse xmlns:ns2="http://ws/"> <return>reassign;retrysubstitute;substituteflow;skipService;skipFlow</return> </S:body> </S:Envelope>	<?xml version="1.0" encoding="utf-8"?> <msgrepair:supplier> <methodrepair>substituteflow</methodrepair> <flow_init>5</flow_init> <flow_end>9</flow_end> </msgrepair:supplier>

As shown in Table V, ARMISCOM was able to diagnose and correct Quality of Service (Delay) and Timeout faults. In the case of Quality of Service (Delay), the supplier diagnoser recognizes chronicle and emits the event to its repairer (fault: QualityOfService, fault type: Delay, flow_init = 8 and flow_end = 8, see first column). With this information the repairer performs inference in the Fault-Recovery ontology for the QualityOfService fault, and returns the possible solution methods to be implemented to correct the fault (reassign, substitute and substituteFlow, see second column). Then, the repair performs the search in metadata: first it searches method reassign, after substitute, and because they are not implemented, subsequently seeks substituteFlow with flow_init = 8 and flow_end = 8 (taken from Distributed Chronicle Diagnosis Response). This one is available to be applied like method to solve the fault. The diagnosis and correction of the Timeout fault is similar. First, the Warehouse Diagnoser recognizes the chronicle and emits the event to its repairer (fault: timeout, fault type: N/A, flow_init = 5 and flow_end = 9, see first column). The repair carries out an inference about the fault in the Fault-Recovery ontology, and returns the possible methods to implement (reassign, retrysubstitute, substituteflow, skipService and skipFlow). Finally, it performs a search in the metadata of the Warehouse repairer to find possible repair mechanisms to implement, the repair mechanism "substituteflow", for flow: flow_init = 5 and flow_end = 9, is the only one available.

C. *Results Analysis*

In the case study, we observe how the knowledge component of ARMISCOM uses hybrid knowledge to manage the different aspects necessary to guarantee the fault tolerance of a SOA application. The different patterns of distributed chronicle are used to diagnose the failures (in this case, we have shown the Quality Of Service (Delay) and Timeout chronicles). When a distributed chronicle is recognized the diagnoser produces a file with the diagnosis, which is read by the repair component. This component uses the Fault-Recovery Ontology to reason about the repair methods that could be used to solve the fault. Finally, with the identification of the part that has been affected (event_init and event_end) and the repair mechanisms stored in the metadata, ARMISCOM can get the best available method to solve the fault in real time.

Our extension of the formalism of chronicles, facilitates the interactions between local diagnosers, without need of a coordinator to manage their interactions. This represents a remarkable improvement in communication and scalability level, with respect to previous studies [13, 14, 15, 16]. In addition, its implementation is very natural in the case study (a recognizer by service).

Some works store subflows modeling them as a set of services that are interconnected with each other, using Petri nets or graphs connections [3, 4, 6, 12, 22, 24, 25, 26, 27, 28]. Additionally, they replace sub-flows in the composition of services, have architectures that allow them to previously find alternate sub-flows, to respond to faults present in the composition in real time. Thus, the mechanisms consist of modeling a SOA application as a graph or path, which can be decomposed into sub-graphs, and achieve equivalent flows based on a similarity criterion, according to the functional and non-functional (e.g. QoS).

In this work, flows have been modeled as events with time constraints, to be in line with the chronicle paradigm. Additionally, the metadata can store the information that characterizes the regions of the events (Initial Flow Event (Event_init), sequence of events that compose it (Transition), Final Event Flow (Event_end)), which defines the region where must be applied the repair strategy (RepairMethod), and determines the equivalent regions. All this information can be used to select the best option, in order to be effective when a service must be replaced.

Additionally, we have designed a repair module that allows us to infer the repair strategies for failures in the services composition, taking into account context information based in the fault and in the flow composition problem, which is performed at runtime. In previous words [9, 10] have correlated recovery actions with fault type, in our case we use a fault-recovery ontology to correlate the faults with the recovery actions, which was implemented as a web service using BC Sun Java EE SE, to encapsulate the JAVA language as a service, and the inference engine FACT++. The various queries performed at service ontology for each failure showed the expected response (repair methods to use) in the reparations. This ontology can increase (e.g. using ontological learning approaches) to include new faults, reparation mechanisms, etc. Also, the metadata provides to ARMISCOM multiple recovery plans, to address the flow fault in the composition of web services. The case study showed how to store different repair mechanisms and to make the request to the meta-data, in order to find the mechanisms best suited to the part affected (which failed). In this way, ARMISCOM can be customized very easily, because can deduce the appropriate repair method to be used for each case.

VI. CONCLUSIONS

We have proposed a reflective middleware architecture for autonomic management of service-oriented applications [17]. ARMISCOM is fully distributed through the services of the SOA application, it is instanced in each service, for both the diagnosis and the reparation of faults of services and of compositions. In order to support this architecture, in this paper, we have designed the knowledge management component of our middleware. This Knowledge is composed of the information from SOA system, of Distributed chronicles which describe the behavior of a SOA application with failures, the distributed metadata which describes the repair methods, and of a Fault-Recovery Ontology.

