Rule-based Monitoring and Error Detecting for Converged Telecommunication Processes

Armando Ordóñez, Luis Eraso
Intelligent Mangement System Group
University Foundation of Popayán
Popayán, Colombia
{jaordonez,luise}@unicauca.edu.co

Paolo Falcarin
School of Architecture, Computing and Engineering
University of East London
London, UK
falcarin@uel.ac.uk

***Abstract*—Convergent process may be defined as a composition of services which integrates functionalities from Web and Telecommunication domains. However, during execution of convergent processes some services may fail; in such cases a reconfiguration process must be triggered to recover normal behaviour of composite process. Previous works have developed mechanisms for reducing reconfiguration time while initial restrictions are maintained; this is achieved by replacing regions of services instead of individual services. The present work presents an approach for monitoring and error detecting in convergent processes using rule production systems based on ITIL model. The approach was tested in the monitoring module of the AUTO framework, whose architecture and performance are discussed to show that this approach can efficiently detect errors and repair convergent processes in telecom environments.**

***Keywords*—Service monitoring; automated reconfiguration; automated planning; convergent services; service composition**

I. Introduction and Background

Convergent process may be defined as a structured set of services (Telecommunication and Web services) that works in a coordinated manner to achieve a common goal [1]. One example of convergent process is a service that manages environmental early warnings (see Fig. 1). Environmental manager is in charge of decision making about environmental alarms and crops. To do so, the manager has information from sensor networks and can also use Telecommunication and Web services to process basic data and send information to both farmers and sensors. Some typical requirements of such systems are: i) to calculate hydrological balance of the zone and receive the resulting map to the mobile, and ii) to emit an alarm to every farmer within a radius of 2 miles from the river if the river flow is greater than 15% of average. For the first request, the system must gather information from sensors, and then the system uses hydrological services from the Internet, sending sensor data and maps fetched from Google maps.

Finally the resulting image is sent via MMS to the user's mobile device. In the second request, sensor data are evaluated. If necessary, an emergency map is generated. This map is created drawing a radius of 2 miles from the sensor. To do so, the system uses geographic services and maps from internet. Finally, the system informs about the alarm to farmers inside the emergency area; the best way to send the information is selected: SMS, Cell Phone call, fixed telephone call, voice message. In both cases, services from Web and Telecommunications are used: these services work together and in coordination to save lives or help to make decisions about crops.

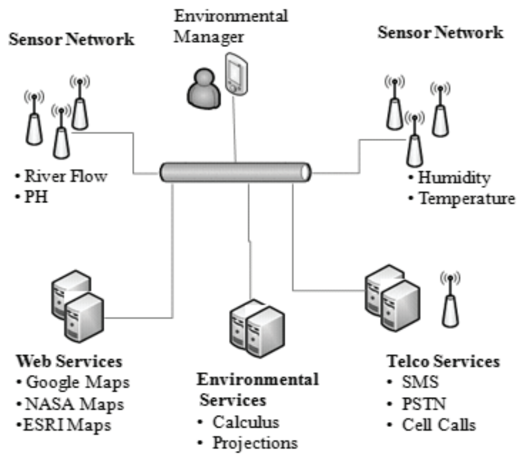

Fig. 1. Telecommunication and web services interaction in environmental management systems

Due to the dynamic nature of the Internet, Web Services may change, become unavailable and grow in number to unmanageable size. This means that manual synthesis and reconfiguration of convergent processes are unfeasible in practice and consequently automation of such tasks becomes necessary [2].

Identification and definition of guidelines for manual composition of convergent services are problems that have been investigated for years [3]; more recently, automation of different phases of convergent composition has been actively researched. Specifically, a framework called AUTO is presented in [4][5][6], which aims at supporting automated composition of convergent services using automated planning for service composition and natural language processing for user request processing.

Among phases for automated composition, reconfiguration has been identified as one of the leading challenges in Service oriented architectures [7][8][9]. Particularly, in the Telecom industry, high reliability is a crucial factor and the reconfiguration process must be as much transparent as possible. However, due to the fact that reconfiguration is time-consuming (the optimal service selection problem is NP-hard)

978-1-4673-7606-8/15/$31.00 ©2015 IEEE

[10], the number of halts and reconfigurations of the whole process must be minimized.

Previous works tackle service reconfiguration issue by replacing failing services, and consequently, most of the existing approaches are focused on service selection of potential replacement [11][12]. However, existing proposals are focused exclusively on Web Services, ignoring typical non-functional features of Telecom services, such as service deployment, real-time requirements, and event-based interactions. Sometimes replacing a failing service with one with different non-functional features prevents that initial user constraints are maintained. One interesting approach in this regard is CHOReOS project (FP7-ICT-2009-5) "Large Scale Choreographies for the Future Internet (IP)" that aims for elaborating new methods and tools related to Future Internet ultra-large-scale (ULS) [13]. CHOReOS includes the study mechanism for adaptability and QoS-awareness based on monitoring choreography-based service composition in service-oriented systems. CHOReOS present a general approach applicable to different scenarios; however, the authors did not consider the Telecommunication domain.

An approach for self-reconfiguration of convergent telecom processes based on automated planning was presented in [14][9]. In this approach, if a failure appears in a service, this faulting service and the subsequent services in the convergent processes were replaced by another set of services considering user preferences during automated composition of the original plan, and during reconfiguration (user situation and context may change). In telecommunication domain, it is not desirable to change the entire process but only the failing services [7]. Besides the new services must be deployed into executable telecom environments, which is usually a time-consuming task.

