International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 3 (2011), pp. 924–931
© MIR Labs, http://www.mirlabs.net/ijcisim/index.html

Service Based System Monitoring Framework

Ajaya Kumar Tripathy[1] **and Manas Ranjan Patra**[2]

[1]Department of Computer Science Engineering,
Silicon Institute of Technology, Bhubaneswar, India

[2]Department of Computer Science,
Berhampur University, Berhampur, India

Abstract: **Web Service Based Systems (SBS) are essentially distributed in nature and in most cases facilitated through third party service providers. This necessitates monitoring of the provisioning of SBS at run-time. Further, monitoring of SBS in a non-intrusive and composition-platform independent manner is a real challenge. This paper proposes a framework for monitoring the compliance of a set of pre-specified requirements of a SBS. The requirements may include behavioral properties of SBS and/or assumptions that service providers may specify in terms of events that can be extracted from SBS at run-time. A Monitor Specification Language (MSL) has been developed to specify the properties of the system to be monitored at runtime. The language has the ability to specify boolean, statistical, and time-related properties. The specifications are automatically translated into executable C programs which act as runtime monitors to monitor the specified properties by capturing run-time events from the business layer, service layer and infrastructure layer of the SBS. However, in the proposed framework the SBS runs quite independent of the monitoring functionality in a non-intrusive manner.**

Keywords: Web Services, Service Based System, Run-Time Monitoring

I. Introduction

A web service is an application that exports a description of its functionality and makes it available using standard network technologies. These functionalities can be accessed through standard XML messages over internet [21]. Software systems that are composed of autonomous web services through a composition process are referred to as ”Web Service Based Systems” (SBS).

Research in Web Services spans over many interesting issues covering all the phases of the service life-cycle, e.g., service description, service discovery, selection and invocation, service composition, service advertisement, service negotiation, service security and reliability. In this paper, we focus on another interesting research topic, namely, SBS monitoring.

The ability to set up a SBS monitoring framework has been increasingly recognized as one of the essential preconditions for the deployment of a SBS. This is because the service composition assumptions and requirements of SBS that are verified at design time, prior to deployment and execution, can still be violated at run-time. This is especially true in case of web service based systems, which are most often developed by composing services that are made available by third parties. Such services are developed and managed autonomously and can change without notification leading to run-time problems. As and when such deviations are detected those must be captured and analyzed so as to take appropriate action. For instance, a bank may suddenly refuse to transfer money to a partner hotel's account. Such an unusual behavior of a bank requires immediate attention to know the actual reason behind it. Similarly, certain statistical information collected during run-time can reveal interesting phenomena. For instance, a sudden increase in the number of customers not accepting the offer of a travel agent may indicate that the agent's offer is considerably high compared to other business competitors. The occurrence of such events has to be reported as soon as possible so that the business analyst can take prompt action.

Some of the recent works have addressed different aspects of monitoring a SBS, e.g., [3, 7, 11, 15, 17, 19, 20]. In this paper we propose a novel solution to the problem of monitoring SBS.

In this paper, we describe a SBS monitoring framework which is independent of any business logic and service composition platform of SBS. The monitoring framework works in parallel with the SBS and allows for easy adaption of the business process. The SBS sends interesting events from the business layer, service layer (incoming/outgoing messages to/from the services used in the SBS) as well as from the infrastructure layer and feeds those into an event bus at runtime. A monitor observes the events from the event bus and accordingly monitors the functional and non-functional service composition assumptions and requirements of the SBS. Further, we provide a temporal logic based, RTML (Run-Time Monitor specification Language) type expressive language for specifying service composition assumptions, and functional as well as non-functional requirements of an SBS. The language allows for specifying boolean, static and time related properties. Beyond composition assumptions, we can also specify properties related to cross layer events, e.g., count the number of clients asking flight for *Venice* when there is a *carnival* (here *Venice* is a part of service level message where as *carnival* is a business layer event).

We have also designed a monitor generator which automatically generates a C program for the monitor which is deployed at run-time, thus reducing the design and implementation efforts [1].

The rest of the paper is structured as follows. In Section II, we describe the state of the art in the Web Service Technology and existing research approaches to the monitoring of SBS. Section III describes our monitor specification language. Section IV, describes an example scenario. Section V gives complete description of the framework. In Section VI we conclude our work and describe our future direction of research.

II. State of the Art

A. Web Service Technologies

Web Services are platform-independent, self-contained, self-describing, modular components that can be published, located and invoked over the Web. In order to achieve interoperability in such an heterogeneous framework, standards are of vital importance [14]. A whole stack of different standards has already been proposed with the aim of supporting the description, discovery, and interoperability of distributed, heterogeneous applications as services.

The functional description of a Web Service is provided by the Web Services Description Language (WSDL) [8]. WSDL describes a set of operations it offers, in-coming and out-going messages, and data types used by the Web Service (defined in terms of XML Schemas). Concrete protocol bindings and physical address port specifications complete a service description, providing a mechanism to locate a Web Service. WSDL defines what a Web Service does, not how it does; it characterizes the service only in terms of its interface, without providing any behavioral description. Such dynamic aspects are crucial for a complete understanding of a web service so that it can be recognized and used by autonomous applications.