In the case of distributed chronicles, previously, in [18], we have extended the formalism of chronicles, with the definition of the notion of sub-chronicles, binding events, among others. Our extension contrasts with the semi-centralized and decentralized chronicle approaches that have been developed previously.

Additionally, chronicles make possible to identify the parts affected by the faults, adding new attributes to the events as fault name, fault type, part of the flow affected by the failure (flow_init and flow_end). With this information, in this paper,

we have described as ARMISCOM determines the equivalent regions, which are sub-flows as events with time constraints. In this way, ARMISCOM can characterize regions with fails to be replaced, which defines the region where must be applied the repair strategy (RepairMethod),

In the case of the Fault-Recovery Ontology component, it has been implemented as a web service, allowing correlated faults present in the composition with repair mechanisms using an inference motor. The Fault-Recovery Ontology component has been developed as an ontology composed of super-classes, classes, properties and individuals using OWL language, which describe a taxonomy about the mechanisms of reparation of faults for a SOA application. This ontology can be enriched in the future to allow inferences about the more complex situations, using functional and non-functional properties of services.

Finally, we have proposed a metadata about each repair methods available at each site, which must be used by the repair component. Using this metadata the repairer deduces the appropriate repair method for each case. Metadata provides representation of multiple recovery plans available at different instances of flows (web services) of the composition of services, using the concept of equivalent regions, which allows to calculate the suitable plan to implement in the case of a fault.

Our middleware requires a knowledge component which manages hybrid knowledge, in order to properly infer the portion of the flow that has failed and find the closest resolution mechanism. This architecture for autonomic management of service-oriented applications is based on hybrid knowledge, according to the needs of each MAPE component. The utilization of the hybrid knowledge (different sources of knowledge) defined in this paper, is one of the advantages of our approach. Additionally, the component of the distributed Knowledge representation designed in this paper, allow the self-healing web service composition fully distributed, representing another significant improvement, in order to reduce the large exchange of messages and to minimize the calculation required in the diagnosis and the reparation, which are the main problems of the centralized approaches [3, 4, 6, 12, 22, 25, 26, 27, 28].

Some improvements are possible. For example, the metadata are defined by an expert. However, this task could be delegated to another component that automatically build it. An example is to use another ontology to infer services and flow equivalences, this would work as a robot that is continuously running and updating the metadata, which can be enriched using the weight field for indicating the degree of equivalence. Also, the ontology can be extended to describe features that allow to infer the services of reparation more exactly

REFERENCES

[1] Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C.: "Grid Information Services for Distributed Resource Sharing". *10th IEEE International Symposium on High Performance Distributed Computing,* pp. 181--184. 2001.

[2] Chan, K., Bishop, J., Steyny, J.,Baresi, L., and Guinea, S.:, "A Fault Taxonomy for Web Service Composition", *Service-Oriented Computing Workshop*, pp. 363-375, 2007.

[3] Huang, G., Liu, X., and Mei, H.: "SOAR: Towards Dependable Service-Oriented Architecture via Reflective Middleware". *International Journal of Simulation and Process Modelling,* vol. 3, no. 1/2, pp. 55-65, 2007.

[4] R. Halima, E. Fki, K. Drira and M. Jmaiel, "Experiments results and large scale measurement data for web services performance assessment". *IEEE Symposium on Computers and Communications*, pp. 83-88, 2009.

[5] WS-Diamond project, "WS-Diamond, IST-516933, Deliverable D4.3, Specification of diagnosis algorithms for Web Services – phase 2", http://wsdiamond.di.unito.it/.

[6] Poonguzhali, S.,, Sunitha, R., and Aghila, G.: "Self-Healing in Dynamic Web Service Composition". *International Journal on Computer Science and Engineering*, vol. 3, no. 5. pp. 2054-2060, 2011.

[7] IBM Corporation. "An architectural blueprint for autonomic computing". Autonomic Computing", Fourth Edition, http://www.ginkgo-networks.com/IMG/pdf/AC_Blueprint_White_Paper_V7.pdf, 2006.

[8] Chiribuca, D., Hunyadi, D. and Popa, E.: "The Educational Semantic Web", *8th WSEAS International Conference on Applied Informatics and Communications*, pp. 314-319, 2008.

[9] Fugini, M.G., Mussi, E.: Recovery of Faulty Web Applications through Service Discovery. *32nd International Conference on Very Large Databases*, pp. 67-80, 2006.

[10] Ardagna, D., Cappiello, C., Fugini, M., Mussi, E., Pernici, B., andPlebani, P.: Faults and recovery actions for self-healing web services. *15th Int. World Wide Web Conf*., 2006.

[11] Sherif, A.; and Amir, Z.: Towards autonomic web services: achieving self-healing using web services. *2005 Workshop on Design and evolution of autonomic application software*, Pages 1 – 5, 2005.