In this paper an approach based on rules for monitoring and detection of errors during execution of convergent services is presented. This approach can possibly reduce the overhead of repairing a complete faulty service process, since fewer services are involved in reconfiguration. Furthermore the time for undoing the task and recreating the whole convergent process is decreased.

Other projects for reconfiguration have also been reported in literature [10][15][16]. However, these works concentrate on Web service selection of replacements, and the initial requirements of users are not considered. The present work proposes a region-based service reconfiguration that maintains the Quality of Service (QoS) initially defined by user requirements. The present approach is based on automated planning for performing the replacement of failing regions. In this context, the main contributions of this work are: a model of error levels presented in the convergent services architecture, a prototype of the rule based system for error detection in AUTO, and a performance report of the reconfiguration approach.

The rest of this paper is organized as follows: section 2 depicts the system architecture and the main components of AUTO [5]; section 3 presents the reconfiguration architecture and algorithm for reconfiguration of convergent processes, Section 4 depicts the model for error detection, section 5 draws the prototype, section 6 presents related work, section 7 depicts the experimentation, and finally, section 8 draws the conclusions and discusses future work.

II. Overview of AUTO Architecture

This section briefly depicts the components of the architecture for automated convergent composition AUTO (see Fig. 2). A deeper explanation can be found in [5]. AUTO defines a series of sequential phases for automated service composition: *creation*, *synthesis*, *execution and reconfiguration*. The input method can be either by voice or text, which means that AUTO can be accessed from a broad range of devices.

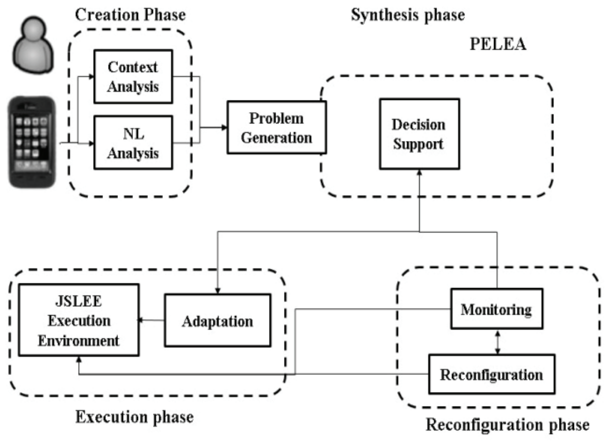

Fig. 2. Architecture of AUTO

The *Creation phase* decomposes the request in constitutive parts: the Natural Language Analysis deals with the treatment of the user request expressed in natural language. In AUTO, the transformation from natural language to planning language is performed by a module composed of a set of underlying components. The functional aspects and implementation of this module has been already described elsewhere [5], so only high-level description is provided here. Request processing in natural language is a set of sequential steps. First, the input is split into tokens, generating simple lexical units from complex sentences. Next, individual units are filtered and tagged according to their grammatical category. Additionally, each token is labelled as either "Control", "Functional" or "Situational" and classified according to three dimensions: device, user and situation. Next, information gathered from User Profile using the user's ID is added to the request. Finally, the request is translated into a planning instance from which a service composition is computed.

These requests as well the context information (user preferences, device capabilities and situational context) are translated into a problem file expressed in Planning Domain Definition Language (PDDL). The automatically generated problem in planning language is the input sent to the *Synthesis phase*, based on the *Planning and Learning Architecture* (PELEA) [17], which performs the synthesis of a plan representing the convergent process using automated planning.

AUTO uses a robust execution environment for telecommunications applications called Java Service Logic Execution Environment (JSLEE). The integration between

PELEA and JSLEE is done in the *Adaptation module*. The adaptation module takes the synthesized plans and creates an executable convergent process. To do so, the adaptation module associates the planning operations to Java Snippets and generates a composite service out of JSLEE Service Building Blocks (SBB). While in other JSLEE solutions [3], web services are invoked through a SOAP resource adaptor, in AUTO instead, SBBs are the basic components of the JSLEE architecture and are responsible of invoking Web services and Telecom functionalities. In situations in which the status and characteristics of the services may change, AUTO can monitor execution of the composite services and also repair the current plan if needed. The reconfiguration process will be explained in detail in the next section

III. AUTOMATED RECONFIGURATION

The execution flow of AUTO tasks within a reconfiguration process is described in Fig. 3, and it is made of 10 phases (see Fig. 3). The *Synthesis* phase creates the *abstract plan* that represents the convergent process ((1) in Fig. 3). However, telecom requirements require the user to be served immediately, so the elapsed time spent building the plan must be minimized. Nevertheless, in order to get the plan that represents the best solution, a big computational effort is necessary. On the average case finding the optimal solution requires exploring a very significant part of the search space, which makes such an effort impractical. Therefore, the best solution in a given time-window may be acceptable to begin the execution, and afterwards the planner may refine the plan while the first plan is executed".