B. SBS Monitoring

The necessity of specifying and monitoring different properties of composition assumptions as well as functional and non-functional requirements of SBS is widely recognized by industry and academia.

Lemana et al. [12] have proposed a SBS monitoring approach with the introduction of the language SLAng. This language is an extension of the existing business process languages. In this language properties are defined as a list of Quality of Service (QoS) parameters. At the implementation stage QoS parameters are assigned to the target business process, this leads to an intrusive approach. The target servers are required to support these QoS parameters. This approach becomes less extensible and flexible. The approach described in [17] creates monitoring agents to monitor the business process. These agents monitor the business process by gathering the network usage information. Another process evaluates the properties for any change in the process. This approach requires the business process to update constantly in order to adopt to new property requirements.

[1]An earlier version of this framework has been described in Ref. [19]. This paper is an extended version of Ref. [19] that: describe a revised version of the framework.

Barsi et al. [4] have proposed an approach for monitoring dynamic service composition with respect to guarantee terms expressed via assertions on services. This approach assumes composition process specified in BPEL. A guarantee term is verified by a call to an external service and the execution of the composition process waits until the monitor returns the result of the check. The composition process may continue or abort with an exception notification on whether the guarantee term is violated. The monitoring that it performs may effect the performance of the monitored system. This approach is intrusive to the normal operation of an SBS.

Another monitoring approach is presented by Baresi et al. [6]. This approach monitors both functional correctness of BPEL orchestration and quality of service agreements set between the service provider and the service consumer. They provide a language called WSCoL (Web Service Constraint Language) [5] which allows designers to specify constraints on BPEL orchestration. Appropriate external services called Monitoring managers are responsible for analyzing WSCoL constraints. The business logic is unaffected by monitor specification. Therefore, we can say the approach is non-intrusive at the specification time. But to allow the process to interact with the external monitors, additional BPEL code is added to the process at deployment time, this leads to an intrusive approach.

Lazovik et al. [13] presents a framework in which service requests are presented in a high-level language called XSRL (Xml Service Request Language). The framework monitors the execution of the request services. Designers can define three kinds of properties: (1) goals that must be true before transiting to the next state (2) goals that must be true for the entire process execution, and (3)goals that must be true for the process execution and evolution sequence. The framework loops between execution and planning. The latter activity is achieved by taking into account context and properties specified for the state-transition system. This makes it possible to discover whether a property has been violated by the previous execution.

Barbon et al. [3] present a monitoring approach extending the open-source Active BPEL engine. This approach defines an executable monitoring language RTML(Run-Time Monitor specification language) to specify properties of SBS to monitor, which is based on events and combines them exploiting past-time temporal logics and statistical functionalities. Monitors run parallel to BPEL (Business Process Execution Language for Web Services) [1] process as independent software modules that verify the guarantee terms by intercepting the input or output messages that are received or sent by the process. The framework supports automatic generation and deployment of monitors using guarantee terms specified in RTML. This is a nice approach but works only at service level for the BPEL processes.

Mahbub et al. [15] present an approach extending the WS-Agreement [2]. This approach supports monitoring of quality and functional properties. It introduces a new language to specify service guarantee terms in terms of:(1)events signifying invocation of operations of a service by the composition process of an SBS system and return from these executions, (2)events signifying calls of operation of the composition process of an SBS system by external services and

return from those executions, (3)the effect that events of either of the above kind have on the state of the SBS system or the service that it deploys. This language has been defined by a separate XML schema and is called EC-Assertion, which is based on Event Calculus (EC) [18] which is a first order temporal logic language. It is a nice approach but limited to only service level BPEL processes.
Tripathy et al. [19] present a non intrusive and SBS composition platform independent monitoring approach . This approach defines an executable monitoring language MSL(Monitor specification language) to specify properties of SBS to monitor, which is based on events and combines them exploiting past-time temporal logics and statistical functionalities. Monitors run parallel to SBS process as independent software modules that verify the guarantee terms by intercepting the input or output messages that are received or sent by the process and interested events from different layer of SBS. The framework supports automatic generation and deployment of monitors using guarantee terms specified in MSL. This is a nice approach but only can monitor instance level properties.

III. Monitor Specification Language

For monitoring SBS functional and non-functional properties and business assumptions (further we will refer these monitoring requirements as SBS properties), it is needed to formally specify these properties. The SBS properties that need to be monitored are expressed in a temporal logic based, executable language called MSL which is defined as follows:
In MSL, SBS properties are specified in terms of *events*. Here, an *event* is something that occurs at a specific instant in time in SBS domain. Events are categorized in to three categories: Service Layer Events, Business Layer Events and Infrastructure Layer Events. For example,

- Sent/received messages by the composed service to/from the atomic services used in the composition. These service to service message communication events are classified as service layer events.
- Something interesting occur in the SBS business domain which may significantly affect the business of the SBS are categorized as business layer event. For example, if a carnival is takes place in a city then the business of ”Travel Agent Service” of that city may increase. So, the happening of carnival is a interesting event for ”Travel Agent Service”. And since these event occur in the business layer, these events are categorized as business layer events for ”Travel Agent Service”.
- If a service provider uses 4 virtual machines to run the services. If the of one of the virtual machine is exceeding the normal load limit then in near future there is a chance of that virtual machine failure. So this may be a interesting event for a service provider. Since this event is related to service infrastructure, we categorize the events from infrastructure layer as infrastructure layer event.

