Event Driven Monitoring for Service Composition Infrastructures

Oliver Moser[1], Florian Rosenberg[2], and Schahram Dustdar[1]

[1] Distributed Systems Group, Vienna University of Technology, Austria
[2] CSIRO ICT Centre, GPO Box 664, Canberra ACT 2601

Abstract. We present an event-based monitoring approach for service composition infrastructures. While existing approaches mostly monitor these infrastructures in isolation, we provide a holistic monitoring approach by leveraging Complex Event Processing (CEP) techniques. The goal is to avoid fragmentation of monitoring data across different subsystems in large enterprise environments by connecting various event producers. They provide monitoring data that might be relevant for composite service monitoring. Event queries over monitoring data allow to correlate different monitoring data to achieve more expressiveness. The proposed system has been implemented for a WS-BPEL composition infrastructure and the evaluation demonstrates the low overhead and feasibility of the system.

Keywords: Monitoring, Composition, Complex Event Processing.

1 Introduction

Service-Oriented Computing (SOC) methodologies and approaches [18] are becoming an established and predominant way to design and implement process-driven information systems [23]. Service composition approaches are often used to orchestrate various loosely coupled services to implement business processes within and across enterprise boundaries. In such loosely-coupled environments, it is crucial to monitor the execution of the overall system and the business processes in particular. This ensures that problems, instabilities or bottlenecks during system execution can be detected before, for example, customers become aware of the problem.

In many applications, the composition engine acts as a central coordinator by orchestrating numerous services that implement the business processes. Therefore, the composition infrastructure needs to provide effective mechanisms to monitor the executed business processes at runtime. These include technical aspects such as monitoring Quality of Service (QoS) attributes of the services [16,15], resource utilization as well as business-related information relevant for Key Performance Indicator (KPI) progression or Service Level Agreement (SLA) fulfillment [21].

A number of monitoring systems have been proposed, however they are mainly tailored for a particular composition or workflow runtime [22,19,12,3,6,5,13]. Thus, these systems are usually tightly coupled or directly integrated in the composition engine without considering other subsystems that live outside the composition engine but still influence the composite service execution (such as message queues, databases or Web services). The lack of an integrated monitoring mechanism and system leads to a fragmentation of monitoring information across different subsystems. It does not provide a holistic view of all monitoring information that can be leveraged to get detailed information of the operational system at any point in time. Additionally, most monitoring approaches do not allow a so-called multi-process monitoring, where monitoring information from multiple composite service instances can be correlated. This calls for an integrated and flexible monitoring system for service composition environments. We summarize the main requirements below, and give concrete and real world examples in Section 2 when describing the case study.

– **Platform agnostic and unobtrusive**: The monitoring system should be *platform agnostic*, i.e., independent of any concrete composition technology and runtime. Additionally, it should be *unobtrusive* to the systems being monitored, i.e., there should be no modifications necessary to interact with the monitoring runtime.
– **Integration with other systems**: The monitoring system should be capable of integrating monitoring data from other subsystems outside the composition engine (such as databases, message queues or other applications). This enables a holistic view of all monitoring data in a system.
– **Multi-process monitoring**: Composite services often cannot be monitored in isolation because they influence each other (cf. Section 2). The monitoring system should enable monitoring across multiple composite services, and its instances. Correlating monitoring data across several independent service compositions can be achieved by matching key information of the payload of the underlying message exchange.
– **Detecting anomalies**: The monitoring system should be capable of unveiling potential anomalies in the system, e.g., problems in backend services due to a high load outside of usual peak hour times. Additionally, the system should support dependencies between monitoring events. This feature can be used to find fraudulent and malicious behavior based on certain event patterns, e.g., absent events in a series of expected events.

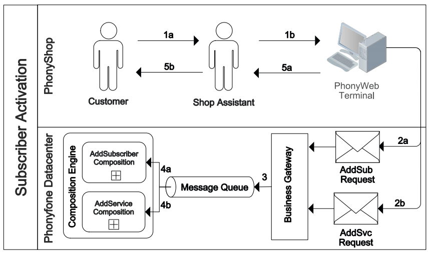
We propose an event-driven monitoring system to address these requirements. Our approach leverages Complex Event Processing (CEP) to build a flexible monitoring system [9] that supports temporal and causal dependencies between messages. The system provides a loosely coupled integration at the messaging layer of a composition engine (e.g., on the SOAP message layer in the context of WS-BPEL). A monitoring component intercepts the messages sent and received by the composition engine and emits various events to the monitoring runtime. Furthermore, the monitoring runtime allows to connect arbitrary subsystems by using event source adapters. These external events are also processed within the monitoring runtime to provide an integrated monitoring environment.

