2013 IEEE Seventh International Symposium on Service-Oriented System Engineering

Monitoring Web Services for Conformance

Bixin Li, Shunhui Ji, Li Liao, Dong Qiu, Mingjie Sun
School of Computer Science and Engineering, Southeast University, Nanjing 211189,Jiangsu, China,
Institute of Software Engineering, Southeast University, Nanjing 211189, Jiangsu, China
webpage: http://cse.seu.edu.cn/people/bx.li/index.htm

***Abstract*—In this article, we paid more attention on researching how to monitor Web services for checking conformance, where an AOP-based run-time monitoring framework was proposed and explored[1]. In the framework, WS-Policy was firstly used to express the monitoring requirement, then monitoring requirement was described as AOP monitoring logic; ActiveBPEL engine was extended to weave the monitoring logic and the service core execution logic dynamically; a checking algorithm was introduced to analyze whether the user's monitoring requirement was satisfied according to MREG (Monitoring Requirement Expression Graph) and EMSC (Extended Message Sequence Charts); and finally, some control and modification measures were adopted in order to rise the quality of the service composition.**

***Index Terms*—Service monitoring, MREG, AOP, WS-Policy, ActiveBPEL**

I. Introduction

Web Service (WS) is a software system where distributed applications communicate with each other on the web. Existing basic service specification framework only shows the standard of service description, release and invocation [1]. As the de-facto standard to describe web service composition, BPEL (Business Process Execution Language) [2] presents how some basic services to be assembled together to form a business process, which affords more complex function. Composite service essentially has dynamic attributes such as re-composition, re-configure, and dynamic binding etc., which show that even if the process is verified to be correct by the conventional testing and validation techniques before running, it is possible that the implemented and executed behaviors are not conformance to the original requirement. Therefore, it's important and necessary to implement run-time monitoring of WS to realize the actual situations. A monitoring framework adds probes with a special purpose to detect anomalous conditions, capture some important run-time information of the process, and check them against the property specification of the service composition. Run-time error information should be reported as soon as possible to the service developers or providers so that suitable countermeasures can be taken in time to enhance the quality of service (or QoS).

In this article, a kind of AOP-based end-to-end monitoring framework was proposed to tackle the conformance monitoring and checking of dynamic Web service composition. Where WS-Policy model was used to express the user's monitoring requirements such as temporal logic, timeliness, security, reliability etc., such expression manner is convenient and explicit; then both EMSC and MREG were proposed to describe the properties of the service: EMSC offers a graphical representation to describe the desired execution behavior of the process, while MREG shows some monitoring concerned properties of the service; in addition, AOP technology [3] was borrowed to extend the BPEL execution engine so that the extended engine can send the monitoring logic to correlative service entity, which makes the monitoring aspect be imported into the corresponding BPEL execution logic automatically and effectively without changing original system. A special checking algorithm is also given to analyze the service execution behavior.

The rest parts of this paper are organized as follows: Section II introduces some basic concepts and terminologies for later reference; Section III includes a motivating sample service which is discussed throughout in whole article; Section IV discusses how to use WS-Policy to describe the user's monitoring requirements on the service; Section V discusses MREG and checking algorithm; Section VI discusses our end-end monitoring prototype framework in detail, where extended MSC [7] and MREG are used to represent desired properties of Web service composition, checking algorithm is designed to check Web service composition for conformance; Section VII discusses how to extend ActiveBPEL; VIII discussed case study and empirical analysis status; Section IX discusses the related work in the monitoring area; Section X draws some conclusions and discusses future work.

II. Basic Terminology

For easy to understand next monitoring framework, it is interesting to explain some basic concepts and terminologies in this section.

- *WS-Policy expression* expresses user's monitoring requirements in a form of WS-Policy model and document(refer to section IV.1).
- *AOP monitoring logic* includes some AOP aspect models which concerns concrete meanings of monitoring requirement, where a summary of AOP implementation logic is used to capture related running information of Web service (refer to section IV.2).
- *Service description execution logic* is a kind of description language for Web service composition, including WSDL, BPEL and OWL-S etc. A service execution

[1]This work is supported partially by the National Natural Science Foundation of China under Grant No. 60973149, and partially by Doctoral Fund of Ministry of Education of China under Grant No. 20100092110022, partially by the Innovation Fund of Southeast University under Grant No. Seucx201119, and partially by the College Industrialization Project of Jiangsu Province under Grant No.JHB2011-3.

978-0-7695-4944-6/12 $26.00 © 2012 IEEE
DOI 10.1109/SOSE.2013.16

IEEE computer society

engine calls some related resource to execute service process according to the information from service description execution logic.

- *Monitoring information* denotes the service running information, being generated during service process execution, captured by some special monitoring logic implementation codes that are relevant to monitoring requirement aspect.
- *Monitoring and checking result* shows some results in detail about whether the monitored information are conformance to property specification and user's property requirements.
- *Modification information* includes some feedback information from checking process, which are used as guidelines for the correction and optimization of service process.

III. Motivating Sample Service

A simple but effective example of the monitoring issue is described for a travel reservation service, which supports the function of *vehicle booking*, *room booking*, *user account validation* and *payment online*. The whole composite service is a business process expressed as a BPEL document. After the user puts in the reservation information, the service process will check user's account. If checking passed successfully, then either Airline or Train service will run in parallel with Hotel service. In this paper, we only pay attention to the reservation implementation phase of the service process. Corresponding BPEL structure is depicted in Figure 1. The business process (`TravelReservation`) contains the basic service *Hotel*, *Airline* and *Train*.

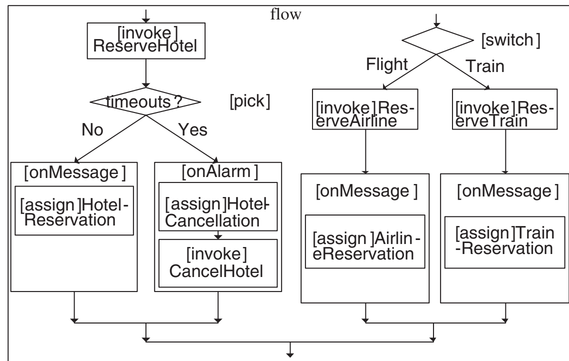

Fig. 1. BPEL document structure of reservation

Since the invalidation of the reservation service will result in delay of the travel, user needs to be informed by the monitoring system in time. Then he can seek for another booking way or modify the travel plan in advance. At the same time, monitoring system should collect the useful monitoring information and feed them back to the service supervisor. That will help improving the quality of the original service process. Moreover, in the composite service available basic services are composed to meet the predefined goal, both functional and non-functional requirements impose that run-time environment be capable of dealing with them. In fact, all these requirements are just the monitoring concerned aspects, we called them *Monitoring Requirements*. For example, *temporal logic* cares the messaging behavior of WS, the executing sequence of the activities and the services; *timeliness* requires the operations and the activities contained in the service must be fulfilled within a constraint time; *security* emphasizes on the message encryption and authentication ways, while *reliability* takes care of the successful reception of multiple messages sent from one service to the other service and the service response efficiency.