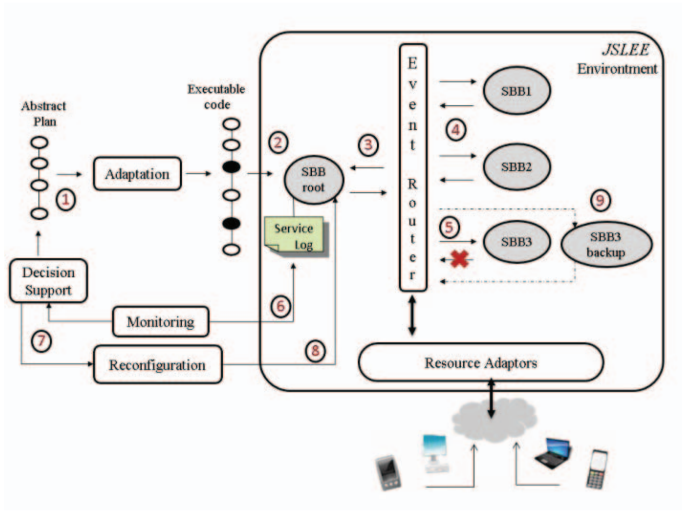

Fig. 3. Reconfiguration schema in AUTO

The additional plans generated after the initial solution comprise the *ranking of alternative plans* that are possibly used later. Later, the *Adaptation* module creates an executable convergent process in *JSLEE* through a *SBB root* (2). *JSLEE* is based on events, so during execution the *SBB root* controls the execution of individual SBB (*SBB1, SBB2* and *SBB3* in Fig. 3) which represent individual services. The communication between SBBs is transmitted through the *Event Router*; similarly the communication between SBB and external networks is transferred by the *Resource adaptors*.

During normal execution, *SBB root* invokes *SBB1* and *SBB2* using events (3), and get the successful response event (4). However, it may happen that *SBB3* does not receive the response event or it may generate an error (5). *SBB root* tracks all the execution results as well as the new "*state of the world*" in the server log (6). The *state of the world* is the set of information described in *planning language* that indicates the preconditions and post-conditions associated with execution of each service. For instance, a payment service changes the state of the world from "not-paid" to "paid", the next section describes in detail the services representation.

The *Monitoring module* gathers information from the execution of the convergent process execution in order to collect necessary data that will be necessary for determining if an error is present. This module gets information from the server log and invokes the decision support module for determining if *SBB3* presents an error. In order to identify if an error is present, the system compare the result of the invocation of a Service with a set of acceptable values specified at design time.

In case of failure, the reconfiguration algorithm is invoked. This algorithm aims for replacing *SBB3* and some surrounded services by other service or set of services that performs the same functionality. The algorithm starts from the current state of the world (represented in *planning language*) and performs a search in the existing services trying to minimize the set of services to be replaced. To do so, *Monitoring module* invokes the *Decision Support module* based on automated planning ((7) in Fig. 3). If the calculated region is very large or if the error is presented in the first service, so other plan is selected from the *ranking of alternative plans*. Regardless if a region or a whole process is changed, the *Reconfiguration module* performs the changes in the Server using Javassist ((8) in Fig. 3), this approach is described deeply in [18]. Finally, the replaced services or set of services replacing the functionality of *SBB3* respond to the *SBB Root* and the execution continues ((9) in Fig. 3).

In order to establish an association between *convergent processes* and SBB in JSLEE, the synthesized processes are translated into Java components. To do so, the abstract convergent processes are integrated with execution patterns (*conditional*, *fork, join,...)* defined as Java snippets. This process is performed at run time.

As explained before, *convergent process*es generated in Java are monitored in the *Monitoring module*. The monitoring in AUTO is done using JSLEE alarms. Alarms monitor the execution of the services, and save the information into the log. The source code of alarms is inserted in the Java code during the translation between abstract *convergent processes* into SBB Components in the *Adaptation module*.

Regarding the alarms, two issues must be considered: firstly, if the number of alarms added to the SBB is very high, the quality of monitoring will be reasonably good but executing the verification process of each service so frequently may negatively affect the performance. This means that, in order to avoid an excessive overhead during monitoring, the number of alarms must be minimized. Secondly, if the failure is raised in more than one service, it could be necessary to replan and/or re-adapt the entire process. Besides, it could be necessary to undo all the tasks performed until the error occurred. The latter may be very time consuming. To address these issues, AUTO incorporates the region-based reconfiguration that will be explained later.

During reconfiguration process, if any service malfunctions during the execution, the failing service can be replaced by one or several services that obtain the same effect. For example, an *operator* called "communicate" may be divided into functions to search the contact number (Search CN), locate contact (Locate C) and make a call (Call). Similarly, a set of tasks may be replaced by a new individual *operator* as long as the *pre-* and *post-* conditions correspond to the original service (see Fig. 4). For instance, in Fig. 4 two paths, {*s1, s2, s3*} and {*s4, s5, s6, s7*}, may be traversed to achieve the goal, which means that the same high-level operator may be associated with different services. This is possible thanks to the fact that each operator may have several service candidates. For the sake of example, let us assume in Fig. 4 that services *s5* and *s6* in the path 2 correspond to the functionality of the operator *o2* in the first graph.

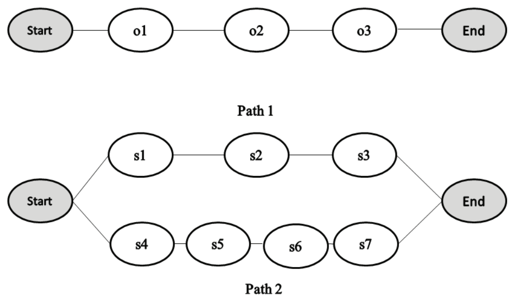

Fig. 4. Levels of services

The central idea is to represent the region to repair using *input*, *output*, *pre-* and *post-* conditions (IOPE). This is the goal to reach using the planning algorithm that creates a new plan. Furthermore, the planning algorithm considers user preferences through the cost function that assigns a cost value to each generated plan [5]. The algorithm continues and, if the repaired region is very large and include too many nodes, the whole plan will be replaced by the existing ranking of plans.

