Monitoring the QoS for Web Services

Liangzhao Zeng, Hui Lei, and Henry Chang

IBM T.J. Watson Research Center Yorktown Heights, NY 10598
lzeng,hlei,hychang@us.ibm.com

Abstract. Quality of Service (QoS) information for Web services is essential to QoS-aware service management and composition. Currently, most QoS-aware solutions assume that the QoS for component services is readily available, and that the QoS for composite services can be computed from the QoS for component services. The issue of how to obtain the QoS for component services has largely been overlooked. In this paper, we tackle this fundamental issue. We argue that most of QoS metrics can be observed/computed based on service operations. We present the design and implementation of a high-performance QoS monitoring system. The system is driven by a QoS observation model that defines IT- and business-level metrics and associated evaluation formulas. Integrated into the SOA infrastructure at large, the monitoring system can detect and route service operational events systemically. Further, a model-driven, hybrid compilation/interpretation approach is used in metric computation to process service operational events and maintain metrics efficiently. Experiments suggest that our system can support high event processing throughput and scales to the number of CPUs.

1 Introduction

Web services are autonomous software systems identified by URIs which can be advertised, located, and accessed through messages encoded according to XML-based standards such as SOAP, WSDL and UDDI. Web services encapsulate application functions and information resources, and make them available through programmatic interfaces, as opposed to the human-computer interfaces provided by traditional Web applications. Since they are intended to be discovered and used by other applications across the Web, Web services need to be described and understood in terms of both functional capabilities and non-functional, i.e., Quality of Service (QoS) metrics.

Given the rapidly increasing number of functionally similar Web services available on the Internet, there is a need to be able to distinguish them using a set of well-defined QoS metrics. Further, in situations where a number of component services are aggregated to form a composite service, it is necessary to manage the QoS for the composite service based on the QoS for individual component services. Most systems for QoS-aware service selection [2][4][5][6] and management [22][23] assume that the QoS information for component services is pre-existing. How to obtain this QoS information is largely overlooked. In this paper, we try to address this fundamental issue.

In general, QoS metrics can be classified into three categories, based on the approaches to obtaining them:

- Provider-advertised metrics. This type of metrics is usually provided by service providers, which is subjective to service providers. One example is the execution prices advertised by service providers.
- Consumer-rated metrics. This type of metrics can be computed based on service consumer's evaluations and feedback, which is therefore subjective to service consumers. For example, the service reputation is considered average according to service consumers' evaluations.
- Observable metrics. This type of metrics can be observed, i.e., computed, based on monitored service operational events, which is objective to both service providers and consumers. Majority of QoS metrics in fact can be observed, including those of IT level and of business level. IT-level metrics include service execution duration, reliability, and etc. At business level, metrics are usually domain-specific and require some modeling efforts to define the formulas [5]. For example, the metric "forecast accuracy" for forecast services in supply chain management is usually defined as:

$$\sum_{i=0}^{n} \frac{|\, actualDemand_i - forecastDemand_i \,|}{actualDemand_i}$$

  In order to compute such a metric value, both actual demand and forecasted demand need to be monitored. It should be noted that the metric value needs to be recomputed whenever the execution of a service instance is completed.

In this paper, we focus on these observable metrics. We adopt a model-driven approach to the definition and monitoring of Web service QoS metrics. We introduce an observation metamodel that specifies a set of standard building blocks for constructing various QoS observation models. An observation model defines the specific QoS metric types that are of interest, as well as rules on when and how the metric values are computed.

An observation model has to be executed by a QoS monitoring system. There are two main issues in designing and implementing such a monitoring system:

- Service monitoring architecture. To detect service operational events, service monitoring needs to be integrated into the SOA infrastructure at large. It is important to leverage existing components in the SOA infrastructure, and to enable detection and routing of the service operational events systematically.
- QoS metric computation. There are three main challenges in designing an efficient computation runtime:
  - *High volume of service operational events.* In large-scale SOA solutions, there can be thousands of business process instances concurrently running. Even if each process instance generates only one operational event per second, there may be thousands of events that need to be processed per second. It is thus important for the runtime to support high event-processing throughput.
  - *Complexity of metric computation.* The ECA rules for metric computation actually create a workflow representable as statecharts. The complexity of metric computation stems from two aspects: the topology of the statecharts and

the formulas for computing the metric values. For example, hundreds of expressions may be triggered directly or indirectly to update a series of metric values due to the occurrence of a single service operational event. Unlike most complex event processing systems that focus on event filtering and composite event detection, metric computation is concerned with the expression evaluation triggered by events. The potentially large number of expressions that need to be evaluated significantly increases the overall complexity of the system.