IV. User's Monitoring Requirement Expression

In this section, we will discuss how to express user's monitoring requirements using WS-Policy model and document.

A. User's Monitoring Requirement

In this article, the *user* includes two kind of users, one denotes the *end-user* who use Web service to realize their wanted functions or services; another denotes the *service integrator* who use existing basic services or composite services to create new composite service for providing more powerful functions. There are different monitoring requirements from the two kinds of users: for *end-user*, he hope to know the status of current running services such as the correctness of function and efficiency, so that he can provide some science empirical data for other users to select wanted Web services later; for *service integrator*, he hope to test the running status of integrated services (i.e., composite service) by monitoring so that he can find problems, fix problems, complete service composition, and improve QoS etc.

In general, user's monitoring requirements can be classified into functional property aspect and non-functional property aspect. Table 1 summarizes some familiar user's monitoring requirement aspects, where three properties in above sections belong to function monitoring requirements, the other four properties belong to non-function or quality monitoring requirements.

Towards the `TravelReservation` sample service, we mainly consider three monitoring requirements presented below.

- **Temporal logic**: It requires all the message sequences in the reservation implementation scene transfers following the BPEL document specification. This requirement is always essential, and it is correlated to the functional property of the service.
- **Timeouts**: It emphasizes that the hotel must be successfully reserved within time t_1 or reservation is canceled when timer value beyond t_1, the whole reservation should be accomplished within time t_2. Here both time t_1 and t_2 are set by the user, we assume t_1 equals 20 seconds and t_2 equals 60 seconds.
- **Reliability**: We need to know the success invocation probability of the Train service in the past 5 minutes and the average response time t3.

TABLE I
MONITORING REQUIREMENT, MEASURE CRITERIA AND CHECKING MEANS

Monitoring requirement	Measure criteria	Checking means
runtime external errors	returned result from Web service	compare returned results with expected results
temporal logic	sequence of operations and messages	analyzing execution trace according to operation contract
timeouts	execution time of operation under onAlarm	timing related operations and observe timeout process mechanism
security	safety standards for service interaction	capture and analyze the communication pattern of SOAP message
reliability	the times of transaction failure per month or per year	create failure database for service process
transaction	final consistency of service-related common data	analyze business-related service execution result using counterexample
performance	the number of requests processed and corresponding time in a given time	successful instance counting and single instance execution step timing

These requirements will be implemented in our monitoring system. The time label (t_1, t_2, t_3) will be vividly shown in EMSC and the user's special condition requirement will be expressed in MREG.

B. WS-Policy Expression

Web Service Policy Framework (called WS-Policy in general) provides a general purpose model and corresponding syntax to describe the policies of a WS. It also gives a flexible and extensible grammar for expressing the capabilities, requirements, and general characteristics of entities in an XML Web services-based system [4]. Using the WS-Policy mechanism, we can associate the policies (as the monitoring aspects) with the subjects (service entity parts, such as message, activity, one component service or the whole composite service) to which they apply. According to the standard policy expression grammar, we can also define our own service policy (i.e., Log, Timeout). This mechanism is convenient to describe the user's various monitoring requirements on the service. We have defined the WS-Policy meta-model (in Figure 2) for a unified monitoring requirement expression. Attribute "*Name*" is the name of the current policy, attribute "*URI*" represents the *URI* of the policy and attribute "ack" is the policy selection identifier. When the value of "*ack*" is "*true*", that means this policy is selected. Furthermore, "*targetType*" denotes the type of the entity (message, activity, basic service or the process) that applies the policy. "*targetName*" shows the name of that entity. The attribute "*URI*" is very important. It specifies the applied policy for the current monitoring requirement. One monitoring requirement can be mapped into several WS-Policy models, similarly one policy model can also served for several requirements. Policy tells the monitoring system to capture what kinds of running information of the service. According to the syntax of WS-Policy, we can also generate WS-Policy document from policy model. As Policy document is XML-based, it can be transmitted on the web easily. Furthermore, policy document also gives the actual binding address information of the service entity, which will help distributing the monitoring logic to the entity.

For the three monitoring requirements on the `TravelReservation` sample service, we need the message execution sequence and the execution time information of the business process. So, we use the policy Log and Timeout. These two policies are self-defined. Policy Log tells the monitoring system to create a monitoring

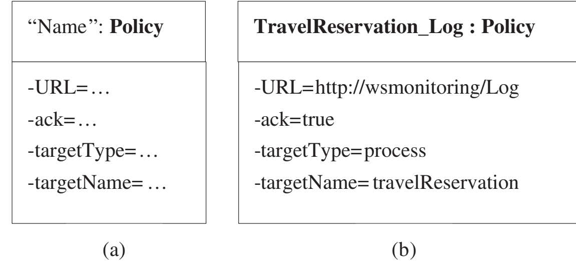

Fig. 2. [a]:WS-Policy meta-model; [b]:Log WS-Policy model.

logic which can intercept message sequence, while policy Timeout tells the system to create a monitoring logic which can get the execution time information. Log WS-Policy model is shown in Fig 2(b). We can see that the name of this policy is "`TravelReservation`_*Log*". *URI* is "*http://wsmonitoring/Log*", which is set in advance. We assume that Log is an acknowledged policy (like as WS-Security, WS-Trust). We can also find that the entity is a process named "`TravelReservation`". Log policy document is also shown as follows:

```
<wsp:Policy Name=http://wsmonitor/Log? xmlns:wsl="..."
    xmlns:wsp=
    http://schemas.xmlsoap.org/ws/2004/09/policy>
   <wsl: Log/>
</wsp:Policy><wsp:PolicyAttachment xmlns:wsa=
      http://schemas.xmlsoap.org/ws/2004/08/addressing>
  <wsp:AppliesTo>
   <wsa:EndpointReference
      xmlns:travel=http://www.txy.com/travel>
     <wsa:Address> http://www.txy.com/travel
      </wsa:Address>
     <wsa:PortType>travel:\verb|TravelReservation|PortType
      </wsa:PortType>
     <wsa:ServiceName>\verb|TravelReservation|Service
      </wsa:ServiceName>
    </wsa:EndpointReference>
   </wsp:AppliesTo>
     <wsp:PolicyReferenceURI="http://wsmonitor/Log"/>
</wsp:PolicyAttachment>
```

Where the label "*<wsp: PolicyReference URI = "http://wsmonitor/Log" />*" specifies the Log policy; while the label "*<wsp:AppliesTo>...</wsp:AppliesTo>*" indicates the detailed address of the service entity that applies the policy. Here, Log policy is applied to the `TravelReservation` service process located on "*http://www.txy.com/travel*".