IV. MONITORING AND ERROR DETECTION

A. Event based monitoring

Monitoring provides an effective and reliable method for error detection. Monitoring systems are responsible for gathering context information and presentation of such information to other components in order to provide tools for performing appropriate actions; in this case, for error detection and convergent reconfiguration.

According to management models of events and incidents described in the ITIL[1] framework (Information Technology Infrastructure Library), an error is the root cause for service interruption and reduces quality of service provisioning. Within services operation, various events may be generated which can be categorized in: *Informative* which indicate regular service operations, *Warning* events that indicate an unusual transaction or *Exception* events that may indicate a failure. An *Incident* is the occurrence of a warning or exception event. When an error occurs it can generate one or many incidents associated with service execution such as decreasing in level agreements (SLAs), sending of incorrect data in output parameters, messages exception, or simply no response to certain requests. In order to detect effectively and reliably errors and their root causes, it is necessary to store, categorize and prioritize properly events including information events and incidents.

Previous work [19] focus on mechanisms to intercept messages of generated events using handler-based, wrapper-based and proxy-based mechanisms. Other works such as [20] focus on a formal and more detailed definition of event types, which facilitates error detection and interoperability between different monitoring systems.

In the present architecture, messages associated to events are stored in the *server log*. Thus *monitoring module* acts as a message handler, which includes tasks such as message interception of informative events and incidents, as well as generation of new warning and exception messages which are inferred from the expected service behaviour and of the rules defined in the module. In order to categorize and prioritize informative events and incidents, to generate new incidents, and to diagnose errors, the module has a rule base and an inference engine that detects errors from associated incidents (warning and exception events). An advantage of rule-based systems is that knowledge base can easily be modified to include new rules, for example, to include rules associated with known bugs (errors which root cause has been identified but which has not been solved appropriately) from a set of associated incidents; thus the effectiveness and reliability of the module is improved. When an error is detected, this information is sent to the decision support module to determine whether the change in a region or entire process is necessary.

B. Components of monitoring module

AUTO monitoring module is detailed in Fig. 5, monitoring module interfaces directly with the *server log* and with the *decision support module*. The main components of the module are: Service Register, Service Representation, Knowledge Base, Working Memory, Inference Engine and Fault Detector. Next, the components are explained in detail. The first step to properly detect errors is to define the expected behaviour of services taking into account functional and non-functional aspects. Common methods for defining functional aspects in web services are WSDL and WADL. Non-functional aspects such as legal constraints, costs of use, reliability, and performance may defined by using policy standards such as WS-Policy.

Decision support module defines the services that will be monitored according to the abstract plan and records such services in the monitoring module ((1) in Fig. 5). A list of all monitored services is stored in the Service Register which generates the service representation and associated policies ((2) in Fig. 5).

[1] www.itil-officialsite.com

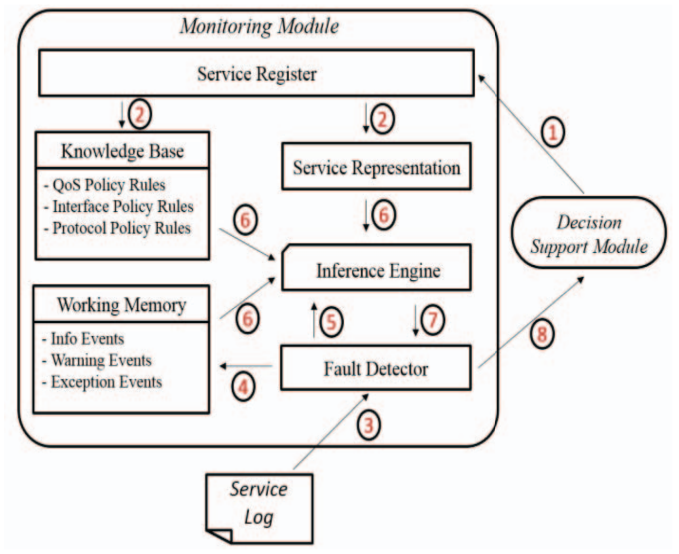

Fig. 5. Monitoring module in AUTO

In order to represent services the present approach uses SOA Ontology [21]. SOA ontology includes a service definition, service contract and service interface (see Fig. 6), as those terms are platform independent so they can be used to represent both Web and Telco services. *Service contract* defines legal aspects and interaction aspects associated with service use. Staring from this model, *Contract Policy Rules* are defined which are stored in the knowledge base. The *service interface* defines how other elements can interact and exchange information with the service, considering the types of information (InformationType) and constraints (Constraints), from this modelling, the *Interface Policy Rules* are defined which are also stored in the knowledge base. Technical aspects associated with the type of service (either Web as SOAP, REST, or as SBBS Telco SIP Servlets, etc.) are stored as *Protocol Policy Rules*.

As mentioned above, each time a service is running, this service records execution information as events in the *server log*. All these events are verified by the *fault detector* and included as facts in the *working memory* ((4) in Fig. 5). When a service is successfully executed, so *Information event* is included, this event includes timestamp, submitted values in inputs, obtained outputs, and execution time. When a service is not successful executed because inputs were not properly submitted (not compliant with defined service interface) so a *Warning event* details wrong inputs. When a service does not respond properly despite having correct inputs (compliant to defined service interface) so an *Exception event* is generated.