- *Metric value persistence*. QoS metric values need to be saved in persistent storage after they are computed/updated, in order to make them available for other components (e.g., service selectors). Given the high volume of service operational events and the complexity of metric networks, an appropriate persistence mechanism is required, in order to support both efficient metric value persistence and queries.

Given QoS metrics are time-critical and time-sensitive information, it is important to develop a high performance metric computation engine that can compute/update metric values in real time.

In order to tackle the above challenges, we design and implement a service *Q*oS monitoring system. It provides a user-friendly programming model that allows users to define the QoS metrics and associated ECA rules. It enables declarative service QoS monitoring in the SOA infrastructure. It employs a collection of model-analysis techniques to improve the performance of metric computation. In a nutshell, the main contributions of this paper are:

- *Monitoring-enabled SOA Infrastructure*. Building upon our previous work on semantic service mediation [21] and semantic pub/sub [18] that enables flexible interoperation among Web services, we further enrich the SOA infrastructure to enable declarative event detection and routing in dynamic and heterogeneous environments. Such an extension allows the QoS for Web services to be monitored with small programming efforts.
- *Efficient QoS computation*. We present a novel hybrid compilation-interpretation approach to QoS metric computation. A series of model-analysis techniques is applied to improve event processing throughput. At build time, custom executable code is generated for each ECA rule. The custom code is more efficient to execute than generic code driven by ECA rules. At runtime, model-driven mediators interpret a transformed observation model to invoke generated code at appropriate points. Also, model-driven planning is adopted to enable wait-free concurrent threads for metric computation, which eliminates the overhead of concurrency control. Our experiments suggest that the system not only can support high event throughput but also can scale to the number of CPUs.

The rest of this paper is organized as follows. Section 2 presents the QoS observation metamodel. Section 3 illustrates the SOA infrastructure that enables service QoS monitoring. Section 4 discusses the design of a high performance metric computation engine. Section 5 briefly describes the implementation and experimentation. Following discussion on related work in Section 6, Section 7 provides concluding remarks.

2 QoS Observation Model

In the presence of multiple Web services with overlapping or identical functionality, service requesters need some QoS metrics to distinguish one service from another. We argue that it is not practical or sufficient to come up with a standard QoS model that can be used for all Web services in all domains. This is because QoS is a broad concept that encompasses a large number of context-dependent and domain-specific nonfunctional properties. Therefore, instead of trying to enumerate all possible metrics, we develop a QoS observation metamodel which can be used to construct various QoS observation models. The observation models in turn define the generic or domain-specific QoS metrics.

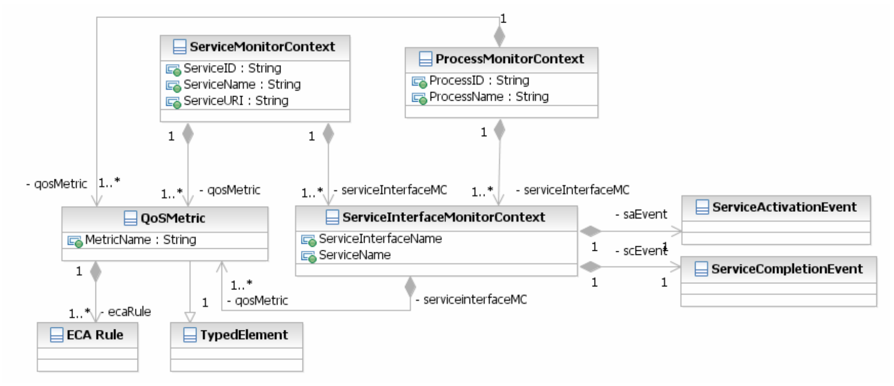


Fig. 1. Simplified Class Diagram of the Observation Metamodel

As indicated by the metamodel in Figure 1, an observation model can include three types of monitor contexts. Each type of monitor context corresponds to a type of entity to be monitored. A *ProcessMonitorContext* corresponds to a business process and specifies how a composite service should be observed. A *ServiceMonitorContext* (resp. *ServiceInterfaceMonitorContext*) corresponds to a service (resp. service interface). These two kinds of monitor contexts specify how component services should be observed. Users can define a collection of QoS metrics in a monitor context. A QoS metric can be of either a primitive type or a structure type, and can assume a single value or multiple values. For the computation logic, we adopt Event–Condition-Action (ECA) rules (c.f. Expression 1) to describe when and how the metric values are computed. Such a rule-based programming model frees users from the low-level details of procedural logic.