The grammar for specifying events in MSL is as follows:
$event ::= \; eventName \mid eventName.(condition)$
$eventName ::= \; [a-z][a-zA-Z0-9]*$
$condition ::= \; type \;\; var \;\; cond \;\; value \mid$
$\quad condition \; V \; condition \mid condition \;\; \wedge \;\; condition$
$type ::= \; int \mid double \mid string$
$var ::= \; [a-zA-Z0-9]+$
$cond ::= \; \neq \mid = \mid > \mid <$
$value ::= \; [-+][0-9]+ \mid [-+][0-9]+.[0-9]* \mid$
$\quad [a-zA-Z]*$

Semantics From previous discussion we can assume that, finally for the framework the event is a message with a message name with zero or more internal variable with variable name, variable type and variable value. This part of the grammar facilitates the specification of events with the condition on the internal variables of the event, $eventName$ specifies the message name, $type$, var and val specifies internal variable type (which may be int/double/string), internal variable name (which is a string) and internal variable value (which may be an integer number/a real number/a string) respectively. Condition is defined as *type var cond value*: where $type$ is the data type of the variable($int/double/string$), var is the name of the variable, $cond$ is a logical condition ($=\;/\;\neq\;/\;>\;/\;<$) on variable and $value$ is a value(number/string) to compare with the variable value.
The following grammar defines the boolean, temporal and statistical formulas. We distinguish boolean formulae *b*, which monitor properties that can be either true or false, and numeric formulas *n*, which monitor properties that define a numerical value (which include temporal and statistical formulae).
$b ::= \; event \mid b \; \vee \; b \mid b \; \wedge \; b \mid b \Rightarrow b \mid \neg \, b \mid n = n \mid n > n \mid Y \, b \mid O \, b \mid H \, b \mid b \, S \, b$
$n \; ::= \; C(b) \mid T(b) \mid b?n \; : \; n \mid n+n \mid n-n \mid n*n \mid n \, / \, n \mid NUM$
$NUM ::= [0-9]* \mid [0-9]+.[0-9]*$
A boolean formula can be an event, or an event with some comparison between internal variables of the event, or a past LTL [10] formula (operators *Y, O, H* and *S*), or a comparison between numeric formulas, or a logic combination of other boolean formulas. A numeric formula can be either a counting formula (operator *C*), or a time measurement formula (operator *T*), or an arithmetic operation on numeric formulas.
The operators $\vee$, $\wedge$, $\neg$, $=$, $>$, $<$ and $\Rightarrow$ have the same meaning as $logical \vee$, $logical \wedge$, $logical \neg$, $logical \; =$, $logical \; >$, $logical \; <$ and $logical \; \Rightarrow$. Past LTL formulas have the following meaning: *Y b* means ”*b* was true in the previous step”, *O b* means ”*b* was true (at least) once in the past”, *H b* means ”*b* was true always in the past” and *b1 S b2* means ”*b1* has been true since *b2*. Numeric formula *C(b)* counts the number of times that boolean formula *b* has been true since the creation of the process instance. Formula *T(b)* counts the sum of the time-spans the formula *b* remains true.

IV. An Example

Intra City Motorbike Rental Service (MRS) is a SBS example scenario (see Figure 1), which is used further in this article to explain the proposed approach in a better way.Consider the MRS act as a broker offering its customers to avail the bikes

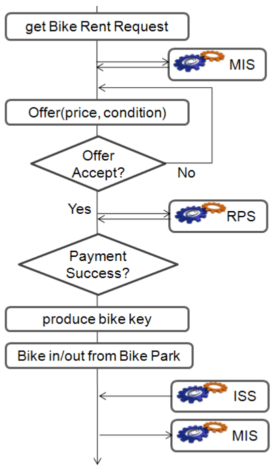


Figure. 1: Intra City Motorbike Rental Service: Scenario

on rent. The bikes are provided by different bike rental companies. A customer can book a bike online for a particular location in a city and then can avail the bike from a bike park nearer to that requested location.Let the MRS is implemented as service based system that consist of a service composition process that interacts with following web services:

- *Motorbike information services* (MIS) which are provided by different motorbike rental companies. It maintains registries of bikes, check the availability for rent request and if available it issues a available bike identification number.
- *Identification Sensor Services* (ISS) which are provided by different motorbike parks to sense motorbike identification and customer identification as they are driven in or out of motorbike park and inform MRS accordingly.
- *Motorbike Rent Payment Services* (RPS) which are provided by different banks at different motorbike parks to collect the rent of motorbikes from MRS clients after use of motorbikes.
- *Client Services* (CS) that provide MRS with a frontend that handles interaction with the end-user.