For Timeout policy, we can also present its WS-Policy model and document in the same way. In fact, as long as the correlative policy for given monitoring requirement had been defined, we can use this WS-Policy mechanism to express the user's monitoring requirement expediently.

V. MREG AND CHECKING ALGORITHM

In this section, we will discuss how *agency end* check returned monitor information and make the conclusion according to service property specification and user's definite monitoring requirement. In our monitoring frame, two kinds of information are provided for the checker located at *agency end* to start the checking process, one is *monitored information*, another is *monitoring requirement expression graph* (we call it MREG), where *monitored information* is service running information captured in *service end*, while MREG is a formal service running specification generated based on the service process description document (e.g BPEL) and user's special monitoring requirement aspect information. Using MREG we can design a stable checking algorithm which is independent on special monitored service property.

A. MREG

Definition 1: : $MEvent$ is a set of message events in a service process. Message event is described in the form of $action(variable)$, where $action$ denotes the basic activities described in related BPEL document such as $invoke$,$receive$, $reply$ and $onMessage$ which marks the $operation$ attribute, while $variable$ is corresponding parameter. In addition, $empty$ message event is also in $MEvent$, $empty$ message event has no actual meanings, it is only used to aid marking and create a link when constructing event transition.Message event transition denotes the sequence execution of related activities and further denotes the interaction process between service process and partner services.

Definition 2: : $MREG(SName, MEvent, R, F)$ is a quaternion which contains following four elements:

- $SName$ is the name of monitored service process.
- $MEvent$ is message events in monitored service process (refer to definition 1). *Empty* message is permitted in actual construction of $MREG$ to keep $MREG$ integrality.
- R is a set of monitoring requirement variables which is consistent with user's different monitoring requirements. R is partitioned into many subsets as $R = \{\epsilon, C, T, S, \cdots\}$, where ϵ is a set of empty variables for aiding to construct a complete $MREG$; C is a set of conditional variables representing conditional variable request of message event transition; T is a set of time variable expressions denoting the time condition for a transition occurs; s is a set of security variables denoting the security conditions should be satisfied by a message event; additionally, R can be extended to express more kinds of monitoring requirements.
- F is a transition function of message event, which reflects the transition status of message event for all kinds of variables in R. For example, function $F(m_1, c_2, t_1) = m_2$ denotes that m_2 will happen and tightly follow m_1 when requirement variables both c_1 and t_1 are satisfied synchronously,where $m_1, m_2 \in MEvent$, $c_1 \in C$, $t_1 \in T$. In practice, the types and numbers of variable conditions that each message transition depends may be different for different service process description documents and user's different monitoring requirements.

B. EMSC

MSC (Message Sequence Chart) provides a trace language for the specification and description of the communication behavior of system components and their environment by means of interchange [7]. In MSC the communication behavior is presented in a very intuitive and transparent graphical manner the same as in UML sequence diagrams, but it offers more useful notations (i.e., time label, local event) that can be used to express monitoring properties of the service expediently. We properly extend one subset of MSC to model the communication process of the WS described by BPEL, which we call it EMSC, a kind of extended MSC. The corresponding relationships between the BPEL document elements and the EMSC members are shown in Table II.

In EMSC, process and its partner service are expressed as process instance and service instance, the specification of behavior properties, such as message transition sequence and message response time in a service interaction scenario, are expressed by describing the receiving and sending of message among services, the sequence relation of local event (for example: variable assignment, timing of a message etc) and message event. We usually call scenario of a single service interaction BMSC, i.e., basic MSC. Message event mainly denotes an operation behavior of basic activities between service process and component service interaction in BPEL document, where basic activity is defined as a set of several relevant message events, while the value change of assign activity variable caused by these operations can be expressed using simple event of a store variable in MSC. For other familiar structure activities, such as sequence, choose, and concurrency, they are defined as sequence execution, branch execution and synchronization concurrent execution, respectively. Structural activity $while$ can not occur in the scenario of single service interaction, this case describing loop execution of a scenario can be described using HMSC (High-level Message Sequence Chart). In service composition process, many local interaction scenarios can form a complete service interaction scenario according to conditional variables in structural activities and logic dependence relations between scenarios. HMSC describes the transition between scenarios and the state change of whole service process and express service running logic in a whole. HMSC is a directed graph which uses BMSC as node and use edge as sequence relation between nodes, where a BMSC may be reused by HMSC to denote the sequence, while and choose relations between scenarios.

EMSC introduces some service conceptions into MSC. Among these conceptions, *message event* in the form of "`action (variables)`" is the most important conception. EMSC can describe the message sending and receiving, the local events (setting value for the variable, clocking the message, etc.) and the BPEL message logic sequence. Besides, the responding time and some other behavior properties of the

TABLE II
CORRELATION BETWEEN BPEL AND EMSC

BPEL	EMSC
process;partner	process instance;service instance
variable	the parameter part of the message event
correlationSet	the dependence identifier of the messages
receive	the message event sent from the service instance to the process instance
assign	simple event the store the variable value
reply	the message event process instance return for service instance
invoke	the message event sent from process instance to service instance and possible return message event
activity	the simple aggregation of serval message events
sequence	serval activities executed in sequence
condition	the local condition on the instance axis
if	the branch activity executed depending on given condition
flow	serval activities executed in parallel
onAlarm	the activity activated by overtime operation
onMessage	the message event passed to process instance and the activity arose

WS can also be represented. EMSC can also describe BPEL structured activities with some special notations (such as par, alt, when). For Travel Reservation sample service, the desired execution behavior property is depicted by EMSC in Fig 3. We can see that EMSC can be smoothly used to model the interaction between BPEL process and its partners. The time label t_1, t_2, t_3 put forward in Section II are marked distinctly. For example, t_3 is the time of accomplishing the train reservation, that means t_3 is equal to the sum time persisting from the beginning *message event* reserveTrain(ReserveTrainIn) to the end *message event* trainReserved(TrainReservedIn).

C. MREG Generation

In above subsection, we discuss how to describe the behavior property of interaction in service process simply and intuitionally using some basic concept and principles from MSC and EMSC. However, during monitoring Web service, users usually expect the running of service process instance not only conformance to service function property specification expressed in form of BPEL or EMSC, but also satisfy their more non-function features. Therefore, it is necessary to combine EMSC and user's special monitoring requirement to generate MREG so as to check captured monitoring information easily.