This paper is organized as follows. Section 2 presents an illustrative example from the telecommunication domain. Section 3 introduces the concepts of our event driven monitoring approach and presents a proof of concept implementation. A detailed evaluation of our system is presented in Section 4. Section 5 gives an overview of the related work with respect to the main contributions of this paper. Finally, Section 6 concludes this paper.

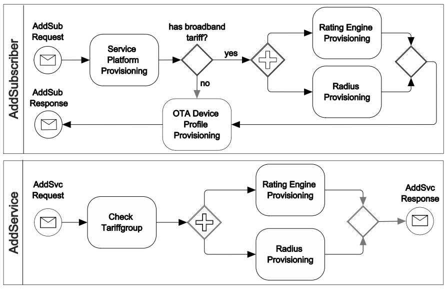
2 Illustrative Example

This section introduces a subscriber activation scenario in a telecommunications enterprise called *Phonyfone*. A subscriber in our example is an active Phonyfone customer. Figure 1a shows a high level view of the activation process. A *Customer* provides required information such as personal and payment data to a *Shop Assistant* (1a). The Shop Assistant creates a new contract by entering the related data into the *PhonyWeb Terminal* (1b). The PhonyWeb Terminal transmits the customer (2a) and service data (2b) to an on-premises *Business Gateway*. The Business Gateway dispatches the two requests into a *Message Queue* (3), which delivers the requests (4a and 4b) to Phonyfone's *Composition Engine*. The Composition Engine triggers the execution of both, the *AddSubscriber* (AddSub) and *AddService* (AddSvc) composite services (Figure 1b). They enable Phonyfone to automate the steps required to create a *subscriber* object for a new customer as well as a *service* object for an additional service, such as mobile Internet access. Upon normal completion of these service compositions, the subscriber becomes active, and the PhonyWeb Terminal provides feedback to the Shop Assistant (5a). Finally, the Shop Assistant can handover a printout of the contract to the customer (5b).

From an operational perspective, Phonyfone adopts composition engines from two different vendors. This situation stems from a recent merger between Phonyfone and its parent company. Phonyfone faces between 500 and 600 subscriber activations per day, with 90 percent of the related composite service invocations executing between 09:00 am and 05:00 pm. On average, the AddSub request requires 4000ms for processing, and the AddSvc request requires 2500ms. The



(a) Subscriber Activation Workflow



(b) Phonyfone Service Compositions

Fig. 1. Phonyfone Enterprise Scenario

combined processing time should not exceed 10 seconds. Deviations in both the throughput as well as the execution time can indicate potential problems, either in the backend systems or the network backbone itself. On the other hand, a rise in subscriber activations can prove customer acceptance for a promotion or special sale. Detecting such anomalies through an appropriate monitoring system *before* customers report them is of uttermost importance, both in terms of ensuring normal operation and deriving customer satisfaction with new or improved products.

2.1 The AddSubscriber and AddService Composition

Both the AddSub and AddSvc composite services are initiated by incoming XML requests (not shown for brevity). The following paragraphs discuss these services and their corresponding monitoring needs in more detail.

AddSubscriber. The AddSub composition (Figure 1b, top) is created by receiving an AddSub request. This request contains the Msisdn (i.e., the phone number of the new customer), the tariff information, the originator (i.e., shop assistant id) and an optional reduced recurring charge (RRC) field with a given start and end date. Provisioning of core subsystems starts with persisting relevant data in Phonyfone's *Service Delivery Platform* (SDP). The SDP provides access to subscriber and service relevant data. Then, if the tariff information provided in the request indicates a broadband product (for mobile Internet access), subscriber data is provisioned to two additional systems (*RADIUS* and the *Rating Engine* for dial-in services). Next, the *Over The Air* (OTA) device profile provisioning stores cellphone model and brand information in a device configuration server. Finally, an AddSub response, indicating a success or failure message, is sent back to the requester.

A potential risk emerges when reasoning about the RRC feature. Phonyfone allows its shop assistants to grant five RRCs per day, however, this limit is not enforced by any system and depends solely on the loyalty of the shop assistant. In the worst case scenario, this loophole can be exploited by a dishonest shop assistant. Monitoring of message exchange ensures compliance with enterprise policies and can be used to detect fraudulent behavior.

AddService. The AddSvc composition (Figure 1b, bottom) starts upon receiving a AddSvc request. It contains a subset of the request from the AddSub composition plus the name of the requested service (e.g., AntiVirus). After checking the subscriber's tariff to ensure that the subscriber is allowed to order the service, the Rating Engine as well as the RADIUS service is provided with service information. Finally, an AddSvc response is returned to the requester, indicating success or failure.