$$\text{Event(eventPattern)[condition]|expression} \quad (1)$$

In an ECA rule, the event pattern component indicates either a service operational event or the value change of a metric value. For example, when a service instance starts execution, a service activation event can be detected. The condition component in a rule is a Boolean expression specifying the circumstance to fire the computation action described in the expression component. The expression consists of an association predicate and a value assignment expression. The association predicate identifies which monitor context instance should receive the event. The operators allowed in the predicate expressions include relational operators, event operators, vector operators, set operators, scalar operators, Boolean operators and mathematical operators, etc. An example ECA rule for metric computation is given in equation (2).

$$Event(E_1 :: e)[e.a_2 > 12] \mid (MC_1.serviceID == e.serviceID)\ MC_1.m_2 := f_1(e) \tag{2}$$

In the above example, when an instance of event E_1, denoted as *e*, occurs, if $e.a_2 > 12$, then the event is delivery to the instance of MC_1 whose serviceID metric matches the serviceID field of event instance *e*, and the metric value of m_2 is computed by function $f_1(e)$. When there is no matching context instance, a new monitor context instance is created. It should be noted that the monitor context represents the entity that is being monitored, which is a service instance in this case. Another example ECA rule is given in equation (3). In the example, when the value of metric $MC_1.m_2$ changes, the value of metric $MC_1.m_3$ is updated by function $f_2(MC_1.m_1, MC_1.m_2)$.

$$Event(changeValue(MC_1.m_2)[] \mid MC_1.m_3 := f_2(MC_1.m_1, MC_1.m_2) \tag{3}$$

3 Monitoring-Enabled SOA Infrastructure

Figure 1 illustrates the proposed monitoring-enabled SOA infrastructure. Basing on the generic SOA infrastructure, three specific components that enable QoS monitoring are introduced. The *Web Service Observation Manager* provides interfaces that allow users to create observation models. The *Metric Computation Engine* generates executable code, detects service operational events and computes and saves metric values. The *QoS Data Service* provides an interface that allows other SOA components to access QoS information via a *Service Bus*. In this section, we mainly focus on the creation of observation models and the detection of service operational events. The details of metric computing and saving are presented in next section.

3.1 Observation Model Creation

We start with the observation model creation. When importing a process schema, the Web Service Observation Manager generates a ProcessMonitorContext first. For each service request in the process, it creates a ServiceInterfaceMonitorContext definition, in which two types of event definitions are also created, namely *execution activation event* and *execution completion event*. For example, if a service request is defined as *R* (*TaskName*, C_{in}, C_{out}), where C_{in} (C_{in}=<C_1, C_2,…, C_n >) indicates input types and C_{out} (C_{out}=<C_1, C_2,…, C_m >) indicates excepted output types, then the execution activation event can be defined as E_s(*PID, SID, TimeStamp, TaskName, ServiceName, ServiceInterfaceName,* <C_1, C_2,…, C_n>), where the *PID* is the process instance ID and the *SID* is the service ID. The execution completion event is defined as E_c(*PID, SID, TimeStamp, TaskName, ServiceName, ServiceInterfaceName,* <C_1,C_2,…,C_m>). Based on these service operational event definitions, the designers can further define the QoS metrics and their computation logic by creating ECA rules.

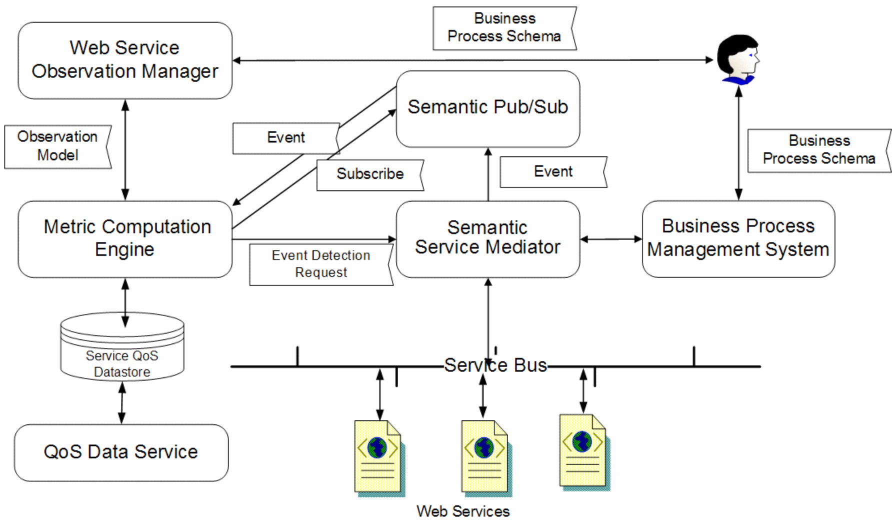


Fig. 2. Simplified QoS Monitoring-enabled SOA infrastructure

3.2 Detection and Routing of Service Operational Events

Given that the observation model is an event-driven programming model, there are two main steps before processing the events to compute the QoS metric values: event detection and event routing. If we assume that the data types are standardized across different process schemas and service interfaces, these two steps can be performed based on the syntactic information on service interfaces and service operational events.