The main steps of generating MREG are as follows:

- Transform each BMSC which denotes Web service process and component service interaction scenarios into a series of partial MREGs according to the sequence of message event occurring, where requirement variable is empty temporarily.
- Link all partial MREGs based on logic relations of HMSC scenarios, where there a empty message event is added between end message event of former scenario and start message event of latter scenario, and a transition with empty monitoring requirement variable ϵ is added between empty message event and former or latter message event.
- Add start message event labeled with 0 and end message event labeled with E for above generated MREG in step 2.
- Add conditional variables to activate message event transition and user's special monitoring requirement aspect information on service process to MREG.

In final MREG graph, circle with a number denotes message event, special cases are using 0 denote process start and E denote final state; the ϵ over the arrows between circles denotes empty variable, which means a direct transition can happen between the message events at two ends of arrow; "C:..." denotes conditional variable which means the variable condition of the transition; "T:..." is the time variable expression, other service message event transitions all can be marked with special symbols.

The MREG of `TravelReservation` service is shown in Figure 5. The circle represents message event, some marked beside with event names means the circle is an actual message event, while some are only annotated with numbers means the circle is a supplemental event. For the solid line, the annotation "$C : flight$" on it means it is a variable condition line, while "$T : t < t_1$" means it is a time condition line. If there is only "ϵ" on the line that means the transformation can carry through directly under any condition. Here, time and condition constraints are the user's special monitoring concern. They will be used for checking monitoring information.

D. Checking Monitored Information

When both the monitored information and MREG have been at hand, we can do the checking and determine whether user's monitoring requirements are satisfied or not. During checking, it is necessary to maintain checking algorithm stable running and use a unified and consistent way to check various user's monitoring requirements. A kind of pseudocode checking algorithm based on MREG is preposed as follows:

We can see from the algorithm that the main task is how to collect monitoring information and MREG on the initial stage, line 3 mark the outermost structure of the service process. In the main body of the algorithm, line 0 extracts the sequence of message events happened in actual running services from monitored information, and records important service behavior information related to MREG's monitoring requirement variable R. The algorithm branches in line 2,

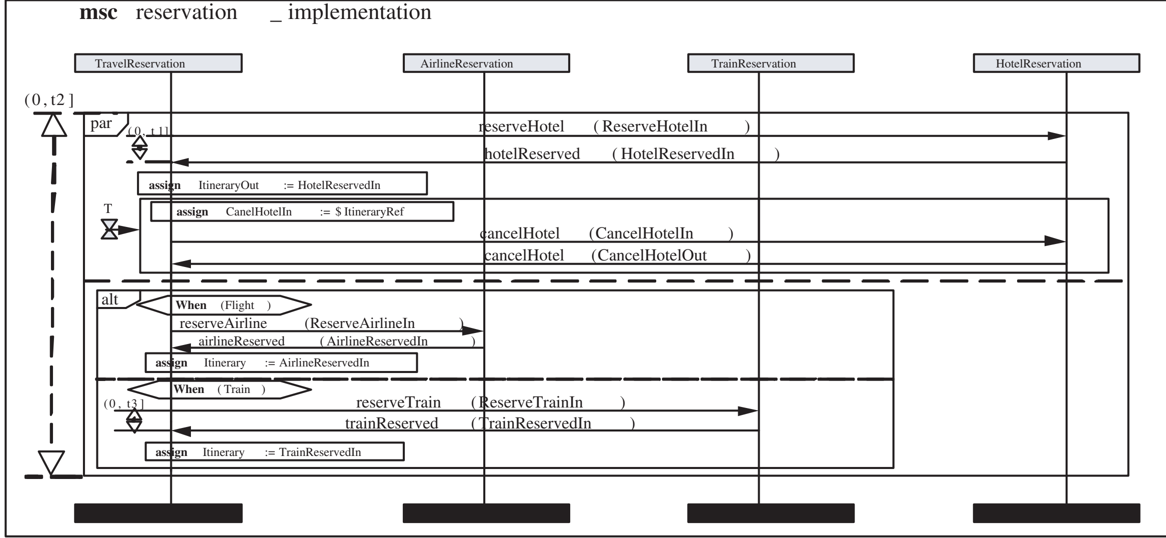

Fig. 3. The reservation implementation scene in EMSC

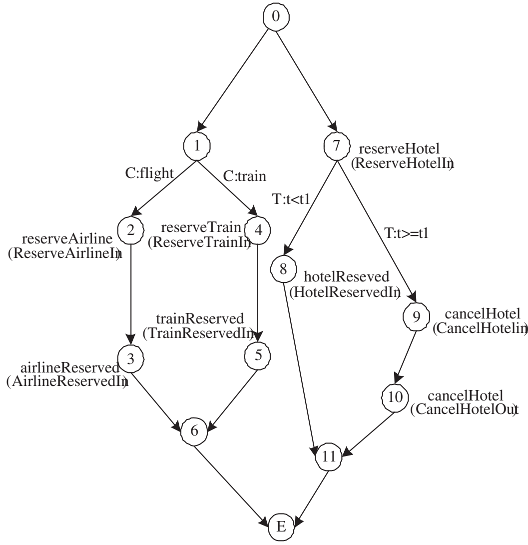

Fig. 4. MREG of reservation implementation

Algorithm 1 Checking monitoring information based on MREG

Initialization Phase
1. Agency gets corresponding files from SIL
2. Agency disposes the useful instances information to get the eventual running results sent to Analyzer
define: m=the number of result items (or the monitoring times)
3. Analyzer gets MREG. define: Static n=activity parallel numbers (from EMSC); time and variable conditions (from EMSC)
Main Phase
if $m! = 0$ **then**
  Analyzer takes out the mth result item, marks its message event sequence as $Sm = (E_{m1}, E_{m2}, , E_{mk})$ and records its other condition information (e.g. interval time)
  define: $p = n$
else if $p > 0$ **then**
  tracking along MREG path from the start state 0 based on S_m;
end if
if If cannot reach the end state E **then**
  print "the mth result item is wrong"; break;
else if all message events in Sm have walked through (no remainder message event in Sm) **then**
  print "the mth result item is wrong"; break;
else if back to state 0 **then**
  $p = p - 1$
end if
if $p <= 0$ **then**
  print "the mth result item is wrong";$m = m - 1$
end if

line 3, line 8 and line 9 corresponds to the checking of sequence, parallel, choose and loop structure in service process respectively and presents analysis steps for the four main structure. In fact, Web service process may be composed with nested basic services or composite services based on above main four kinds of structure forms, the checking algorithms in different structures should be used synthetically and layer upon layer. Because checking algorithm are based on MREG, it not only checks whether service execution behavior conformance to BPEL specification, but also checks whether all kinds of actual message event occurs, activity completion, basic service invoke, and the running of whole service process to satisfy user's all kinds of monitoring requirement aspect conditions. With the addition of user's monitoring requirement aspect, MREG can extend requirement variable set expression without changing the main structure, checking algorithm's checking capability is expanded indirectly and adaptively and furthermore service running behaviors and more properties can be checked.