In this explanatory example scenario, we assume the message flow of MRS are as follows. CS activate the MRS for a motorbike rent by sending a rent request message mentioning the location at which the motorbike is required and the personal identification i.e, `rentRequest(UID, location)`. Then MRS checks the availability of motorbike at the requested location by invoking MIS (by sending `isAvailable(location)` message). MIS respond with the availability status: saying no by sending `notAvailable` message / saying yes by sending a available message with a available motorbike identification i.e, `available(mBikeID)`. If MRS gets a available response then acknowledge CS by sending a price and condition offer message i.e, `offer(price, condition)` otherwise acknowledge CS as motorbike is not available by sending `notAvailable` message. If CS accepts the offer of MRS, then it sends `startPayment(accInfo)` message to MRS. Other wise it rejects the offer by sending *offerReject* message. Then MRS sends a new offer to CS. In case of offer acceptance, MRS invokes RPS to get the payment by sending `makePayment(accInfo, cost)`. RPS acknowledges MRS about payment success/fail by sending `paymentSucc`/`paymentFail`. By getting `paymentSucc`, MRS acknowledges MIS that the bike is booked (by sending `bikeBooked(mBikeID)`). Then the MIS updates its database accordingly and sends a bike identification number with unlock key to CS (by sending `bikeKey(mBikeID, unlockKey)` message). When bike enter/exit to/from a bike park, ISS inform MRS by sending `bikeIn(mBikeID, UID)`/`bikeOut(mBikeID, UID)`, subsequently inform MIS to update its database by sending `bikeIn(mBikeID, UID)`/`bikeOut(mBikeID, UID)`. Figure 2 depicts the message flow of MRS as explained above.

Despite of the simplicity of the domain, due to the business need and/or providing better service, the MRS provider may want to monitor the following properties.

The class-I of properties are those that constrain the correct behaviors of the composition. For example: Property 1: A bike should not enter to a bike park unless it is dispatched from any bike park. (Violation of this boolean property indicates ISS is malfunctioning in some bike park.)

Property 2: Allow the client to pay only if there is a bike available in a bike park nearer to the requested location by the client. (Non-violation of this boolean property ensure that MRS accept rent if and only if there is at least one free (not booked) bike available at the requested location.)

Property 3: A bike should not go out of a bike park if a key for that bike is not issued by the MRS prior. (Violation of this boolean property indicates MRS is malfunctioning.)

Property 4: A person can take a bike out of the bike park if and only if a bike key is issued to him/her earlier and he/she has already paid the rent. (Non-violation of this boolean property ensure that MRS accept rent and issued a bike key to the person who is taking the bike out of the bike park.)

The MRS provider may also be interested in counting number of times a given event occurs in the execution of MRS. For example:

Property 5: Count the number of offers offered to the user before the user accepts. (This information may be helpful for the MRS to know the most popular rent offer.)

The MRS provider may also be interested in measuring the time spent to perform certain activities, for example:

Property 6: Measure the MRS transaction completion time. (Sudden increase/decrease of MRS transaction time may lead to some functional problem.)

Property 7: Measure the MRS transaction completion time excluding payment time. (Sudden significant increase/ decrease of MRS transaction completion time may lead to some functional problem, excluding RPS.)

The MRS provider may also be interested in collecting statistical information, related to all instances of MRS. For example:

Property 8: The rent request is never refused by MRS.

Property 9: Count the number of times the bike is unavailable.

Property 10: Measure the average MRS transaction completion time. (This can give a statistical idea about MRS transaction completion time.)

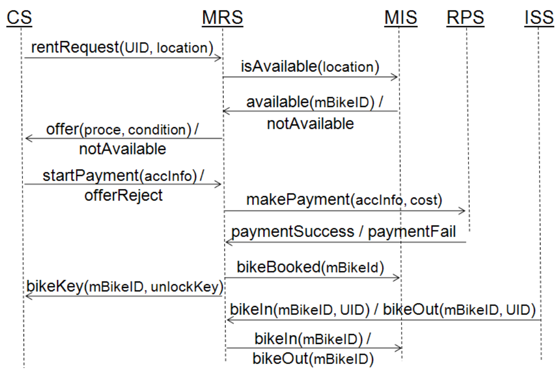

CS
MRS
MIS
RPS
ISS
rentRequest(UID, location)
isAvailable(location)
available(mBikeID) / notAvailable
offer(proce, condition) / notAvailable
startPayment(accInfo) / offerReject
makePayment(accInfo, cost)
paymentSuccess / paymentFail
bikeBooked(mBikeId)
bikeKey(mBikeID, unlockKey)
bikeIn(mBikeID, UID) / bikeOut(mBikeID, UID)
bikeIn(mBikeID) / bikeOut(mBikeID)

Figure. 2: MRS message flow diagram.