[12] Poonguzhali1, S.; JerlinRubini, L.; Divya, S.: "A Self-Healing Approach for Service Unavailability in Dynamic Web Service Composition". *International Journal of Computer Science and Information Technologies*, vol. 5 Issue 3, p 4381, 2014.

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

[14] Cordier, M.O., Krivine, J., Laborie, P., Thi´ baux, S.: "Alarm processing and reconfiguration in power distribution systems". *IEA-AIE'98*. pp. 230–240, 1998.

[15] Cordier, M.O., Dousson, C.: "Alarm driven monitoring based on chronicles". *Safeprocess'200*0. Pp 286–291, 2000.

[16] Quiniou, R., Cordier, M.O., Carrault, G., Wang, F.: "Application of ilp to cardiac arrhythmia characterization for chronicle recognition". *ILP'2001*. pp. 220–227, 2001.

[17] Vizcarrondo, J., Aguilar, J., Exposito, E., Subias, A.: "ARMISCOM: Autonomic Reflective MIddleware for management Service COMposition". *4th Global Information Infrastructure and Networking Symposium (GIIS 2012),* IEEE Communication Society, 2012.

[18] Vizcarrondo, J., Aguilar, J., Exposito, E., Subias, A.: "Crónicas Distribuidas para el Reconocimiento de Fallas", *Revista Ciencia e Ingeniería.* vol. 36, no. 2, pp. 73-84, 2015.

[19] Vizcarrondo, J., Aguilar, J., Exposito, E., Subias, A.: "Building Distributed Chronicles for Fault Diagnostic in Distributed Systems using Continuous Query Language (CQL)", *International Journal of Engineering Development and Research (IJEDR),* vol.3, no. 1, pp.131-144, 2015

[20] Aguilar, J. "An artificial immune system for fault detection", *Intl. Conf. on Industrial, Engineering and other Applications of Applied Intelligent Systems*, pp. 219-228, 2004.

[21] Aguilar, J., Hernández, M. "Fault tolerance protocols for parallel programs based on tasks replication", *8th Intl Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 397-404, 2000.

[22] Ardissono L., Console L., Goy A., Petrone G., Picardi G., Segnan M, "Enhancing Web Services with Diagnostic Capabilities". *Third European Conference on Web Services*, pp. 182-191, 2005.

[23] Fugini ,M. Mussi G: "Recovery of Faulty Web Applications through Service Discovery". *32nd International Conference on Very Large Databases*, pp. 67-80, 2006.

[24] WS-Diamond: WS-Diamond, IST-516933, Deliverable D5.1, Characterization of diagnosability and repairability for self-healing Web Services, 2005.

[25] Feng X., Wang H., Wu Q., Zhou B, "An adaptive algorithm for failure recovery during dynamic service composition," in *Pattern Recognition*

and Machine Intelligence (A. Ghosh, R. De, and S. Pal, Eds). Springer Berlin / Heidelberg, vol. 4815, pp. 41-48, 2007.

[26] Feng X., Wu Q., Wang H., Ren Y., Guo C, "ZebraX: A model for service composition with multiple QoS constraints", In *Advances in Grid and Pervasive Computing* (C. Cerin, K.-C. Li, Eds.), Springer Berlin/Heidelberg, vol. 4459, pp 614-626, 2007.

[27] Canfora G., Di Penta M.., Esposito R., Villani M, "A framework for QoS-aware binding and re-binding of composite web services", *Journal of Systems and Software*, vol. 81, pp. 1754-1769, October 2008.

[28] Saboohi P., Amini A., Abolhassani H., "Failure recovery of composite semantic web services using subgraph replacement,, *International Conference on Computer and Communication Engineering (ICCCE),* pp. 489-493, 2008.

Juan Vizcarrondo is System Engineer, and obtained a Msc in Computer Science at the Universidad de los Andes, Mérida-Venezuela, and a PhD in Computer Science at the Universidad de los Andes. He works at the Cenditel since 2007.

Jose Aguilar is a System Engineer graduated in 1987 from the Universidad de los Andes, Merida, Venezuela. M. Sc. degree in Computer Sciences in 1991 from the University Paul Sabatier-Toulouse-France. Ph. D degree in Computer Sciences in 1995 from the University Rene Descartes-Paris-France.. He completed post-doctorate studies at the University of Houston, researcher at the Microcomputer and Distributed Systems Center (CEMISID) at the same university. Member of the Mérida Science Academy and the International Technical Committee of the IEEE-CIS on Artificial Neural Network.

Ernesto Exposito earned his engineer degree in computer science from the "Universidad Centro-occidental Lisandro Alvarado" (Venezuela, 1994). He earned his PhD in “Informatique et Télécommunications” from the Institut National Polytechnique de Toulouse (France, 2003). He is Professor in computer sciences at the Institut National des Sciences Appliquées (INSA) of Toulouse.

Audine Subias received a PhD degree in 1995 and a M.S.degree in 1992 in Informatique Industrielle, both from Paul Sabatier University, in Toulouse, France. Since 1997 she is Associate Professor in control and discrete event systems at the Institut National des Sciences Appliquées (INSA) of Toulouse.