This is a stable checking algorithm which is independent from some special monitored service property. The complexity of algorithm is related to variable m and the length of message event sequence. Checking algorithm is usually performed to

check once or time-after-time given service process instance running, the structure of process is well-designed and keeps unchanged almost, so the value of m is fixed and not too large. Similarly, the length of message event sequence of a single service execution is also controlled. Therefore, for a given service process document and given running result, the execution time complexity of above checking algorithm is acceptable.

In conclusion, EMSC provides a good way to describe the interaction property specification of the BPEL service process. MREG is generated based on EMSC, furthermore it is endowed with more monitoring conditions and it is convenient for checking monitoring information. The checking algorithm will be shown in section IV.3.

VI. Monitoring Prototype Framework

We give the monitoring framework for WS composition in section VI.1, while in section VI.2 we simulate the monitoring of the sample service. Lastly, the monitoring information derived from the three monitoring requirements of the sample service is checked against MREG in section VI.3.

A. Monitoring Framework

The functional logic of a service is regarded as core concern, while non-functional logic is regarded as cross-cutting concern. Cross-cutting concern is usually closely correlated with the monitoring requirements. It commonly spans multi-modules of the service. AOP is an aspect-oriented programming technology, which can use the aspect to encapsulate cross-cutting concern logic. Aspect can be added into the existing application smoothly and effectively. In article [13] AOP is used to extend the BPEL language, forming AOP4BPEL which has good modularity and flexibility. But this mechanism will change the original BPEL document and a lot of service facilities need to be improved. In this paper, we regard AOP aspect as Monitoring Logic and use AOP to extend the open-source BPEL engine ActiveBPEL [6]. The monitoring prototype framework is shown in Figure 5. Solid line with arrow denotes the information path, which shows the interaction among the components. Here, components indicate extended ActiveBPEL engine, partner services (e.g. S_1, S_2, S_3), and other system components (e.g. Agency, Analyzer).

As in section IV.2, we expressed monitoring requirement as WS-Policy. Policy document contains the address information of the monitored service entity. We extend ActiveBPEL engine with an important component Weaver. Weaver weaves the monitoring logic with BPEL execution logic in three steps: Firstly, it receives the WS-Policy document and finds the matched BPEL; then, it distributes the policy to the related partner service; finally, partner service generates the needed monitoring logic and imports monitoring logic to the service core logic on run-time. The partner service side is installed with Monitoring Logic Adapter which can generate AOP aspect automatically according to special AOP language syntax and the policy identifier information contained in WS-Policy document. After adding monitoring logic into correlative service core logic, it will produce the monitoring information during the execution of the process. In Figure 5, we can see that order is sent from the engine to Agency. In fact, order indicates which kind of information (e.g. message sequence) and which samples (during a period, e.g. the last 5 minutes) of the service are concerned in current monitoring. Then, Agency will use the order to request samples from SIL (Sample Information Library). SIL stores many monitoring results received from basic services. The results are stored in the form of file. Moreover, SIL can create a table which records the date and the time each file is generated. After searching from the table, SIL will send the useful monitoring information to respond the Agency. Then Agency will send those monitoring result samples to Analyzer afterward. Analyzer checks the result against the MREG property specification of BPEL service. Finally we gain the current once monitoring conclusion information of the composite service. In this paper, we use AspectJ [5] as the AOP language. In AspectJ, aspect has three important conceptions: join-point, point-cut and advice. A series of join-points belonged to one aspect are defined during the service executing, and point-cut can match the aspect with its interesting join-points. Advice defines the action actualized at the corresponding join-points. Actually, each aspect is correlated with a point-cut, and it defines some operations needed to be executed before, after or around the matched join-points. Through this approach, AOP separates the monitoring logic from the BPEL core execution logic smoothly. It will intercept the monitoring information during the execution of the BPEL process.

B. Monitoring and Checking Steps

In end-to-end monitoring framework, each end includes the components to support monitoring, each component implement important special task. The arrows with labels in the framework denotes the information flow and interaction process during monitoring, where it is user's monitoring requirement to drive the running of whole Web service monitoring framework, which lead to following monitoring and checking steps:

- User brings forward the monitoring requirements expression in form of WS-Policy mode and document.
- Generating MREG based on service description document and constrain conditional information of the real monitoring requirement aspects concerned by users.
- Monitoring logic generator transforms the monitoring requirements expression to AOP monitoring logic, i.e., AOP model.
- ActiveBPEL Engine receives monitoring logic and find, match BPEL service process instance to be monitored, and then partitions AOP monitoring logic into AOP_1,AOP_2,...,AOP_n according to the value scope of attribute $target$ in monitoring logic, supposing that there are n basic services S_1, S_2,...,S_n as monitoring objects of composite service process, and distributes monitoring logic to each partner service according to the detail

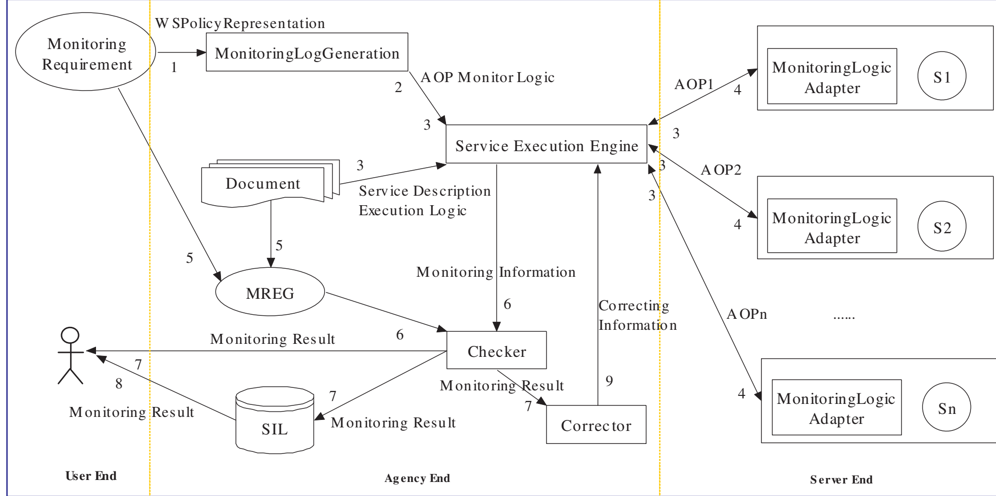

Fig. 5. End to End Service Monitoring Prototype Framework

address binding information in each service description document. At the same time, Weaver complete the task of weaving AOP monitoring logic and service execution logic.