However, such an assumption is impractical. Since services are operated in heterogeneous and dynamic environments, it is inappropriate to assume that all the service providers adopt the same vocabulary to define service interfaces. To improve the flexibility of SOA solutions, we have introduced semantics in service mediation [3], wherein service interfaces can be semantically matched with service requests. Therefore, when there are not any syntactically matched service interfaces for a service request, semantic match is applied to identify service interfaces. In cases of semantic matches, the data format transformations are required when invoking the matched service and returning the execution results to service consumers. In such cases, semantic matching is also required between the event definitions in observation models and the actual operational events detected. Fortunately we can leverage the same semantic-mapping capability provided by semantic service mediation to transforms operational events into formats that conform to the event definitions in the observation model.

If we assume that a service request is defined as $R(TaskName, C^r_{in}, C^r_{out})$ and $C^r_{out}=<C_1,C_2,…,C_m>$, the generated service activation event definition in the observation model is then $E_c(PID, SID, TimeStamp, TaskName, ServiceName, ServiceInterfaceName,<C_1,C_2,…,C_m>)$. We also assume that the matched service interface is defined as i $(serviceInterfaceName, C^i_{in}, C^i_{out})$, and that the execution output is $<o_1,o_2,\ldots,o_l>$. If $<o_1,o_2,\ldots,o_l>$ does not exactly match $<C_1,C_2,\ldots,C_m>$, but is *semantically compatible* (see Definition 1),, a semantic transformation that converts $<o_1,o_2,\ldots,o_l>$ to $<o'_1,o'_2,\ldots,o'_m>$ is needed. Similarly, if the detected service completion event $e_c(pID, sID, timeStamp, taskName, serviceName, <o_1,o_2,\ldots,o_l>)$ dose not exactly match the event definition E_c, same semantic transformation from $<o_1,o_2,\ldots,o_l>$ to $<o'_1,o'_2,\ldots,o'_m>$ is also required before the service completion event is emitted.

Definition 1. (Semantic Compatibility) $<o_1,o_2,\ldots,o_l>$ is semantically compatible with $<C_1,C_2,\ldots,C_m>$, if for each C_i, there is a o_j that is either an instance of C_i or an instance of C_i's descendant class.

In our design, the Metric Computation Engine takes observation models as input and generates event detection requests to the Semantic Service Mediator. The Semantic Service Mediator maintains a repository of service event detection requests (not shown in the Figure 1). Whenever a service execution is activated or completed, it searches the repository to determinate whether a service activation (or completion) event needs to be emitted. The search is done by semantically matching the service input and output with entries in the event detection request repository.

Similarly, it is impractical to assume that different process schemas use standardized data types and service interfaces. Therefore, when the event definitions in observation models are derived from service requests, it is necessary to consolidate those semantically matched monitored events. For example, consider two service requests $R^1(TaskName^1, C^1_{in}, C^1_{out})$ and $R^2(TaskName^2, C^2_{in}, C^2_{out})$ in two process schemas PS^1 and PS^2. Two execution activation event definitions can be generated as E_s^1 $(PID, SID, TimeStamp, TaskName, ServiceName\ ServiceInterfaceName, <C_1, C_2,\ldots,C_n>)$ and E_s^2 $(PID, SID, TimeStamp, TaskName, ServiceName, ServiceInterfaceName, <C_1, C_2,\ldots,C_m>)$ in two observation models OM^1 and OM^2 respectively. If $<C_1, C_2,\ldots,C_n>$ is semantically matched with $<C_1, C_2,\ldots,C_m>$, then the service operational events detected when executing PS^1 (resp. PS^2) should also be transformed and delivered to context instances in OM^2 (resp. OM^1). These transformations are performed by a semantic pub/sub engine [4]. Specifically, the Metric Computation Engine takes observation models as input and generates event subscriptions for the semantic pub/sub engine, relying on the latter to perform event transformation and event routing. For example, given OM^1, the Metric Computation Engine subscribes to event E_s^1 $(PID, SID, TimeStamp, TaskName, ServiceName, ServiceInterfaceName, <C_1, C_2,\ldots,C_n>)$. When an event e_s^2 $(pID, sID, timeStamp, taskName, serviceName, serviceInterfaceName,<o_1,o_2,\ldots,o_m>)$ (an instance of E_s^2) is published from the service mediator, the event is transformed to $e_s^1(pID, sID, timeStamp, taskName, serviceName, serviceInterfaceName, <o'_1,o'_2,\ldots,o'_n>)$ by semantic pub/sub and delivered to the appropriate context instances of OM^1.