The MRS provider may also be interested in monitoring properties, related to a particular subset instances of MRS, satisfying certain constraints on internal variables of events. For example: Property 11: Number of times bike is not available at "Barmunda bike park". (This will help to take decision to increase number of bikes in "Barmunda bike park".)
Finally, the MRS provider may also be interested in monitoring properties related to events coming from different layers. For example: Property 12: Number of times bike is not available at "Puri bike park" during "Car Festival". (This will help to take decision to increase number bikes in "Puri bike park" during "Car Festival".)

V. Monitoring Framework

Our SBS monitoring framework has been designed with the objective to support three different key monitoring features for SBS. The three key features of this approach are: (i) the monitoring is performed in parallel with the operation of an SBS without affecting its performance, (ii) non-intrusive SBS monitoring (i.e, monitoring without interfering the SBS process execution or with out changing the original SBS), and (iii) the monitoring framework is independent of the service composition platform.
In this framework a human user (typically, provider of an SBS) can request to monitor the runtime operations of a system to see whether certain specified properties are satisfied or not and indicate any deviations once they are detected.
Our monitoring framework is depicted in Figure 3. Here, it is assumed that at run-time a process execution engine executes the composition process of an SBS and delivers its functionality while capturing events from all layers (business layer, service layer and infrastructure layer)and pushes the events into an *Event Bus*.
The framework has 3 main components, namely an *Event Bus*, a *User Interface* and a *Monitoring Engine*. Figure 3 shows a high level representation of the proposed framework.

A. Event Bus

The "***Event Bus***" collects events from the SBS and puts the events in an event queue. The monitors consume the events from the queue. The types of *events* the *Event Bus* receives are: messages received from the composition process or sent to the composition process by one of its partner services, interesting events from business layer/Infrastructure layer.

The format of the events are as follows:
[sourceID]eventName{[varType:varName=val]}*
where, $sourseID$ is the source identification number (If the *event* coming from service layer then *sourceID* is the process instance number of the SBS. If the *event* is coming from infrastructure layer then *sourceID* is -1. The *sourceID* of business layer is 0.), $eventName$ is the name of the event, $varName$ is the name of a internal variable of the *event*. $varType$ is the type of the variable $varName$ (different $varType$ are $int/double/string$), val is the value of the variable $varName$. One *event* can have no or some internal variables. Each variable is in the form of $varType : varName = val$. Two variables are separated by a ",".
The types of *eventName* the event bus accepts are as follows:
$partnerService\_I\_messageName$
$partnerService\_O\_messageName$
where, $partnerService$ is the name of the partner service, I: indicates that the message is a input message for the partner service, O: indicates that the message is an output message for the partner service, $messageName$ is the name of the message.
Examples of events:
`[1]CS_O_rentRequest{string:UID=XYZ67432, string:location=Puri}`:
This is a service layer event with process instance number 1 with event name $CS\_O\_rentRequest$ with two internal variables "UID" and "$location$". This event name indicates that $rentRequest$ is a out going message from the partner service *CS* of *MRS* SBS.
`[-1]virtualMachineLoad{int:load=60}`: This is an example of infrastructure layer event with event name $virtualMachineLoad$ with one internal variable "*load*".
`[0]cartFestivalStart`:
This is an example of business layer event with event name $cartFestivalStart$.
MSL Specification of MRS Properties:
Some of the properties we have introduced in Section IV can be defined by the following MSL formulae:
Property 1: A bike should not enter to a bike park unless it is dispatched from any bike park.
$\texttt{MRS\_I\_bikeIn} \Rightarrow O(\texttt{MRS\_I\_bikeOut})$
Property 2: Allow the client to pay only if there is a bike available in a bike park nearer to the requested location by the client.
$\texttt{MRS\_I\_startPayment} \Rightarrow O(\texttt{MRS\_I\_available})$
Property 3: A bike should not go out of a bike park if a key for that bike is not issued by the MRS prior.
$\texttt{MRS\_I\_bikeOut} \Rightarrow O(\texttt{MRS\_O\_bikeKey})$

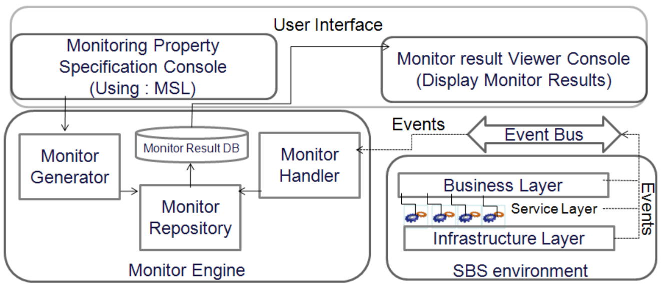


Figure. 3: SBS monitoring Framework

Property 4: A person can take a bike out of the bike park if and only if a bike key is issued to him/her earlier and he/she has already paid the rent.
$\texttt{MRS\_I\_bikeOut} \Rightarrow (O(\texttt{MRS\_O\_bikeKey})$
$\Rightarrow O(\texttt{MRS\_I\_paymentSuccess}))$
Property 5: Count the number of offers offered to the user before the user accepts.
$C(\texttt{MRS\_I\_notAvailable})$
Property 6: Measure the MRS transaction completion time.
$T(\neg(\texttt{MRS\_O\_bikeKey})\ S\ (\texttt{MRS\_I\_rentRequest}))$
Property 7: Measure the MRS transaction completion time excluding payment time.
$T(\neg(\texttt{MRS\_O\_bikeKey})\ S\ (\texttt{MRS\_I\_rentRequest}))$
$-$
$T(\neg(\texttt{MRS\_I\_paymentSuccess} \vee \texttt{MRS\_I\_paymentFail})$
$S\ (\texttt{MRS\_O\_makePayment}))$