It is important to note that an AddSvc request will fail if the related AddSub request has not been successfully processed, i.e., the subscriber does not (yet) exist in Phonyfone's subscriber database. On the other hand, the AddSub request alone does not complete the subscriber activation – both the AddSub and the AddSvc request have to be processed timely and in the right order. This implies that a monitoring solution must be capable of (1) detecting situations where the temporal ordering between requests is wrong and (2) finding out which client issued the requests, e.g., to fix it. Finally, both service compositions are deployed on composition engines from different vendors, which hampers correlation of monitoring data due to non-uniform logging facilities. A holistic view on monitoring data from the various runtime environments greatly improves manageability and enforcement of monitoring requirements.

3 Event Driven Monitoring

Considering the monitoring needs of Phonyfone, it is rather obvious that a log file or basic performance reports cannot cover the requirements discussed in Section 1 and 2. A comprehensive monitoring solution that is capable of reflecting causal and temporal context within the composition engine's message exchange is needed. Moreover, the necessary platform independence requires that the monitoring solution has to operate on a layer that is common to most composition platforms. Operating on the *message level* makes the system we propose agnostic to implementation details of the monitored platform. Hence, it supports a large variety of current and future composition platforms. The temporal and causal requirements are covered by interpreting the message exchange as a stream of *events*. Each incoming and outgoing message is associated with an event object that has certain properties (cf. Section 3.1). Clearly, some messages are dependent on other messages, both from a causal as well as temporal point of view. These dependencies can be modeled by combining single message events into more complex composite events. This concept is generally known as *Complex Event Processing* (CEP) [14] and represents the fundamental building block of the *event-driven* monitoring system we will discuss below.

3.1 Event Model

As aforementioned, the monitoring approach we propose is built around the concept of events. From various available definitions [7], we follow a rather technical view of an event as a detectable condition that can trigger a notification. In this context, a notification is an event-triggered signal sent to a runtime-defined recipient [11]. In the Phonyfone scenario, each monitoring requirement breaks down to a detectable condition that can be represented by events of various types. An *event type* represents the formal description of a particular kind of event. It describes the data and structure associated with an event. As a simple example, detecting the anomalies described in Section 2 can be covered by an event type `AddSubCompositionInvocation`. This event type has several data items such as a timestamp or the execution duration. Using these data items, an *event processor* can filter the vast amount of events and separates events of interest, e.g., events of type `AddSubCompositionInvocation` where the timestamp is between 09:00 and 05:00pm and the execution time exceeds 4000ms.

To reason about and to provide a generic way to deal with the various event types, an *event model* is needed. Such an event model organizes the different event types and generalizes them by introducing an event hierarchy. Thus, it is possible to leverage simple base events to define more specific and complex events. Figure 2 displays a simplified version of our proposed event model. Various other types of events, such as service or composition life cycle events, are out of scope of this work.

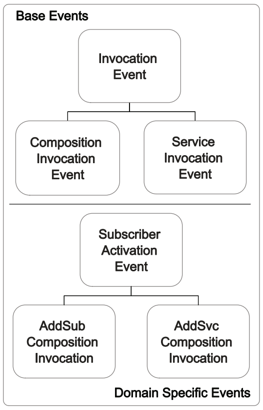

Base Events
Invocation Event
Composition Invocation Event
Service Invocation Event
Subscriber Activation Event
AddSub Composition Invocation
AddSvc Composition Invocation
Domain Specific Events

Fig. 2. Our event model distinguishes between *Base Events* and *Domain Specific Events*. Base Events are triggered upon invocation of either a service composition or a single service. Both `CompositionInvocationEvent` and `Service-InvocationEvent` inherit common attributes, such as a timestamp, the execution duration and a success indicator, from an abstract `InvocationEvent`. Domain Specific Events are user-defined and built upon Base Events. They are designed for a particular monitoring use case, e.g., our subscriber activation scenario. Additionally, they have access to the related message context to analyze the payload of the underlying message exchange. This feature enables the correlation of events based upon message payload, such as the `<Msisdn>` of the XML requests mentioned in Section 2.

3.2 Event Processing Language

In the previous section, we argued that an event processor is required to select specific event objects out of a stream of events, based on certain properties. Similar to SQL of relational databases, most event processors support an *Event Processing Language* (EPL). However, and contrary to relational databases, the processing model of event stream processors is *continuous* rather than only when queries are submitted. In particular, user defined queries are stored in memory and event data is pushed through the queries. This approach provides better performance and scalability than disk based database systems for large amounts of data, which makes it ideal for our application domain.