4 High Performance Metric Computation Engine

Given a monitoring-enabled infrastructure to detect and route service operational events, it is imperative that these events be processed efficiently, and that the QoS metric values be computed and saved efficiently as well. The main challenges of the system design are tri-fold: high volume of service operational events, complexity of expressions involved in metric computation and persistence of metric values. Although most complex event processing systems [12][13][14][15] support high throughput of events, they primarily focus on event filtering and compound event detection. They do not address metric computation, where event data triggers and contributes to a complex flow of computation. Further, they don't consider the issue of state persistence. In this paper, we advocate a series of model analysis techniques to improve event throughput in a monitoring environment. In this paper, we only sketch out the high-level design but omit more detailed descriptions, due to the limitation of space. More complete descriptions of these techniques can be found in [11].

4.1 Model Transformation and Execution Framework

As we discussed earlier, event-driven rule-based programming is user friendly, particularly for business integration developers. However, because of the overhead in locating rules to be executed at runtime, the event-driven model does not lend itself to efficient execution, especially when the number of rules is very large, such as in the case of service QoS monitoring. In our design, the rule-based model is transformed to a state-based model, wherein statecharts are adopted to reorganize the rules. The rationale for such a model transformation is that statecharts organize the rules by states, which can greatly reduce the overhead in locating rules at runtime.

The construction of statecharts is based on user-defined ECA rules: a state represents either an event or a metric, while a transition between two states represents the triggering relationship (see Table 1). For example, if the event pattern is a service operational event in an ECA rule (see expression 2 for an example), then there is a transition from the event state to the metric state. In another case, the event pattern is the value change of a metric (see expression 3 as an example), and the corresponding transition is from one metric state to another metric state.

Table 1. Transforming the ECA rules to Statecharts

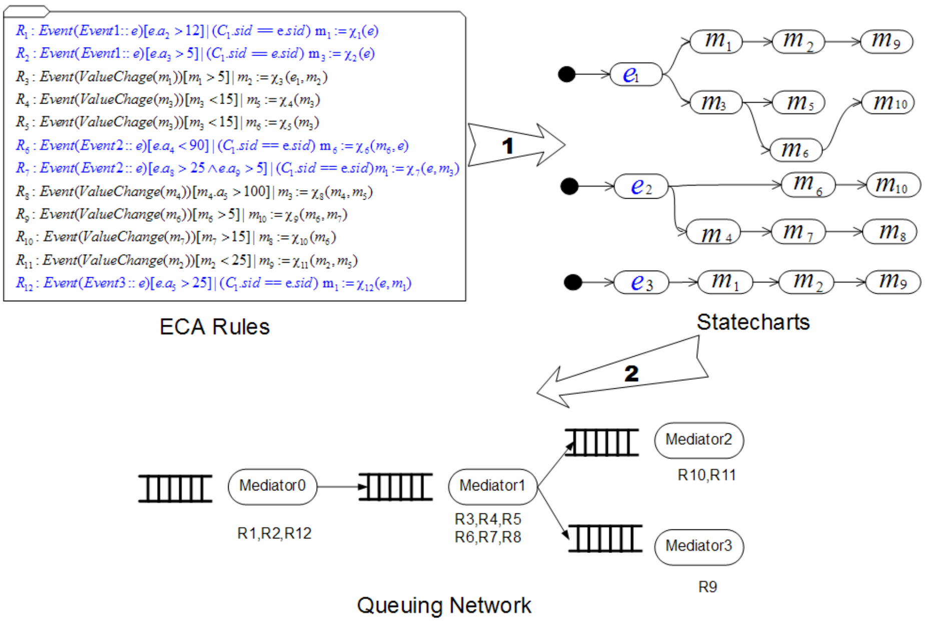
ECA rule type	Control-flow Transitions in Statechart
Event pattern is a service operational event	$\begin{tikzcd} \boxed{\text{Event}} \arrow[r] & \boxed{\text{Metric}} \end{tikzcd}$
Event pattern is value change of a metric	$\begin{tikzcd} \boxed{\text{Metric1}} \arrow[r] & \boxed{\text{Metric2}} \end{tikzcd}$

With the above transformation, each service operational event initiates a statechart. Thus, the execution of the ECA rules is converted to the execution of statecharts. An example of transformation is shown in Figure 3. In the example, three statecharts are generated from twelve ECA rules. The advantage of executing statecharts is that the overhead of a full rule set scan is eliminated when identifying the rules to be executed at runtime, as the next rules that need to be executed can be located via the outgoing transitions of the current state.