From the rules stored in the knowledge base, events stored in the working memory and service representation; the Fault Detector proceeds to call the *inference engine* ((5) in Fig. 5), as a result certain rules are activated which allow the generation of new events either informative, warning or exceptions. When Exception events are detected in working memory this information is sent to *decision support module* ((7) and (8) in Fig. 5) which is responsible for determining if it is necessary to change a service, a region or the entire process, thus ending the monitoring process.

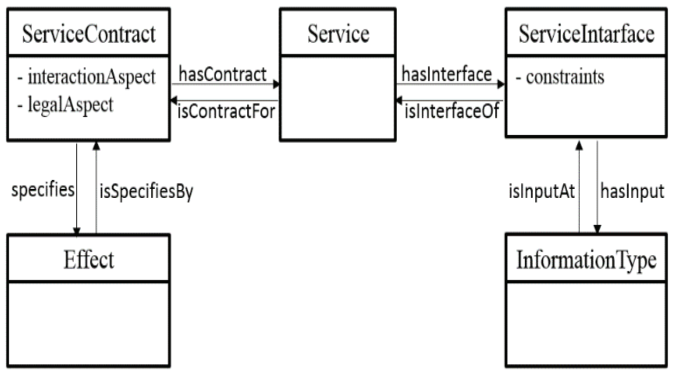

Fig. 6. Service concept in SOA Ontology (Adapted from [21])

C. Services levels for error detection

SOA ontology defines service contract and services interface as a fundamental part of service modelling. From this model, different levels have been defined for definition of rules that allow monitoring.

1) Level 1. Contract policy rules

The first level corresponds to contract policies which include service interaction issues such as reliability or performance, and legal aspects such as cost or responsibilities associated with execution of services. An example of rules associated with this level may be:

- Service reliability is 99 %, i.e. in 100 executions only it may fail once.
- Service performance is between 10ms and 50ms, i.e. the service must respond within this range of values.
- In addition, a general policy can be defined to monitor performance of all services including generation of warning events when service performance is at 95% of the maximum allowed (45ms in the previous example). The latter allows detection of errors before these errors affected contracted issues (legal and technical).

Given the flexibility of rule-based systems and formality of representation model of proposed services, definition and administration of the policies is facilitated, the latter allows easy modification of rules (values of reliability or performance) to suit the requirements of user requests.

2) Level 2. Interface policy rules

At this level, ranges requirements, values types for inputs and outputs as well as restrictions of pre-conditions are formalized through policies. A concrete example may be a basic Health service that using a User ID as input retrieves: weight, body temperature, respiratory rate, heart rate, and blood pressure.

- Restriction on inputs which must be integer between 000000 and 999999 (allowable values for User ID in the system).
- Restriction on outputs for height which must be floating numeric between 30.0 and 250.0 considering that measurement units are centimetres (cm).

- As in the case of Level 1, general policies can be defined to alert when sending or receiving values close to maximum values.

3) Level 3. Protocol policy rules

Since services may fail at any time, so they may return error messages associated with the protocol which are not part of inputs and outputs defined in normal service operation. Consequently, it is necessary to include policies which help to determine the root cause of error from received messages. A concrete example is the response messages of the HTTP protocol.

- If the result of the service includes a message with Error code 404 (Not Found), so the services has not been found, therefore an exception is generated and the support module is notified to perform the appropriate action.
- If service result includes a HTTP message with code 500 (Internal Server Error), so it indicates that the service exists and has been invoked according to interface requirements, but an exception has been risen and it should be notified.

V. Rules Modelling and Prototype

The objective of this section is to present rules modelling that represent the different levels of the model. Drools rule engine is implemented on the Java platform and has its own language for the definition of the rules (DRL) which are stored in the knowledge base; the facts (Log events server) are instances of classes that are inserted into the working memory. The inference engine is based on the RETE algorithm. Next it is shown the implementation of policies for a basic health service that serves patient information from its ID.

A. Registration and modelling of service

Initially when a service is registered in the monitoring module, this service must be mapped to the model defined by the SOA Ontology considering the contract interface and the associated restrictions. A code snippet showing this process is shown in Fig. 7, initially some information is registered: the service interface and contract, the types of information (InformationType), age, weight and height, as well as some restrictions for value ranges and data types that can have outputs and inputs. Finally service representation and event log are inserted as facts in the Rules Engine session to be monitored.

B. Failure detection and server registration

Information event messages generated during service execution are sent to the fault detector and stored in the service log for future use. Basic information events are the one that corresponds to a service calls and include the time when the service call was done and input and output values. When an event is generated, the fault detector is responsible for inserting as a fact in the working memory and then it executes the inference engine in order to detect an error occurrence.

```
ServiceContract sc = new ServiceContract();
ServiceInterface si = new ServiceInterface();
InfoType id =
    new InfoType("id",InfoType.DataType.INT);
InfoType age =
    new InfoType("age",InfoType.DataType.INT);
InfoType weight =
    new InfoType("weight",InfoType.DataType.FLOAT);
InfoType height =
    new InfoType("height",InfoType.DataType.FLOAT);

ConstraintNumberRangeValue<Integer>
    ageConstraint= new ConstraintNumberRangeValue
        <Integer>(age, 0, 120 );
ConstraintNumberRangeValue<Float>
    heightConstraint=new ConstraintNumberRangeValue
        <Float>(height, 30.0f, 250.0f );

si.getInputs().add(id);
si.getOutputs().add(age);
si.getOutputs().add(weight);
si.getOutputs().add(height);
si.getConstraints().add(ageConstraint);
si.getConstraints().add(heightConstraint);

Service bhSingns =
    new WebService("BasicHealthSings", sc, si);
IncidenceRegister incidenceRegister =
    new IncidenceRegister();
ksession.insert( incidenceRegister );
ksession.insert( bhSingns );
```