- Monitoring logic adapter confirms the legality and validity of current monitoring aspect required to be monitored and monitoring requesters, and transforms each confirmed AOP_1,AOP_2,..., and AOP_n to corresponding monitoring logic code (i.e., AspectJ code, AspectC code or AspectC++ code etc.) which matches each basic service. Monitoring logic code can be executed in parallel with service code function code when service is executed, where AOP logic is inserted indirectly into service execution code to monitor the running process of code and capture monitoring information neatly and effectively, and feedbacks monitored information to ActiveBPEL Engine in a form of special file used in AOP model.
- Checker ChView uses designed special checking algorithm to check monitored information and make checking conclusion based on monitoring information from ActiveBPEL Engine and MREG.
- Monitoring and checking results are feedback to user and stored in SIL(i.e., sample information library), so that user can inqure monitoring and checking results later[2]. In addition, ChView sends monitored result to Modifier when result is wrong.
- User can inquire historical monitored results of corresponding service process in SIL to aid to choose service provider.
- Modifier analyzes monitored result and feedbacks some guideline information about modification and optimization of service process to service agent end of ActiveBPEL Engine. Agent end decides whether it is necessary to deliver monitored result and modification suggestion to service provider. Finally, relevant control modifications and assessment actions are adored selectivity and QoS of composite service is improved.

[2]monitoring and checking result information is stored in SIL in a form of database table, where date, time, corresponding store file including monitoring information, and monitored result are recorded

VII. ACTIVEBPEL ENGINE EXTENSION

We have discussed the main structure and running process of monitoring prototype framework, where core component is the extension to ActiveBPEL Engine on agent end. In this section, we will discuss how to extend an open source BPEL execution engine ActicveBPEL engine. Extended ActicveBPEL engine can transform AOP monitoring logic for user's special monitoring requirement to corresponding service entity so as to support collecting monitoring information.

A. ActiveBPEL Engine

ActiveBPEL engine accepts the definition of BPEL processes, create process instances and execute them. ActiveBPEL engine includes three main parts: *engine*, *process* and *activity*. *Engine* executes one or more matched BPEL processes based on the sequence of activities to be included or links. Each *activity* may call some external partner component services, thus many component services are associated according to operation sequence of activities to form Web service workflow process composed by the interaction and collaboration of a lot of component services. ActiveBPEL engine create process instance according to BPEL process definition (i.e., XML file) and execute this instance. ActiveBPEL engine is created by an engine factory who still take charge of management of supporting services. Engine configuration manage all supporting services of ActiveBPEL engine by using default provided by an object and configuration file $aeEngineConfig.xml$. The main functions of ActiveBPEL engine includes process creation, data process with input and output, evaluating expression, recording process log etc., main components of a

process includes Partner links, Partners, Variables, Correlation sets, Fault handlers, Compensation handlers, Event handlers, top-level activity and basic activity etc.

When an input message (a message relevant to invoke activity) or a PICK activity alarm arrives, i.e., an initial activity of BPEL process is triggered, ActiveBPEL engine will distribute input message to existing correct process instance according to *correlation sets* or create a new matched process instance, and further execute whole process based on internal links between activities of the process. BPEL process is composed of three type of activities: *basic activity*, *structural activity*, and *special activity*. *basic activity* deal with some simple behaviors, for example, receive message, response message, invoke service, and variable assignment etc.; *structural activity* has some construction type such as branches, loop, and sequence etc, which are in general composed of many basic activities; *special activity* can declare scope, deal with the stop of a process and compensation etc. Each activity has a series of execution states, it enters and leave these states now and then based on happened internal events during execution.

B. ActiveBPEL Engine Extension

In our monitoring framework, Weaver is regarded as an object of application process service, corresponding process class is configured in configuration file $aeEngineConfig.xml$. The class mainly extracts the value of $target$ attribute in AOP monitoring logic model, analyzes the hierarchy of $target$ (i.e., belongs the process itself or belongs to one or more single basic service in the process), and then add monitoring logic information and special identifier (e.g., identification information on service agent end) to the first communication message between basic services and process. Along this way, monitoring logic is distributed to each basic service, monitoring logic is dealt with by monitoring support system (e.g., monitoring logic adapter) at each basic service, and monitoring logic is weaved dynamically into service core execution code in final service execution.

C. Monitoring Logic Adapter

Web service is loose coupling and dynamics: loose coupling means that each basic service involved in a composite service process may be located in different Internet site, implemented in different programming languages and supported by different software platform; dynamics means that it is a dynamic selection process for a BPEL service process to determine what basic services to be called, while each basic service could has some unexpected changes such as upgrade of version etc. It is worse that service provider can not notify users these changes timely. Therefore, agent end transform monitoring logic to service end to perform monitoring. Each basic service participating BPEL service process in the framework has its monitoring logic adapter. These adapters will transform monitoring logic into local monitoring logic code matching with basic service automatically based on attribute values in AOP model, and capture service running information safely and controllably when local service is called and executed.

Monitoring logic adapter is located at the end of service provider for coordinating the real implementation of monitoring logic and local service. Adapter can check the legality of monitoring requirements and generate monitoring logic code.

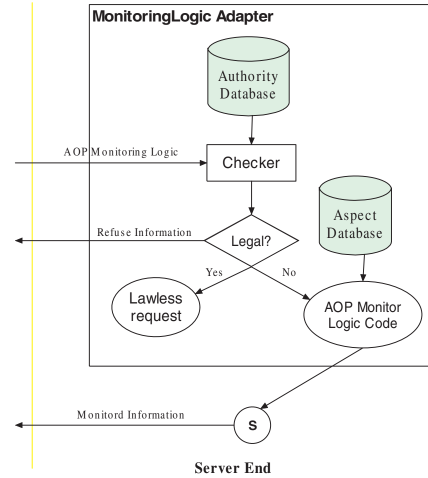

Fig. 6. Monitoring Adapter Running Graph

Figure 6 shows a monitoring adapter running graph, where Checker, Authority Database, Aspect Database etc. are involved. ActiveBPEL engine on the agent end adds AOP monitoring logic in XML file format into the SOAP message for calling basic services, and then transform the message to a monitoring logic adapter installed in end service S; Adapter calls authority record of each different-level agents from Authority Database while Checker check the legality of monitoring requirement (including whether the agent grade is high enough and aspect to be monitored is permitted or not etc.); If it an illegal monitoring requirement, adapter will produce error report and return refuse information; If it a legal monitoring requirement, adapter will call corresponding Aspect template from Aspect Database and produce AOP monitoring logic code for corresponding local service S implement code according to monitoring logic content.