There are two approaches to executing statecharts, compilation and interpretation. Both approaches have their own advantages and drawbacks. We discuss the interpreting approach first. In order to execute the statecharts, the interpreter not only interprets the state transition logic, but also interprets the expressions in the rules. Given that the operators that appear in expressions can be relational, set, vector, scalar, and etc., interpretation is less efficient than compilation [9]. With a compilation approach, executable code is generated from a statechart. As custom code is generated at buildtime for the execution of statecharts, this reduces CPU cycles at runtime. However, the compilation approach entails another potential performance issue. When using multi-threads to process events, thread scheduling relies on the lock-based scheduling mechanism provided by either the operating system or language runtime (e.g., JVM). Such lock-based scheduling usually results in high system overhead [10], especially in multiple CPUs systems.

We take advantages of both approaches and propose a hybrid approach. In the hybrid approach, state transition logic is interpreted, while the expression in a rule is compiled into standalone executable code. The advantages of such a hybrid approach are twofold. On the one hand, by interpreting the state transition logic, the computation engine can plan the execution of rules in finer granularity, i.e., at the transition level instead of the statechart level. For example, information about the dataflow among the rules can be used to plan the wait-free execution of the expressions (details can be found in next subsection). On the other hand, the execution of an individual expression is done by executing pre-complied code, which enjoys the efficiency of the compilation approach.

Adopting the hybrid approach, we further develop a queuing network to execute the statecharts, in order to enable dynamic CPU allocation at runtime. At deployment time, the ECA rules in different statecharts are distributed to a collection of mediators.



$R_1 : Event(Event1 :: e)[e.a_2 > 12] \mid (C_1.sid == e.sid)\ \mathrm{m}_1 := \chi_1(e)$

$R_2 : Event(Event1 :: e)[e.a_3 > 5] \mid (C_1.sid == e.sid)\ \mathrm{m}_3 := \chi_2(e)$

$R_3 : Event(ValueChage(m_1))[m_1 > 5] \mid m_2 := \chi_3(e_1, m_2)$

$R_4 : Event(ValueChage(m_3))[m_3 < 15] \mid m_5 := \chi_4(m_3)$

$R_5 : Event(ValueChage(m_3))[m_3 < 15] \mid m_6 := \chi_5(m_3)$

$R_6 : Event(Event2 :: e)[e.a_4 < 90] \mid (C_1.sid == e.sid)\ \mathrm{m}_6 := \chi_6(m_6, e)$

$R_7 : Event(Event2 :: e)[e.a_8 > 25 \wedge e.a_9 > 5] \mid (C_1.sid == e.sid) m_1 := \chi_7(e, m_3)$

$R_8 : Event(ValueChange(m_4))[m_4.a_5 > 100] \mid m_3 := \chi_8(m_4, m_5)$

$R_9 : Event(ValueChange(m_6))[m_6 > 5] \mid m_{10} := \chi_9(m_6, m_7)$

$R_{10} : Event(ValueChange(m_7))[m_7 > 15] \mid m_8 := \chi_{10}(m_6)$

$R_{11} : Event(ValueChange(m_2))[m_2 < 25] \mid m_9 := \chi_{11}(m_2, m_5)$

$R_{12} : Event(Event3 :: e)[e.a_5 > 25] \mid (C_1.sid == e.sid)\ \mathrm{m}_1 := \chi_{12}(e, m_1)$

Fig. 3. Execution Model Transformation

Each mediator in the network possesses a work item queue, an interpreter and a thread pool. The queue buffers available work items. The interpreter executes the complied code of expressions in the rules. The thread pool enables multi-thread concurrent processing on work items, wherein the number of thread can be configured dynamically. The threads in different thread pools have the same level of priority. The CPU resource allocation for a mediator is determined by the size of its thread pool. By configuring the size of the thread pool dynamically, CPU resource can be dynamically allocated.

The collection of mediators forms a queuing network, wherein the number of mediators and the topology of the network are determined by the topologies of statecharts. The strategies for constructing the queuing network are: (i) The order of rule execution is preserved by the network topology. This is achieved by first sorting the rules based on the execution sequence in each statechart and then distributing the rules to an ordered collection of mediators based on the rules' execution order. (ii) The communication cost among mediators is minimized by eliminating data access contention among the threads in different mediators. This can be done by distributing rules that access the same data into the same mediator. An example of queuing network is shown in Figure 3.