B. Monitor Engine

Monitor engine is the most important and most complex part of the framework. It has 4 main components, namely *Monitor Generator* which generates the monitors, *Monitor Repository* which stores all the monitors, *Monitor Handler* which receives *events* from *event bus* and sends the received events to appropriate monitors in the *Monitor Repository* and *Monitor result DB* stores the results of the monitors.

1) Monitor Generator

is a MSL compiler, designed using Bison [9] as parser generator and Flex [16] as lexical analyzer generator. This compiler translates the MSL specified formula to a C program named `MonitorID.c` and stores it in the *Monitor Repository*, where `ID` is the serial number of the monitor. Also the name of the created monitor (i.e, MonitorID) is registered (i.e, stored) in the *Monitor Registory* (i.e, a registry which stores name of created monitors). `MonitorID.c` contains a parse tree of the MSL formula and a parse tree update function implementing Algorithm 1. Each node of the parse tree along with its child sub trees represent a formula. Each node of the parse tree stores the formula values (truth/numerical). Hereafter, ”node value” would mean the value of the formula it represents. The root node stores the value of the total formula i.e, value of the monitor. When the *Monitor Handler* wakes up a monitor by sending an *event*, the Update-Tree function updates the formula value at each node of the parse tree of the corresponding monitor according to the following formula value update rules.

Formula Update Rules : Update-Tree function uses following rules to update the parse tree node values i.e, the value of sub formulas of a total formula.

1. $fVal(condition)\, i,e.\, fVal(\texttt{type var}\, cond\, \texttt{value})$

   $i.e,\ Formula\ value\ of\ acondition := true$

   ”If one of the internal variable of the occurring event has *type* = `type`, *name*=`var` and has the *value* satisfying *cond*(i.e, $= \mid \neq \mid > \mid <$) comparison to `value`”.

2. $fVal(condition\ \wedge\ condition) :=$

   $fVal(condition)\ \wedge\ fVal(condition)$

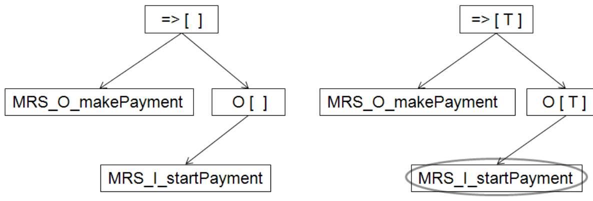


Figure. 4: Left: Parse Tree, Right: Updated Parse Tree

3. $fVal(event) :=\ true$ ”If *event* is occurring”.
4. $fVal(event.condition) :=$

   $fVal(event)\ \wedge\ fVal(condition)$
5. $fVal(Y\ b) := oldfVal(b)$
6. $fVal(O\ b) := old\ fVal(Ob)\ \vee\ fVal(b)$
7. $fVal(H\ b) := old\ fVal(Hb)\ \wedge\ fVal(b)$
8. $fVal(b1\ S\ b2) :=$

   $fVal(b2)\ \vee\ (old\ fVal(b1\ S\ b2)\ \wedge\ fVal(b1))$
9. $fVal(C\ b)$:=if $fVal(b)$ then $(old\ fVal(C\ b){+}1)$ else $fVal(C\ b)$
10. $fVal(T\ b)$:=if $fVal(b)\ \wedge\ old\ fVal(b)$ then $(old\ fVal(T\ b)\ +\ elapsed)$ else $old\ fVal(T\ bf)$

Note

- *fVal* of $(b1 \vee b2) \mid (b1 \wedge b2) \mid (b1 \Rightarrow b2) \mid \neg\, b1 \mid (b1 = b2) \mid (b1\ >\ b2)$ are as per the normal logical operator rule. For example:

  $fVal(bf1\ \wedge\ bf2) := fVal(bf1)\ \wedge\ fVal(bf2)$
- *fVal* of $b?n1 : n2 \mid n1{+}n2 \mid n1{-}n2 \mid n1{*}n2 \mid n1\ /\ n2$ are as per the standard mathematical rules.

Algorithm 1 Update-Tree(*event*, Monitor)

```
1: if eventName is matching with a node of the parse tree
   of Monitor.c then
2:     Update node values of all nodes of the parse tree using
       ”Formula Updating Rules”.
3:     Store root node value as the current monitor result of
       this monitor in Monitor result DB.
4: end if
```

Example: The following example conceptually shows the function of the Update-Tree algorithm. As an example let us conceptually demonstrate the monitor for Property 1 as mentioned in Section IV i.e, the conceptual structure of the generated ”parse tree” and the function of Update-Tree on the generated ”parse tree”.