VIII. Empirical Studies

In this section, we will discuss what tools we have designed and developed, how to simulate sample service, how to analyze monitored information, and how about the monitored result etc.

A. The Design and Implementation of WSMonitor

WSMonitor was implemented Java language in Eclipse platform, where SWT/JFace package and RCP(Rich Client Platform) technology are borrowed to create GUI and different desk applications. Right now, WSMonitor has following five function modules: (1) monitoring requirement expression; (2)monitoring logic generation; (3)monitoring adapter; (4)checker; (5)modifier.

B. Simulating Sample Service

We assume partner service S_1, S_2, S_3 shown in Figure 6 represents Hotel, Flight and Train service respectively. According to the WS-Policy Log and Timeout presented in section 2.2, Weaver will find the suited `TravelReservation` BPEL process and distribute the two policies to Hotel, Flight and Train service. Then partner services will generate following Log and Timeout AspectJ aspects with the help of Monitoring Logic Adapter.

```
Public aspect AOP_Log{
   pointcut log_\verb|TravelReservation|Interface();
     execution(*\verb|TravelReservation|..*());
   before():log_\verb|TravelReservation|Interface(){
     Date date=new Date();
     SimpleDateFormatdf=new;
     SimpleDateFormatat("yyyy-MM-dd hh:mm:ss");
     Signature s=thisJoinPoint.getSignature();
     MLogFile.println(df.format(date)+
     "[Monitor LOG]Entering:"+s.toString());}
   after():lo g_\verb|TravelReservation|Interface();{
     Date date=new Date();
     SimpleDateFormatdf=new;
     SimpleDateFormatat("yyyy-MM-dd hh:mm:ss");
     Signature s=thisJoinPoint.getSignature();
     MLogFile.println(df.format(date)+
     "[Monitor LOG]Exiting:"+s.toString());}}

public aspect AOP_timeout{
  pointcut timeout_\verb|TravelReservation|Interface():
    execution(* \verb|TravelReservation|.Hotel..*(..))
    ||execution(*\verb|TravelReservation|.Airline..*(..))
    ||execution(*\verb|TravelReservation|.Train..*(..));
  void around():timeout_\verb|TravelReservation|Interface(){
    Signature s=thisJoinPoint.getSignature();
    TimingContext tcx1=TimingContext.getContext();
  //System.out.println("[Monitor Time]Entering:
  //"+s.toString()+?at:"+tcx1.Start());
    proceed();
    TimingContext tcx2=TimingContext.getContext();
  //System.out.println("[Monitor Time]Entering:
  //"+s.toString()+?at:"+tcx2.Start());
    Date date=new Date();
    SimpleDateFormat df=
        new SimpleDateFormat("yyyy-mm-dd hh:mm:ss");
    System.out.println(df.format(date)+"\n"
        +"[Monitor Time]<"+s.getName()
        +"Fulfillment time:"
        +(tcx2.Commit()-tcx1.Start()+"ms>"); }}
  class TimingContext{
   private long m;
   private TimingContext()
    {m=System.currentTimeMillis();}
   public static TimingContext getContext()
    {return new TimingContext();}
   public long Start(){ return m; }
   public long Commit(){return m; }}
```

Log aspect defines a "*before and after*" type advice action (point-cut) "*log_*`TravelReservation`*Interface*". This aspect will record the start and the end of the message events. The Log monitoring information will be stored in MLogFile. Timeout aspect defines a "around" type advice action "*timeout_*`TravelReservation`*Interface*". This aspect will record the execution time of each arisen message event in the service process. The Timeout monitoring information will be stored in MTimeFile.

Finally, the monitoring result information (Table III) is stored in SIL.

TABLE III
THE TABLE OF SIL

Date	Time	File
2011-03-10	09:51:21	MLogFile
2011-03-10	09:51:33	MTimeFile
2011-03-11	01:45:04	MLogFile
2011-03-11	01:45:12	MTimeFile
2011-03-11	01:47:10	MLogFile
2011-03-11	01:47:32	MTimeFile

In Table 2, we can find that monitoring on the `TravelReservation` service has been implemented for three times. Detailed monitoring information files are shown below.

C. Analysis of Monitored Information

The monitoring system gets the property specification EMSC and MREG in section V and the monitoring results from SIL in section VII.2.

The value of variable p is decided by the original design business process, so it is also immutable for one given service. The checking algorithm above is only for a particular business process which includes the parallel structure and the selective structure; however, as these two structures are the most complicated BPEL activity forms, other process structures can be easily checked with appropriate algorithm change as well. For the `TravelReservation` service sample, in the initialization phase (line 1) of the algorithm Agency gets MLogFile and MTimeFile under some condition constraint from SIL. Agency finds that monitoring has been carried out for three times, so on line 2 variable m is initialized with the value" 3 ". As on line 3, Analyzer makes known from the EMSC that two BPEL activities can run in parallel, so it defines a static variable n with the value" 2 ". At the same time, Analyzer gets the time variables t_1, t_2, t_3, the condition variables flight, train and MREG (in Figure 4). There are three useful result items seen in Table II, so on line 1 of the algorithm main phase Analyzer gets the first result item (2008-08-10) from MLogFile. The item reveals that the execution message event sequence S_1 *=(reserveHotel(ReserveHotelIn),hotelReserved(HotelReservedIn), reserveAirline(ReserveAirlineIn), airlineReserved (AirlineReserved))*. At the same time, Analyzer records other condition information (such as " C:flight ") and the message event interval time value. Variable p is set with the value of variable n. From line 2 to line 7 is the processing segment that checks the current result item. After Comparing with MREG, Analyzer finds that there is a way "$0 > 7 > 8 > 11 > E$" with the way "$0 > 1 > 2 > 3 > 6 > E$" side-by-side in MREG (in Figure 5), so the message sequence is correct. Here, the symbol "$>$" represents the transfer from the left state to the right state. On second thoughts, the information on the same day from MTimeFile shows time t is $19541ms$ (the sum of $12305ms$ and $7236ms$). Because 19541ms is less than 20 seconds which is the value of t_1. So the way "$7 > 8$" is correct. Taking into account time t_2, Analyzer finds in this running instance t_2 equals the sum of 12305, 7236, 10928 and 8920, that is $39389ms$. That is also less than 60 seconds (given in section 2.1). Finally, Analyzer gets the monitoring conclusion information for the first time, the user's monitoring requirements have been considered carefully. The checking on the first result item has been completed. On line 8, the value of m is decreased with 1, the whole main phase loops into the next time checking task. Similarly, the next two monitoring result items can also be analyzed.