From our illustrative example, we can identify two requirements for an EPL. First, the EPL must provide capabilities to model complex *patterns* of events. Considering the coupling between invocations of the AddSub and AddSvc composite services, this sequence can be modeled as the following event pattern: `AddSub` $\rightarrow$ `AddSvc(msisdn = AddSub.msisdn)`. The arrow ($\rightarrow$) denotes that an AddSvc event follows an AddSub event. Moreover, we need to specify that the Msisdn included in both messages need to be the same. A *filter criteria* (msisdn = AddSub.msisdn) can be set to connect several events through a common attribute, such as the Msisdn. Besides detecting the presence of a particular event, the EPL should also support detecting the absence of an event, e.g., where the PhonyWeb Terminal issues an AddSub request but no related AddSvc request. Second, the EPL has to support event stream *queries* to cover online queries against the event stream, ideally using a syntax similar to existing query languages such as SQL. This feature can be used to analyze specific information about the business domain, e.g., the number of failed subscriber activations during a particular time window or even the related Msisdns: `SELECT msisdn FROM AddSub.win:time(15 minutes) WHERE success = false`. This query returns all Msisdns of AddSub requests that failed within the last 15 minutes. Furthermore, event stream queries can be leveraged to deal with marketing specific requirements such as finding the most wanted postpaid tariff of the last business day.

Reasoning about the fraudulent behavior of a shop assistant discussed in Section 2 leads us to yet another EPL requirement. Detecting an unusual high number of requests with a reduced recurring charge set can be handled by defining the following event pattern: `[5] (a = AddSub → b = AddSub(originator = a.originator and a.reducedrc = reducedrc and a.reducedrc > 0).win: time(1 day)`. The repeat operator ([n], $n > 0$) triggers when the statement following the operator evaluates to true n times. However, we need to know which shop assistant triggered the activation. Thus, the final requirement for our EPL is the combination of both event pattern matching and event stream queries: `SELECT a.originator FROM PATTERN ([5] (a = AddSub → b = AddSub(originator = a.originator and a.reducedrc = reducedrc and a.reducedrc > 0).win:time(1 day)`. This query will unveil the dishonest shop assistant.

3.3 Architectural Approach

Figure 3 depicts a high-level overview of the proposed system architecture for our monitoring runtime. As discussed earlier, our approach is agnostic to the *Composition Runtime* under observation. The Composition Runtime is comprised of a *Composition Processor* and a *Messaging Stack*. The Composition Processor interprets and executes the service composition. It uses the Messaging Stack to process incoming and outgoing messages. The former represent process

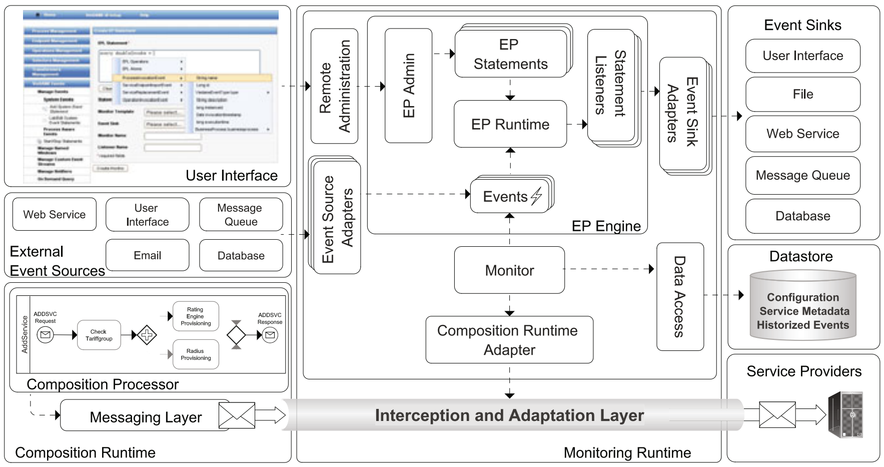


Fig. 3. Architectural Approach

instantiation messages (cf., BPEL receive elements) whereas the latter represent outgoing service invocations (cf., BPEL invoke elements).