Fig. 7. Service mapping to SOA Ontology

An example of such messages following JSON nomenclature is described in Fig. 8. In the second service call an error occurs because the result for height is 274.0 which exceeds the range of allowed values defined in the interface (30.0 to 250.0).

```
[{
 "timestamp" : "14-04-2015 11:01:44.598",
 "serviceName" : "BasicHealthSings",
 "inputs":[
    {"parameter":"id","dataValue":"123456"}
        ],
 "outputs":[
    {"parameter":"age","dataValue":"30"},
    {"parameter":"weight","dataValue":"73.5"},
    {"parameter":"height","dataValue":"171.0"}
        ]},
{
 "timestamp" : "14-04-2015 11:01:45.821",
 "serviceName" : "BasicHealthSings",
 "inputs":[
    {"parameter":"id","dataValue":"789123"}
        ],
 "outputs":[
    {"parameter":"age","dataValue":"25"},
    {"parameter":"weight","dataValue":"68.3"},
    {"parameter":"height","dataValue":"274.0"}
        ]
},]
```

Fig. 8. Information Events of services calls

C. Rules definition

Knowledge base stores rules defined for policies at different levels. These rules implement logic for accomplish restrictions at each level. Every rule has a name that identifies it, the "when" section which defines restrictions to be accomplished for the rule to be triggered, and a "then" section defining the actions to take when a rule is triggered.

Fig. 9 shows an example of a rule that implements the policy to throw an exception; this exception is thrown if there is a violation of a restriction for maximum and minimum values in a service output. In the "when" section initially it is verified that there is a service ($sName) registered in working memory, additionally it verifies that there are call events to this service (ServiceCallEvent). Then it is verified that the service interface ($sInterface.outputs) contains numeric outputs, specifically the type INT or FLOAT. From the service call events (ServiceCallEvent) the output parameters values ($pValue) are taken. Finally the output values ($pValue) with the maximum permissible values according to restrictions are checked.

If the rule is accomplished, that is an output value is out of range, the execution will be launched, i.e., the code in "then" section is executed. In the example a new exception event (NumberRangeValueExceptionEvent) is generated, next this event is added to the event log, thus the fault detector notices the error and send the event to decision support module.

```
rule "ConstraintNumberRangeValue_PolicyRule"
dialect  "mvel"
when
   Service(
       $sName:serviceName,
       $sInterface:serviceInterface )

   ServiceCallEvent(
       serviceName==$sName,
       $callEventOutputs : outputs )

   $infoItem: InfoType(
       $itName:name,
       dataType in (DataType.INT, DataType.FLOAT)
       ) from $sInterface.outputs

   $pValue: ParameterValueBinding(
       parameter==$itName,
       $dataValue: dataValue
       ) from $callEventOutputs

   $constraint : ConstraintNumberRangeValue(
       informationType==$infoItem,
       maxValue <= toNumber($dataValue, $infoItem)
       ) from $sInterface.constraints
   $incidenceRegister : IncidenceRegister()

then
   NumberRangeValueExceptionEvent exception =
       new NumberRangeValueExceptionEvent(
              $sName, $constraint, $pValue );

   $incidenceRegister.eventList.add( exception );

end
```

Fig. 9. Rule implementing constraints policies for maximum values

The above rule applies to services that define numerical outputs including restrictions for maximum and minimum values in their service interfaces. The same steps may be followed to define new restrictions, for example when an input or output type is email. In this case the first step is to create the class that defines the constraint (ConstraintEmailValue), then it is defined the rule that implements the policy (ConstraintEmailValue_PolicyRule), and finally it is defined the event which is triggered when the restriction is violated (EmailValueExceptionEvent or EmailValueWarningEvent).

VI. Related Work

Vaculin and Sycara [20] describe a mechanism for monitoring semantic web services based on OWL-S. The authors present a model based on events that facilitates log management and services monitoring. This work presents an ontology comprising different kinds of events such as procedure calls, inputs assignation, outputs processing, precondition evaluation, results assessment, fault events and errors, among other. During execution various events are emitted, these events are included in the ontology and are used to generate a log record. The ontology is platform independent and can be used for diverse monitoring systems. The work focuses on the definition of the ontology, but does not detail the architecture and other components required for a monitoring framework.

A proposal for monitoring cloud services is described in [22]. This work shows how to discover suitable payment services based on SLA service level agreements. To do this, services and SLA are modelled through an extension of WSMO that includes QoS requirements. This modelling ensures interoperability in requirements definition and facilitates monitoring. This approach includes also the definition of dependencies between services and an algorithm that allows correct fault detection when these faults are dependent on other services. Unlike the present approach, this work focuses on services offered in the cloud infrastructure (IaaS) and also in the monitoring services discovery, but does not detail how monitoring services should be built.