Property : Payment process starts only after client accepts the offer.
MSL specification: $\texttt{MRS\_O\_makePayment} \Rightarrow$
$O(\texttt{MRS\_I\_startPayment})$
Left hand side of the Figure 4 shows conceptually the structure of parse tree and the right hand side figure of Figure 4 shows the effect of Update-Tree function on it after receiving the *event*
`MRS_I_startPayment{int:accInfo=546718}`.

2) Monitor Handler

is responsible for receiving new *events* from *Event Bus*, creating the required new instances of the existing monitors in the *Monitor Repository* and waking up appropriate monitors to consume the incoming *event*. The following Event-Monitor-Handler algorithm does all these tasks.

Algorithm 2 Event-Monitor-Handler(*event*)

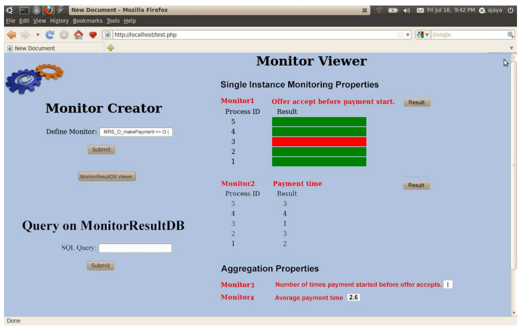
```
1:  Find $sourceID$ of the event.
2:  if $sourceID \leq 0$ then
3:      Wake up all instances of all monitors stored in the Monitor Repository to consume the incoming event.
4:  else
5:      for each MonitorID stored in the Monitor Registory do
6:          if $sourceID$ ia a new $sourceID$ then
7:              Add the $sourceID$ in the $sourceID$ list against the MonitorID.c
8:              Create a new instance of MonitorID.c and save this instance as $MonitorID\_sourceID$.c in the Monitor Repository.
9:          end if
10:     end for
11: end if
12: Update-Tree(event, MonitorID_$sourceID$.c)
```

C. Aggregation Functions

To monitor the properties related to multiple instances of SBS systems two aggregation functions named **ForAll** and **AddAll** are added in the framework. These functions are designed above the results of monitors (specified using MSL) stored in *MonitorResultDB*. In addition to that a run time sql query on *MonitorResultDB* is added in the framework to facilitate the user to make some reasoning on the outputs of the running monitors if necessary.

- **ForAll**(b): If b (a boolean MSL formula) is true for all instances of SBS process then **ForAll**(b) is true, otherwise false.
- **AddAll**(n): It add value of n (a numeric MSL formula) for all instances of SBS process.

Using these two aggregation functions now we can specify and monitor properties related to multiple running instances of SBS process. For example, property 8, 9, 10, 11 & 12 for MRS can be specified using these two aggregation functions and MSL as follows.
Property 8: The rent request is never refused by MRS.
ForAll($\texttt{MRS\_O\_notAvailable}$)
Property 9: Count the number of times the bike is unavailable.
AddAll($C(\texttt{MRS\_O\_notAvailable})$)
Property 10: Measure the average MRS transaction completion time.
AddAll($T(\neg(\texttt{MRS\_O\_bikeKey})\ S\ (\texttt{MRS\_I\_rentRequest}))$)
/ **AddAll**($C(\texttt{MRS\_I\_rentRequest})$)
Property 11: Number of times bike is not available at "Barmunda bike park".
AddAll($C(\texttt{MRS\_I\_rentRequest} \wedge O(\texttt{MRS\_I\_rentRequest}.(location = \text{"}Barmunda\text{"})))$)
Property 12: Number of times bike is not available at "Puri bike park" during "Car Festival".
ForAll($(\neg\texttt{CartFestivalEnd}\ S\ \texttt{CartFestivalStart})\ ?C(\texttt{MRS\_I\_rentRequest.(location} = \texttt{Puri)}) : 0$)


New Document - Mozilla Firefox
Fri Jul 16, 9:42 PM
ajaya
File Edit View History Bookmarks Tools Help
http://localhost/test.php
Google
New Document
Submit
MonitorResultDB Viewer
Submit
Result
Result
Done

Monitor Creator

Define Monitor: MRS_O_makePayment => O (

Query on MonitorResultDB

SQL Query:

Monitor Viewer

Single Instance Monitoring Properties

Monitor1 Offer accept before payment start.

Process ID	Result
5	🟩
4	🟩
3	🟥
2	🟩
1	🟩

Monitor2 Payment time

Process ID	Result
5	3
4	4
3	1
2	3
1	2

Aggregation Properties

Monitor3 Number of times payment started before offer accepts. 1

Monitor4 Average payment time 2.6

Figure. 5: User Interface Snapshot

D. User Interface

The "$User\ Interface$" gives access to the monitoring service for the users. This interface provides a web page for defining new monitors using MSL & aggregation functions, for viewing the monitoring results of the deployed monitors, viewing *MonitorResultDB* structure and firing SQL queries on *MonitorResultDB*. For instance, in Figure 5 shows snapshot of one of the web pages of User Interface.