The *Monitoring Runtime* introduces an *Interception and Adaption Layer* (IAL), which enables the inspection and, eventually, adaptation of message exchange [16]. The *Monitor* is the key component of our Monitoring Runtime. It serves two main purposes. First, it extracts service related metadata, such as endpoint URLs, and persists this information in a *Datastore* using the *Data Access* component. Second, it emits *Events*, such as `ServiceInvocationEvents` (cf. Section 3.1). The Monitor uses a *Composition Runtime Adapter* (CRA) to access the IAL. The CRA covers all implementation specific details of the monitored Composition Runtime. The Events emitted by the Monitor are handed over to the *Event Processing Runtime* (EP Runtime), which is part of the *Event Processing Engine* (EP Engine) incorporated in our system. The EP Runtime provides the capabilities discussed in Section 3.2. An *Event Processing Statement* (EP Statement) is registered with the EP Runtime and represents the choice of an event pattern, an event stream query or a combination of both. For each EP Statement, one or more *Statement Listeners* can be configured. Each Statement Listener is provided with a reference to the Event in case the EP Runtime produces a match for the underlying event pattern or query. To make use of this information, a Statement Listener has a list of registered *Event Sinks*. An Event Sink subscribes to the information represented by the Event objects, and will leverage this information to perform further analysis, aggregation or simply alert an operator. *Event Sink Adapters* encapsulate these details to abstract from the implementation and protocol specifics of the supported Event Sinks. Similarly, *Event Source Adapters* provide integration for *External Event Sources*, which can push external event data into our monitoring system. This allows to reflect data in EP statements that does not originate from the monitored service (composition), hence enabling a holistic view on monitoring data from various heterogeneous systems. Finally, a *User Interface* (UI) simplifies numerous administrative tasks. It provides a statement builder, auto completion features and syntax highlighting, and is decoupled from the Monitoring Runtime through the *Remote Administration Interface* to allow distributed deployment.

3.4 Implementation

This section briefly describes the Java based proof of concept implementation on top of our existing VieDAME framework. Figure 4 provides an overview of implementation technologies and highlights components related to event driven monitoring. For an in-depth discussion of VieDAME please refer to [16].

For our prototype implementation, WS-BPEL [17] is used as the underlying composition technology. *Engine Adapter*s provide support for particular *BPEL Runtime* providers, such as ActiveBPEL or Apache ODE. The interception of the SOAP messages that are created by `<invoke>` or `<receive>` BPEL activities is done by the AOP based *Message Interception Layer*. *JBoss AOP* is leveraged to weave this processing layer in between the BPEL Runtime and the *SOAP Stack*, e.g., Apache Axis. Using AOP as a technology to extend a base system, i.e., the BPEL Runtime, stems from the fact that it minimizes the coupling

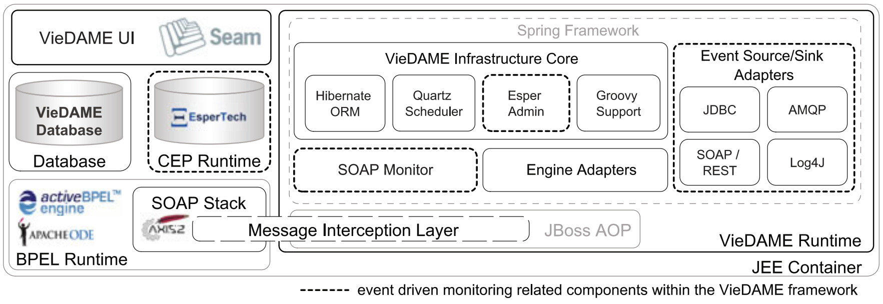


Fig. 4. Implementation Technologies

between the base system and our monitoring runtime. Thus, it effectively fulfills the requirements in terms of unobtrusiveness and platform independence from Section 1.

The *SOAP Monitor* taps the Message Interception Layer using an Engine Adapter suitable for the underlying BPEL runtime. It processes the message context and extracts service and process metadata, which is then stored in the *VieDAME Database*. Most important, it creates event objects that are passed to the *Esper CEP Runtime* [9] for event processing. Esper matches the processing language expressiveness and performance requirements postulated earlier and is easily embeddable into existing systems. Additionally, it provides XML DOM and StAX based support for XML payload processing of the message exchanges. Both processing modes use XPath expressions to access certain message elements, e.g., the `<Msisdn>` element from the AddSub and AddSvc messages from Section 2. For an in-depth discussion of the incorporated EPL please refer to the excellent Esper documentation [10].

The *Infrastructure Core* of our system provides common functionalities, such as Object/Relational Mapping (ORM), task scheduling or dynamic language support. To allow runtime integration and adaptation of event sinks such as a message queue, or external event sources, e.g., an SMTP host, Groovy is leveraged. It supports scripting and runtime deployment of templates that are used to create and modify the various supported *Event Sink Adapters* and *Event Source Adapters*.