Emeakarohaa et al. [23] propose an infrastructure to detect violations in the SLA (DeSVi - Detecting SLA Violation infrastructure). For this, initially low-level metrics are defined such as load and unload bandwidth, which allow adequate monitoring of resources; then it is defined a mapping mechanism (LoM2HiS) to convert these metrics to high-level requirements defined in SLA such as service availability. The proposed framework is tested on two cloud applications, during tests time intervals are discussed for measurement depending on the type of application. Authors focus on monitoring cloud services, but clarify similarities with monitoring Web services and Grid Systems. The main contribution of this work is the mapping mechanism LoM2HiS therefore it does not delve into formal description of SLA. Regarding the mechanism used specifically to detect the occurrence of a violation of SALs the authors mention that the logic is implemented by a Java routine that compares SLA values with defined high-level metrics.

Other works focus on formalizing QoS attributes as part of SLA definition for a service, and how these attributes can be used as a basis for Web services monitoring [24] and [25].

However, they do not detail how to implement monitoring mechanisms to verify compliance with defined QoS attributes. Table 1 summarizes the characteristics of the related work.

TABLE I. RELATED WORK

	FEATURES		
	Domain	***Contribution***	***Difference***
[20]	Semantic web services	Definition of event ontology	It does not define a mechanism to infer the existence of errors from generated events.
[22]	Cloud services monitoring, IaaS	WSMO extension for defining QoS based error detection caused by dependencies between services	It focuses on the discovery of existing monitoring services that meet the requirements, but not in its construction.
[23]	Cloud services monitoring	Defines a mapping mechanism to convert low-level metrics to high-level requirements.	The inference is defined error routine in Java; therefore it has no sufficient flexibility as a rule-based system.
[24] [25]	Web services monitoring	Definition of SLAs as a basis for defining a monitoring system	Implementation of monitoring mechanism is not detailed

VII. EXPERIMENTATION

In order to evaluate the time that the monitoring module spend to execute the verification of a possible error, several tests were performed taking into account variations in the number of service call events that are inserted into the working memory and the number of errors presented. Basically time in milliseconds that the module takes to execute the main tasks was registered:

- Knowledge base creation time
- Service register time
- Inference engine execution time

Results are presented in Fig. 8 (a), (b) and (c) respectively:

It can be seen that the task that spent more time is the creation of knowledge base; it is around 2.6 and 2.8 seconds. It should be kept in mind that this task is only done once and is independent of the number of facts, for this reason the variation of time with respect to the number of events is minimal. Same as the knowledge base creation, the service register time remains constant over the number of events or errors.

Finally, it can be seen in figure 8-c there are an important variance in the time needed to execute the inference engine in relation to the number of events inserted into the working memory. But it makes no difference whether these are informative events or events that generate exceptions and fire a rule. In this respect, it is important to define the period of time in which the inference engine will be called. On one hand, if a long time is used, there will be several events in the working memory and the inference engine execution will be slow; on the other hand, if a short time is used, it executes fast but it can saturate the processor.

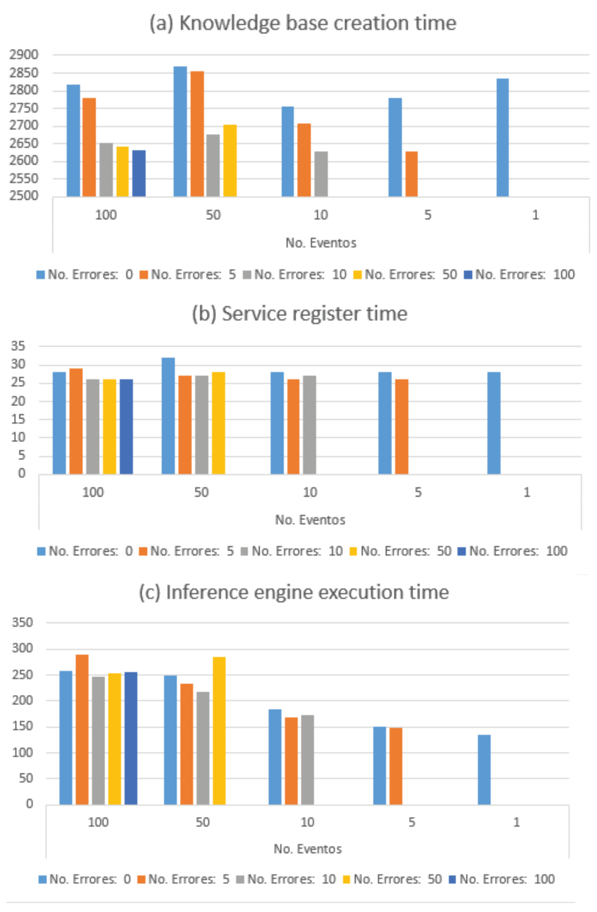

Fig. 10. Execution time vs. Number of events

VIII. CONCLUSIONS

Convergent service composition requires that such services recover from failures efficiently. This work presents the results of the ongoing work towards the definition of a mechanism for planning-based reconfiguration of convergent services across the web and telecom domains. Here an approach for services monitoring in telecom environments using a rule based approach is presented. This approach uses a model based on ITIL. Future work will include the use of different planning architectures for performing the reconfiguration of different failing regions at the same time. Equally further testing to evaluate performance and quality of the approach in cloud-based platforms for convergent services will be carried out.

REFERENCES

[1] Y. Cardinale and M. Rukoz, "Fault tolerant execution of transactional composite web services: An approach," Proc. Fifth Int, Conf. on Mobile Ubiquitous Computing, Systems, Services and Technologies, November 2011, pp. 158–164.