VI. Conclusion and Future Work

In this paper, we have presented a framework for monitoring several properties of web service based systems. An event based approach has been proposed that separates business logic from the monitoring functionality and supports cross-layer SBS monitoring. The proposed framework does not depend on the service composition platform. Further, a monitoring language has been developed to formally specify the properties of a service based system. The specification is automatically translated into an executable C program which is used by the framework while monitoring the specified behavior of the system.
We continue our work to extend the proposed framework to support the monitoring of service level agreements, service based system failure-handling, repairing and adaptation triggered by information provided by the monitors. Further, we plan to provide an experimental evaluation of the usability and practical effectiveness of the proposed framework in different application domains.

Acknowledgment

We thank the team members of the Service Oriented Application unit of FBK, Trento, Italy for their valuable suggestions and validation of concepts incorporated in this work.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith,

S. Thatte, et al. Business process execution language for web services, 2003.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement). In *Global Grid Forum*, 2004.

[3] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. 2006.

[4] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM, 2004.

[5] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. *Service-Oriented Computing-ICSOC 2005*, pages 269–282, 2005.

[6] L. Baresi and S. Guinea. A dynamic and reactive approach to the supervision of BPEL processes. In *Proceedings of the 1st conference on India software engineering conference*, pages 39–48. ACM, 2008.

[7] T. Chau, V. Muthusamy, H. Jacobsen, E. Litani, A. Chan, and P. Coulthard. Automating SLA modeling. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 126–143. ACM, 2008.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, 2001.

[9] C. Donelly and R. Stallman. Bison: The YACC-Compatible Parser Generator. Free Software Foundation. *Cambridge, MA*, 1992.

[10] E. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, 8:995–1072, 1990.

[11] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[12] D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements. In *Proc. of the 9th IEEE Workshop on Future Trends in Distributed Computing Systems-FTDCS*, pages 100–106. Citeseer, 2003.

[13] A. Lazovik, M. Aiello, and M. Papazoglou. Associating assertions with business processes and monitoring their execution. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 94–104. ACM, 2004.

[14] F. Leymann. Web services: Distributed applications without limits. *Business, Technology and Web, Leipzig*, 2003.

[15] K. Mahbub and G. Spanoudakis. Monitoring WS-Agreement s: An Event Calculus–Based Approach. *Test and Analysis of Web Services*, pages 265–306, 2007.

[16] V. Paxson et al. Flex–fast lexical analyzer generator. *Free Software Foundation*, 1988.

[17] A. Sahai, V. Machiraju, M. Sayal, A. Van Moorsel, and F. Casati. Automated SLA monitoring for web services. *Management Technologies for E-Commerce and E-Business Applications*, pages 28–41, 2002.

[18] M. Shanahan. The event calculus explained. *Artificial intelligence today: Recent trends and developments*, page 409, 1999.

[19] A. K. Tripathy and M. R. Patra. An event based, non-intrusive monitoring framework for web service based systems. In *Proceedings of the* 6^{th} *International Conference on Next Generation Web Service Practices: NWeSP 2010*, pages 201–206. IEEE, 2010.

[20] A. K. Tripathy and M. R. Patra. Modeling and monitoring sla for service based systems. In *In Proceedings of the International conference on Intelligent Semantic Web - Services and Applications:ISWSA 2011, Jordan*, pages 60–65. ACM, 2011.

[21] W3C. Web services description requirements, October 2002. http://www.w3.org/TR/ws-desc-reqs/.

Author Biography

Ajaya Kumar Tripathy holds a Professional M.Tech. Degree in e-Government from Trento University, Italy in 2008. In 2007 he has completed the M.Tech. Degree in Computer Science from Utkal University, India. From 2008 - 2010 he was a PhD student in SOA Research Unit at Fondazione Bruno Kessler (FBK) in Trento, Italy. Mr. Tripathy also worked as a research assistant (internee) in Indian Statistical Institute, India and Createnet Research Center, Trento, Italy. Currently he is a faculty in the Department of Computer Science and Engineering, Silicon Institute of Technology, Bhubaneswar, India. As a researcher he has published research articles on Web Services. His research interests include Run-Time monitoring of Web Services and SLA monitoring for Web Services and Pattern Recognition.

Dr. Manas Ranjan Patra holds a Ph.D. Degree in Computer Science from the Central University of Hyderabad, India. Currently he is heading the Post Graduate Department of Computer Science, Berhampur University in India. He has about 23 years of experience in teaching and research in different areas of Computer Science. He had visiting assignment to International Institute for Software Technology, Macao as a United Nations Fellow and for sometime worked as assistant professor in the Institute for Development and Research in Banking Technology, Hyderabad. He has about 75 international and national publications to his credit. His research interests include Service Oriented Computing, Software Engineering, Applications of Data mining and E-Governance. He has presented papers, chaired technical sessions and served in the technical committees of many International conferences.