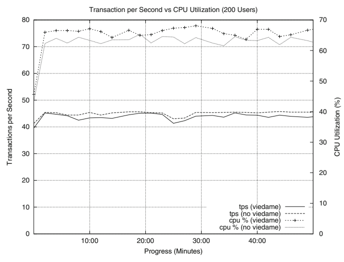
4 Evaluation

For the evaluation of our event-driven monitoring system, we used the VieDAME based prototype discussed in Section 3.4. Additionally, we created an ActiveBPEL [1] implementation of the subscriber activation scenario from our illustrative example. Due to space restrictions, only results for the AddSub process are shown below[1].

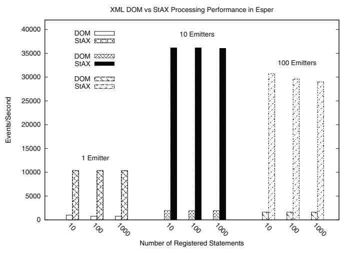
[1] Full results of the load tests can be obtained on the project website `http://viedame.omoser.com`

4.1 System Performance

We created a 50 minute load test using an industry grade tool (Mercury Loadrunner), measuring the performance of a default ActiveBPEL based (no viedame) process execution in comparison to a process execution where our monitoring system was enabled (viedame). The prototype was hosted on a quad core machine (3Ghz, 8GB RAM) running Linux. Implementations of the orchestrated Web services were created using JAX-WS and provide a randomized processing delay to mimic backend logic. Figure 5a shows the number of transactions per second (TPS) and the CPU utilization (CPU) for a 200 concurrent users scenario, with the viedame scenario having 100 EP statements registered in the system. The results show that performance in terms of TPS is almost identical, while the CPU utilization is only slightly higher in the viedame scenario. Both numbers confirm that the monitoring system we propose does not impose a real performance penalty on existing systems. Compared to our findings from previous work [16], we could effectively minimize the processing overhead by introducing a CEP engine for measuring QoS data and relying on a relational database only for historical data.



(a) Monitoring Overhead



(b) XML DOM vs StAX Performance

To drill down to the source of the processing overhead, we used a Java profiler (YourKit) to determine the component that uses most CPU cycles. The profiler showed clearly that the XML processing required by the domain specific events is rather time consuming. Besides the XML DOM processing mode, Esper contains a plugin that supports StAX (Streaming API for XML). We created another test scenario where we evaluated the event processing performance isolated from the BPEL engine to compare both processing modes. This scenario featured test runs for 1, 10 and 100 concurrent event emitters, i.e., Java threads concurrently issuing XML requests. The tests were executed with Esper having 10, 100 and 1000 statements registered. Figure 5b shows that the StAX based processing mode has far superior performance than the XML DOM based mode. Using the StAX mode, Esper is capable of processing more than 35000 events/second in the 10 emitter scenario, compared with 1900 events/second for the DOM processing mode in the same scenario. Service compositions with more than 2000 messages per second are rather rare. Hence, it is very unlikely that the CEP runtime will constrain the performance of a composition engine that uses our monitoring system, even in the most demanding environments.

4.2 Discussion

Stakeholders. The primary stakeholders of the monitoring system we propose are *domain experts*. Similar to a business analyst, a domain expert has solid knowledge of the business domain, e.g., the telecommunications domain. Moreover, and different from the business analyst, a domain expert has more in depth know-how of the technical details of the underlying services and systems. Domain experts only have basic programming skills, which separates them from developers or system engineers.

Defining Event Queries. When applying the proposed Monitoring Runtime to the Phonyfone scenario, the domain expert does not have to deal with a complex system or service setup. The inherent unobtrusiveness of our approach, i.e. the automatic capturing of services and the related service compositions deployed in VieDAME, enables the domain expert to focus on the definition of monitoring events and the analysis of monitoring results. The first step in defining monitoring events is to decide which event type satisfies the monitoring requirement. The domain experts leverages the UI to create EP statements using base events or domain specific events (cf. Section 3.1). Certainly, existing events can be combined into more complex events. For the following, we assume that the domain expert wants to determine the average response times of all available services (Listing 5a), and creates a domain specific event (Listing 5b) triggered in case the AddSvc request for an AddSub request is not received within 10 seconds.

```
1 select
2   median(executionTime), operation.name
3 from
4   ServiceInvocationEvent
5 group by
6   operation.name
7
```

(a) Querying Service Response Times

```
1 every AddSubAlert =
2   AddSub ->
3     (timer:interval(10 sec)
4   and not
5     AddSvcAlert = AddSvc(
6       msisdn = AddSubAlert.msisdn
7     )
```