[2] Object Management Group. Profile for Advanced and Integrated Telecommunication Services (TelcoML), Object Management Group Standard. 2012.

[3] C. Venezia, P. Falcarin: "Communication web services composition and integration", In Proc. of IEEE International Conference on Web Services (ICWS-06), 2006.

[4] A. Ordonez, V. Alcazar, J.C. Corrales, P. Falcarin, "An Automated User-Centered Planning Framework for Decision Support in Environmental Early Warnings," Proc. IBERAMIA, pp. 591-600, 2012.

[5] A. Ordonez, V. Alcázar, J.C. Corrales, and P. Falcarin, “Automated context aware composition of advanced telecom services for environmental early warnings”. Expert Systems with Applications, vol. 41, no 13 2014.

[6] A. Ordonez, J.C. Corrales, J. C. and P. Falcarin. , “HAUTO: Automated composition of convergent services based on HTN planning”. INGENIERÍA E INVESTIGACIÓN, Vol. 34, No. 1. 2014, pp.66-71,.

[7] K.-J. Lin, J. Zhang, Y. Zhai, and Xu, B., “The design and implementation of service process reconfiguration with end-to-end QoS constraints in SOA,” Service Oriented Computing and Applications, Vol. 4, No. 3, 2010, pp. 157–168.

[8] M. Shiaa, P. Falcarin; A. Pastor, F. Lecue; E. Silva, and L. Ferreira Pires, "Towards the automation of the service composition process: Case study and prototype implementations", IEEE, ICT Mobile Summit, 2008.

[9] A. Ordonez, V. Alcázar, J. C. Corrales, and P. Falcarin, “Automated context aware composition of Advanced Telecom Services for environmental early warnings,” Expert Systems with Applications, vol. 41, no. 13, pp. 5907–5916, Oct. 2014.

[10] T. Yu, Y. Zhang and K. Lin, “Efficient algorithms for web services selection with end-to-end QoS constraints,” ACM Transactions on the Web (TWEB), Vol. 1, No. 1, 2007, pp. 6.

[11] E. Kaldeli., A. Lazovik, and M. Aiello, M, “Continual planning with sensing for web service composition,” AAAI, April 2011, pp. 1198–1203.

[12] K. Lin, J. Zhang Y. Zhai, “An efficient approach for service process reconfiguration in SOA with end-to-end QoS constraints,” In: Proc. of IEEE int. conf. on e-commerce technology (CEC). 2009.

[13] M. Lescevica, E. Ginters, E. and R. Mazza, “Unified theory of acceptance and use of technology (UTAUT) for market analysis of FP7 CHOReOS products”. Procedia Computer Science, Vol. 26, 2013, pp.51-68.

[14] A. Ordóñez, H. Ordóñez, C. Figueroa, C. Cobos, and J. C. Corrales “Dynamic reconfiguration of composite convergent services supported by multimodal search”. In Business Information Systems 2015, pp. 127–139. Springer International Publishing.

[15] D. Ardagna and B. Pernici, “Adaptive service composition in flexible processes,” IEEE Trans Softw Eng Vol. 33, No. 6, 2007, pp. 369–384.

[16] J. Wang, J. Yu, P. Falcarin, Y. Han; M. Morisio, "An approach to domain-specific reuse in service-oriented environments", In Proc. of Int. Conf. on Software Reuse (ICSR-08), Springer, 2008.

[17] C. Guzman, V. Alcazar, D. Prior, E. Onainda, D. Borrajo, J. Fernandez-Olivares and E. Quintero, “PELEA: a domain-independent architecture for planning, execution and learning,” in Proc. Scheduling and Planning Applications workshop ICAPS conf., Freiburg, 2012, pp. 38–45.

[18] D. Adrada, E. Salazar, J. Rojas and J.C. Corrales, “Automatic code instrumentation for converged service monitoring and fault detection,” Proc. Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference, May 2014, pp. 708-713

[19] K. Bratanis, D. Dranidis, and A. J. H. Simons, “An extensible architecture for run-time monitoring of conversational web services,” in Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond, New York, NY, USA, 2010, pp. 9–16.

[20] R. Vaculin and K. Sycara, “Semantic web services monitoring: An OWL-S based approach,” in Hawaii International Conference on System Sciences, Proceedings of the 41st Annual, 2008, pp. 313–313.

[21] The Open Group., "Technical Standard: Service Oriented Architecture Ontology." Berkshire : The Open Group, 2010. 1-931624-88-7.

[22] A. V. Dastjerdi, S. G. H. Tabatabaei, and R. Buyya, “A dependency-aware ontology-based approach for deploying service level agreement monitoring services in Cloud,” Softw. Pract. Exp., vol. 42, no. 4, pp. 501–518, Apr. 2012.

[23] V. C. Emeakaroha, M. A. S. Netto, R. N. Calheiros, I. Brandic, R. Buyya, and C. A. F. De Rose, “Towards autonomic detection of SLA violations in Cloud infrastructures,” Future Gener. Comput. Syst., vol. 28, no. 7, pp. 1017–1029, Jul. 2012.

[24] B. Koller and L. Schubert, “Towards autonomous SLA management using a proxy-like approach,” Multiagent Grid Syst, vol. 3, no. 3, pp. 313–325, Aug. 2007.

[25] G. Dobson and A. Sanchez-Macian, “Towards unified QoS/SLA ontologies,” in Services Computing Workshops, 2006. SCW’06. IEEE, 2006, pp. 169–174.