4.2 Execution Planning

One of the key techniques for improve event processing throughput is multi-threaded concurrent processing. However, event throughput normally is not proportionate with the number of concurrent threads deployed, because of the runtime overhead incurred by the concurrency control mechanism. QoS monitoring requires that QoS metric values be persistent and we use a relation database for this purpose. In order to reduce the amount of I/O between the Metric Computation Engine and the datastore, a cache is also instituted. Therefore, either the datastore or the cache needs to provide concurrency control. Although modern RDBMs support row-level locking, such an option substantially deteriorates database performance. On the other hand, if concurrency control is implemented in the cache, a rollback segment needs to be maintained for each transaction. Given the large volume of events and that each event occurrence initiates a transaction, a large number of rollback segments need be managed by the cache. These rollback segments occupy significant memory and eventually impair performance. Therefore, an approach of supporting concurrent threads without locking, such as a lock-free approach, is more appealing [16][17]. However, these lock-free approaches rely on either the hardware or programming language support on for compare-and-swap [3]. Aiming at a solution that is independent of hardware or programming languages, we plan the execution ahead of time using information in the observation model The basic idea is that we plan the rule execution in each mediator: if the execution of two rules update the same metric or one rule produces operands for another rule, then these two rules need to be executed sequentially; otherwise these two rules can be executed concurrently. It should be noted that the execution order relationships between the rules are derived before the runtime. Therefore, there is not much runtime overhead involved when planning the executions.

5 Implementation and Experimentation

Our implementation leverages the Websphere Process Server (WPS) [24]. WPS is a SOA solution platform that contains a BPEL engine and provides a service bus for Web services. The proposed Metric Computation Engine uses a message driven bean to receive service operational events routed from the semantic pub/sub engine. *We have also developed a dashboard to display the metric values from the QoS Data Service. W*e have conducted a series of experiments to demonstrate the functionality of Web service QoS monitoring. We first created a business process called *"patient visit" (see Figure 4) and deployed it on WPS. From the service request definitions in the process, a skeleton observation model was generated by the Web Service Observation Manager that consisted of one process monitor context, six service interface monitor contexts and twelve service operational event definitions. Given the skeleton model, we then created about forty metric definitions and ECA rules. We deployed the complete model into the Metric Computation Engine, wherein the model information was transformed and executable Java classes were generated. These generated Java classes were distributed to five mediators. When the process "patient visit" is executed, the related service operational events are detected and published to the Semantic Pub/Sub engine. When these events are routing to the Metric Computation Engine, the metric values are computed and saved. Eventually, the computed metric values are displayed on the dashboard in realtime fashion.*

To test the system throughput, we designed an event emitter that sends simulated service operational events to the Metric Computation Engine with a given sending rate (i.e., number of events per second). On an Intel CPU Linux server, the Metric Computation Engine can process 660 events/sec. In order to test its salability, we deployed the Metric Computation Engine on multiple CPU hardware platforms (2 and 4 CPUs). The experiment results (1210 events/sec and 2012 events/sec respectively) demonstrate that our system is scaled to the number of CPUs.

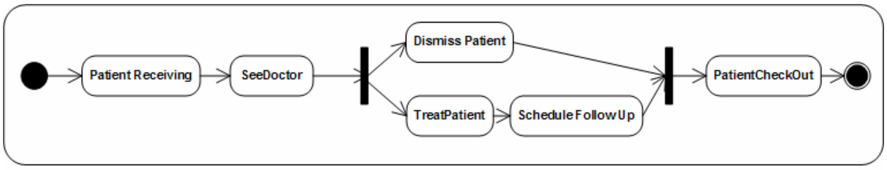


Fig. 4. An Example of Business Process

6 Related Work

In this section, we review work in the areas of QoS management and event processing systems. QoS management has been widely studiesd in the context of middleware systems [18][19][20]. These efforts have addressed the following issues: QoS specification to allow description of application behavior and QoS parameters, QoS translation and compilation to translate specified application behavior into candidate application configurations for different resource conditions, QoS setup to appropriately select and instantiate a particular configuration, and QoS adaptation to runtime resource fluctuations. Most efforts in QoS-aware middleware, however, are focused on the network transport and system level. Little work has been done at the application and business process levels.