(b) Detecting Absent Messages

Fig. 5. EP Statements for Base and Domain Specific Events

For the base event from Listing 5a, only the EP statement needs to be entered in the UI. Considering the domain specific event from Listing 5b, the EP Runtime needs to know how to interpret the `msisdn` attribute (line 6) of the AddSub and AddSvc events. The domain expert has two choices to communicate the meaning of the attribute. First, if a schema definition for the XML message exchange is available, the XPath expression required to reference the attribute can be inferred automatically. Second, in lack of a related schema definition, the XPath expressions have to be setup manually. Assuming that no schema definition is available, the `<Msisdn>` element from the AddSub request (cf. Section 2) is selected with the expression `/soapenv:Envelope/soapenv:Body/addSubscriber/subscriber/msisdn`.

Once the EP statements have been defined, the domain expert chooses a EP statement listener template. These templates support the domain expert in integrating Event Sinks (cf. Section 3.3). Assuming that a remote Web service should be invoked for events matching the statements shown in Listing 5, the domain expert chooses the Web service statement listener template and adapts it to specific needs, e.g., defines the endpoint URL or authentication credentials. All required steps can be performed during runtime, which minimizes time-to-market and maximizes responsiveness to new and changing monitoring requirements. For more examples of domain specific events, please consult the project website[2].

5 Related Work

A large body of monitoring work exists that can be classified as assertion-based monitoring. Baresi and Guinea [3,4] propose WSCoL, a constraint language for the supervision of BPEL processes. Monitoring information is specified as assertions on the BPEL code. Their approach uses Aspect-Oriented Programming (AOP) to check the assertions at runtime. In [12], Baresi et al. propose SECMOL, a general monitoring language, that sits on top of concrete monitoring languages such as WSCoL. Thus, it is flexible by separating data collection, aggregation and computation from the actual analysis. Sun et al. [19] also propose a monitoring approach based on AOP. Their goal is to check business process conformance with the requirements that are expressed using WS-Policy. The properties (e.g., temporal or reliability) of a Web service are described as Extended Message Sequence Graph (EMSG) and Message Event Transferring Graph (METG). A runtime monitoring framework is then used to monitor the corresponding properties that are then analyzed and checked against the METG graphs. Wu et al. [22] propose an AOP-based approach for identifying patterns in BPEL processes. They use a stateful aspect extension allowing the definition of behavior patterns that should be identified. If identified, different actions can be triggered. It also allows to monitor certain patterns by using history-based pointcuts. However, monitoring is restricted to instances of a BPEL process.

In contrast to the aforementioned approaches, we do not use assertions to define what needs to be monitored using a proprietary language. We propose a holistic and flexible monitoring system based on CEP techniques. Our system measures various QoS statistics by using a non-intrusive AOP mechanism and has access to message payloads of the composition engine. Additionally, it allows to connect external event sources that might be of interest. CEP techniques are very powerful because they operate on event streams (i.e., the monitoring data). CEP operators, e.g., for temporal or causal event correlation, can then be used on event streams to define what monitoring information needs to be retrieved. Domain experts can then use the simple visual tool to define event queries without programming expertise.

Another group of work focuses on monitoring temporal properties of Web service compositions. Kallel et al. [13] propose an approach to specify and monitor temporal constraints based on a novel formal language called XTUS-Automata. Monitoring itself is based on AO4BPEL [8]. Similar to that, Carbon et al. [2] propose RTML (Runtime Monitoring Specification Language) to monitor temporal properties. Their approach translates monitoring information to Java code to monitor instances of services and process classes. Contrary to these approaches, we do

[2] `http://viedame.omoser.com`

not need to translate a formal language to specific monitoring code. However, our system provides an holistic view on all available monitoring information as event streams. An event query language can then be used retrieve temporal properties.

Suntinger et al. [20] provide a visualization approach using the notation of an Event Tunnel. It allows to visualize complex event streams of historical information and enables the detection and analysis of business patterns. This can be used, for example, to detect and prevent fraud. Finally, Beeri et al. [6,5] propose a query-based monitoring approach and system. It consists of a query language allowing to visually design monitoring tasks. The designer also allows to define customized reports based on those queries. Complementary to our work, we also allow a query based approach based on an event query language. However, we do not provide reporting functionality.

6 Conclusion

In this paper we introduced a generic monitoring system for message based systems. It interprets each message as an event of a particular type and leverages complex event processing technology to define and detect situations of interest. Our system can be used to correlate events across multiple service compositions and integrates with external event providers to minimize fragmentation of monitoring data. Moreover, the inherent unobtrusiveness of the approach makes our monitoring system applicable to most current and future message based systems. This minimizes integration and setup costs. The evaluation of the prototype shows only a very small performance penalty. We also emphasized on the practical usefulness of our approach.