Monitoring requirement Reliability concerns the historical run-time information of the basic service Train. Since we

want to know the past 5 minutes' situation, SIL filters out the correlative date files. We assume now is 2008-08-11 01:48:00, so the latter two monitoring result items are considered, they are MLogFiles on 08-11 01:45:04 and 08-11 01:47:10 and MTimeFiles on 08-11 01:45:12 and 08-11 01:47:32. We find that t_3 equals $35431ms$ (the sum of 28426 and 7005). We also find that Train service is disabled in the last execution, as message event "trainReserved(TrainReservedIn)" was missed. In conclusion, the success invocation probability of the Train service is 50and the average response time t_3 is $35431ms$ in the past 5 minutes. The monitoring system will notify the user that the Train service invoked from current provider is not reliable. Then, the user will ask for another service candidate to provide a service holding the same function. This measure will help improving the quality of the composite service.

IX. Related Works

There are many works related to run-time monitoring of Web Service Composition. The work in [8] introduces a special monitor into the original BPEL process. BPEL is annotated with some assertion expressions. Then it is transformed into a new BPEL document with monitoring ability. The advantage of this method is that monitor is also basic service included in BPEL, so new BPEL can run on standard engine. But the extended BPEL essential influence the performance of the original process then our AOP method. With regard to the efficiency of AOP technology, article [14] makes a lot of research. The work described in [9, 12] also modifies ActiveBPEL engine for monitoring purpose. But [9] just adds some new general modules while our method benefits from AOP to separate the cross-cutting monitoring logic from the core business logic efficiently. A strongpoint of article [9] is that it considers not only the process instance monitoring, but also the process class monitoring, our method uses SIL to do the same thing. Article [12] also extends ActiveBPEL engine with AOP technology, but it does not show the way to generate the AOP aspect from the monitoring requirement expressly. Moreover, [12] uses the algebra specification language to represent the functional property of the service, this language is so sophisticated that a lot of rewriting and substituting regulations need to be imported when checking the monitoring information. In our method, EMSC and MREG property specifications are simpler, and they can express both the functional and some non-functional properties. Furthermore, our checking algorithm based on MREG can check both the message sequence and the time related property of the service easily.

The work in [10] uses UML 2.0 Sequence Diagrams to express Safety and Liveness property. Then the message event track is checked against Finite State Automaton. During the checking, negative and assert semantic is added to the message event in the automaton, while our checking algorithm utilizes some variable conditions to help the path tracing. In our method, MREG is given in order to easily checking the monitoring information. With the help of the designed checking algorithm, only basic data structure is applied during the checking, which reduces the cost overhead of the system. Article [11] gives a monitoring mechanism based on OWL-S, and extends the grammar of OWL-S to tackle the errors in the monitoring phase.

X. Conclusions and Future Work

In this paper, we have presented an integrated process to express monitoring requirement by WS-Policy, describe the property of WS by EMSC and MREG, perform run-time monitoring using AOP and check the monitoring information based on MREG. However, they are the first step towards a more powerful Web Service monitoring system. In the future, we plan to enhance the system to support more monitoring requirements such as security, transaction and so on. In fact, with the help of AOP, diversiform monitoring requirements can be achieved by adding more special designed AOP aspects. On the other hand, we have chosen BPEL as the description specification of the service, yet other web service workflow specifications may be also taken into account in the future.

References

[1] E. Newcomer, G. Lomow *Understanding SOA with Web Services*, Addison Wesley Press, Boston, 2004.

[2] BEA Systems, IBM, Microsoft, et al *Business process execution language for Web services version 1.1*, http://dev2dev.bea.com/technologies/webserv ices/BPEL4WS.jsp, February 2005.

[3] G. Kiczales, J. Lamping, et al *Aspect-Oriented Programming*, In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Finland, June 1997, pp.220-242.

[4] IBM, BEA Systems, Microsoft, et al *Web Services Policy Framework*, http://www.ibm.com/ developer works/library/specification/ws-polfram/, March 2006.

[5] G. Kiczales, E. Hilsdale, et al, *An overview of AspectJ*, In: Proceeding of the 15th European Conference on Objected-Oriented Programming (ECOOP 2001), 2001, pp.327-353.

[6] ActiveBPEL engine V4.0 *ActiveBPEL engine V4.0*, http://www. activebpel.org/infocenter/ActiveBPEL/v40/index.jsp,2007.

[7] ITU-T, Z.120 *Message Sequence Chart (MSC)*, ITU-T, USA, 1999.

[8] L. Baresi, C. Ghezzi, et al, *Smart Monitors for Composed Services*, In: Proceeding of the 2nd international conference on Service oriented computing (ICSOC), New York, USA, November, 2004, pp.193-202.

[9] F. Barbon, P. Traverso, M. Pistore, et al, *Run-time monitoring of instances and classes of web service compositions*, In: Proceedings of the 2006 IEEE International Conference on Web Services(ICWS), Washington, DC, USA, 2006,pp.63-71.

[10] Y. Gan, M. Chechik, et al, *Run-time Monitoring of Web Service Conversation*, In: Proceeding of the 2007 conference of the center for advanced studies on Collaborative research, Ontario, Canada,2007, pp.42-57.

[11] R. Vaculin, K. Sycara, *Semantic Web Services Monitoring: An OWL-S based Approach*, In: Proceeding of Hawaii International Conference on System Sciences (HICSS), Pittsburgh, USA, 2007,pp.313.

[12] D. Bianculli, C. Ghezzi, *Monitoring Conversational Web Services*, In: Proceeding of the 2nd International Workshop on Service oriented software engineering (IW-SOSWE), Dubrovnik, Croatia, 2007.

[13] A. Charfi, B. Schmeling, et al, *Reliable, Secure and Transacted Web Service Compositions with AO4BPEL*, In: Proceeding of the 4th European Conference on Web Services (ECOWS'06), Zurich, Switzerland, 2006.

[14] A. Houspanossian, M. Cilia. *Extending an Open-Source BPEL Engine with Aspect-Oriented Programming*, In: Proceeding of the Argentinean Symposium on Software Engineering (ASSE'05), Rosario, Argentina, August, 2005.