QoS-Aware service composition [1][2][4][5][6][7][8][19][20] aims for selecting component services to optimize the overall QoS of a business process. In [2][7], the system assumes that the QoS information of components is pre-existing, and therefore, the overall QoS of composite service can be computed based on formulas. In [8], the formulas that compute the QoS of a workflow based on both the QoS of component services and the workflow schemas are discussed. However, it only focuses on the QoS at IT level. In [5], a QoS-aggregation system is presented. It provides an editor for the QoS aggregation function that allows users to specify QoS attributes and their aggregation formulas. It also provides an interpreter that evaluates a workflow's global QoS. Again, it assumes that the QoS information of component services is pre-existing. Further, it does not provide the details on how to compute the aggregation formulas efficiently. Different from above works, this paper tries to tackle the fundamental issue: monitor and compute the QoS of component/composite services, both at IT and business level. Further, it discusses the design and implementation of a high performance metric computation engine.

Complex event processing systems [12][13][14] focus on event filtering and compound event detections However, in service QoS monitoring, event filtering logic is relatively simple. Complicated computation happens after the events are filtered, i.e., when the event data is used to compute/update a collection of metrics. rFurther, most of the complex event processing systems do not support state persistence, even though it is a critical requirement for a service QoS monitoring system to save metric values.

7 Conclusion

In this paper, we advocate computing the QoS metrics of services by monitoring the executions. An observation model is proposed, which allows users to define the metric types and formulas. We design a monitoring-enabled SOA infrastructure to enable the systematic detection and routing of service operational events. Further, we implement a high performance metric computation engine that can support high event throughput. Our further work includes supporting the metric network (i.e., probabilistic, system dynamics and extensible user-defined dependency) and a careful study of the system.

References

1. Menasce, D.A.: QoS Issues in Web Services. IEEE Internet Computing 6(6) (2002)
2. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality Driven Web Services Composition. In: WWW 2003 (2003)
3. Prakash, S., Lee, Y.H., Johnson, T.: A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. IEEE Transactions on Computers 43(5) (1994)

4. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An Approach for QoS-aware Service Composition based on Genetic Algorithms. In: GECCO 2005, ACM Press, New York (2005)
5. Canfora, G., Di Penta, M., Esposito, R., Perfetto, F., Villani, M.L.: Service Composition (re)Binding Driven by Application-Specific QoS. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, Springer, Heidelberg (2006)
6. Nguyen, X.T., Kowalczyk, R., Han, J.: Using Dynamic asynchronous aggregate search for quality guarantees of multiple Web services compositions. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, Springer, Heidelberg (2006)
7. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineering 30(5) (2004)
8. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.J.: Modeling quality of service for workflows and web service processes. Web Semantics Journal: Science, Services and Agents on the World Wide Web Journal 1(3), 281–308 (2004)
9. Rao, J., Pirahesh, H., Mohan, C., Lohman, G.M.: Compiled query execution engine using jvm. In: ICDE 2006 (2006)
10. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley Professional, Reading (2006)
11. Zeng, L., Lei, H., Chang, H.: Model-analysis for Business Event Processing. IBM Systems journal (2007) (to appear)
12. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: SIGMOD 2006 (2006)
13. Wang, F., Liu, P.: Temporal management of RFID data. In: VLDB 2005 (2005)
14. Complex Event Processing, http://en.wikipedia.org/wiki/Complex_event_processing
15. Luckham, D.: Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, 1st edn. Addison-Wesley Professional, Reading (2002)
16. Ennals, R.: Efficient Software Transactional Memory, Intel Research Cambridge Technical Report: IRC-TR-05-051 (2005)
17. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free Synchronization: Double-ended Queues as an Example. In: ICDCS (2003)
18. Zeng, L., Lei, H.: A Semantic Publish/Subscribe System. In: IEEE CEC (East) (2004)
19. Gillmann, M., Weikum, G., Wonner, W.: Workflow Management with Service Quality Guarantees. In: SIGMOD 2002 (2002)
20. Nahrstedt, K., Xu, D., Wichadakul, D., Li, B.: QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. IEEE Comm. Magazine 39(11) (2001)
21. Zeng, L., Benatallah, B., Xie, G.T., Lei, H.: Semantic Service Mediation. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, Springer, Heidelberg (2006)
22. Zeng, L., Lei, H., Jeng, J.-J., Chung, J.-Y., Benatallah, B.: Policy-Driven Exception-Management for Composite Web Services. In: IEEE CEC 2005 (2005)
23. Zeng, L., Jeng, J.-J., Kumaran, S., Kalagnanam, J.: Reliable Execution Planning and Exception Handling for Business Process. In: Benatallah, B., Shan, M.-C. (eds.) TES 2003. LNCS, vol. 2819, Springer, Heidelberg (2003)
24. Websphere Process Server, http://www-306.ibm.com/software/integration/wps/