For future work, we are planning to provide visual support for EP statement creation to assist users who are not familiar with the underlying event query language to define basic event patterns and queries. Another point of interest includes service life cycle events, which were out of scope of this work. We will investigate how life cycle events can be combined with other runtime events to extract information related to service evolution.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-Cube).

References

1. Active Endpoints: ActiveBPEL Engine (2007), `http://www.active-endpoints.com/` (last accessed: September 07, 2010)
2. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-Time Monitoring of Instances and Classes of Web Service Compositions. In: IEEE Intl. Conf. on Web Services (ICWS 2006), pp. 63–71. IEEE Computer Society, Los Alamitos (2006)
3. Baresi, L., Guinea, S.: Self-supervising BPEL Processes. IEEE Transactions on Software Engineering (2010) (forthcoming)
4. Baresi, L., Guinea, S.: Dynamo: Dynamic Monitoring of WS-BPEL Processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 478–483. Springer, Heidelberg (2005)

5. Beeri, C., Eyal, A., Milo, T., Pilberg, A.: Query-based monitoring of BPEL business processes. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD 2007), pp. 1122–1124. ACM, New York (2007)
6. Beeri, C., Eyal, A., Milo, T., Pilberg, A.: BP-Mon: query-based monitoring of BPEL business processes. SIGMOD Rec. 37(1), 21–24 (2008)
7. Chandy, K., Schulte, W.: Event Processing - Designing IT Systems for Agile Companies. McGraw Hill Professional, New York (2010)
8. Charfi, A., Mezini, M.: AO4BPEL: An Aspect-oriented Extension to BPEL. World Wide Web 10(3), 309–344 (2007)
9. EsperTech: Esper (2009), `http://esper.codehaus.org` (last accessed: October 25, 2009)
10. EsperTech: Esper EPL Documentation (2009), `http://esper.codehaus.org/esper-3.5.0/doc/reference/en/html/index.html` (last accessed: September 11, 2010)
11. Faison, T.: Event-Based Programming: Taking Events to the Limit. Apress (2006)
12. Guinea, S., Baresi, L., Spanoudakis, G., Nano, O.: Comprehensive Monitoring of BPEL Processes. IEEE Internet Computing (2009)
13. Kallel, S., Charfi, A., Dinkelaker, T., Mezini, M., Jmaiel, M.: Specifying and Monitoring Temporal Properties in Web Services Compositions. In: Proc. of the 7th IEEE European Conf. on Web Services (ECOWS 2009), pp. 148–157 (2009)
14. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
15. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection. In: Proc. of the 4th Intl. Workshop on Middleware for Service Oriented Computing (MWSOC 2009), pp. 1–6. ACM, New York (2009)
16. Moser, O., Rosenberg, F., Dustdar, S.: Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In: Proc. of the 17th Intl. World Wide Web Conf. (WWW 2008), pp. 815–824. ACM, New York (2008)
17. OASIS: Web Service Business Process Execution Language 2.0 (2006), `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel` (last accessed: April 17, 2007)
18. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer 11 (2007)
19. Sun, M., Li, B., Zhang, P.: Monitoring BPEL-Based Web Service Composition Using AOP. In: Proc. of the 8th IEEE/ACIS Intl. Conf. on Computer and Information Science (ICIS 2009), pp. 1172–1177 (2009)
20. Suntinger, M., Schiefer, J., Obweger, H., Groller, M.: The Event Tunnel: Interactive Visualization of Complex Event Streams for Business Process Pattern Analysis. In: IEEE Pacific Visualization Symposium (PacificVIS 2008), pp. 111–118 (2008)
21. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Leymann, F., Dustdar, S.: Monitoring and Analyzing Influential Factors of Business Process Performance. In: Proc. of the 13th IEEE Intl. Enterprise Distributed Object Computing Conf. (EDOC 2009), pp. 141–150. IEEE Computer Society, Los Alamitos (2009)
22. Wu, G., Wei, J., Huang, T.: Flexible Pattern Monitoring for WS-BPEL through Stateful Aspect Extension. In: Proc. of the IEEE Intl. Conf. on Web Services (ICWS 2008), pp. 577–584 (2008)
23. Zdun, U., Hentrich, C., Dustdar, S.: Modeling Process-Driven and Service-Oriented Architectures Using Patterns and Pattern Primitives. ACM Transactions on the Web (TWEB) 1(3), 14:1–14:14 (2007)