WS-Policy based Monitoring of Composite Web Services

Abdelkarim Erradi1, Piyush Maheshwari2,1, Vladimir Tomic3,1

1School of Computer Science and Eng. 
Uni. of New South Wales, Sydney, Australia
aerradi@cse.unsw.edu.au

2IBM India Research Lab 
New Delhi, India
piyush@in.ibm.com

3NICTA, Sydney, Australia
vladat@computer.org

Abstract

MASC (Manageable and Adaptive Service Compositions)1 is a policy-based middleware for monitoring and control of composite Web services execution. The monitorable requirements are specified in the WS-Policy4MASC language that extends WS-Policy by defining new types of monitoring and control policy assertions. This paper focuses on MASC monitoring capabilities to detect business exceptions and runtime faults. Our solutions are complementary to the existing approaches and provide: synchronous and asynchronous monitoring both at the SOAP messaging layer and the process orchestration layer, greater diversity of monitoring and control constructs, as well as the externalization of monitoring and adaptation actions from definitions of business processes. We implemented a MASC proof-of-concept prototype and evaluated it on monitoring and adaptation scenarios from a stock trading case study. Our performance studies indicate that MASC overhead and scalability are acceptable.

1. Introduction

Organizations are increasingly using composite Web services to automate business processes, via dynamically selecting and assembling a set of autonomous and loosely-coupled Web services, possibly from different service providers. However, Web services based integration builds a web of interdependencies between collaborating systems and introduces various challenging interoperability and management issues as participating services may change or behave in unpredictable ways.

Hence, composite Web services execution has to be continuously monitored to provide sense-and-respond feedback loops and to check compliance to runtime policies in order to detect and adapt to business exceptions and faults. The latter are non-linear and their occurrence is often unpredictable (i.e., the timing and the sequence of the exceptions and fault events affecting the composite service often cannot be defined in advance). Current composite service specification languages and orchestration engines offer limited support to accommodate these requirements. There have been several research projects in this area, such as [1, 4]. However, monitoring and runtime adaptability is not yet adequately supported by dominant Web service composition languages, such as WS-BPEL [8]. Additionally the exception handling mechanisms offered by the process orchestration engines do not provide sufficient support for monitoring to detect and handle the broad range of business exceptions or faults that may occur during the process execution. Moreover, the monitoring logic is often scattered across different modules and tangled with the functional specification and implementation of the normal process flow. This negatively impacts maintainability and increases design complexity and development costs. To address the requirements for manageable and adaptive composite Web services, we propose a policy-based approach to runtime monitoring to detect business exceptions and faults. The central part of the approach is our lightweight Web service management middleware MASC (Manageable and Adaptive Service Compositions) [5] that performs runtime monitoring and adaptation while incurring with low run-time overhead. For formal specification of policies MASC uses WS-Policy4MASC [11, 12], which is our novel extension of the Web Services Policy (WS-Policy) Framework [13]. This externalization of monitoring and adaptation aspects yields higher degree of flexibility, promotes reusability and contributes to keep the specification of the base process simpler and easier to maintain.

Our previous work, such as [2, 3], concentrated on adaptation strategies and mechanisms to handle business exceptions and manage runtime faults and quality of service (QoS) degradations of one or more constituent Web services. This paper complements our past publications and focuses on the monitoring strategies and mechanisms supported by our approach.

This section provided an introduction to our research while the next section summarizes our motivating scenarios. The third section compares our research with related work. Then the paper discusses key monitoring patterns supported by both WS-Policy4MASC and the MASC middleware. Section 5 overviews WS-Policy4MASC language. Section 6 elaborates MASC middleware monitoring components and their implementation. In Section 7, we present MASC middleware evaluation on a stock trading case study to
illustrate the effectiveness of our approach. The last Section concludes and outlines our future work.

2. Motivating Example

The need for monitoring and dynamic adaptation of composite Web services can be illustrated using parts of the Stock Trading case study that we have used to determine requirements for our work in this area and evaluate our solutions. The base Trading Process is determined requirements for our work in this area and evaluate our solutions. The base Trading Process is initiated when a human investor places an investment or redemption order with their FundManagerService. The latter, after verifying the order, invokes the FinancialAnalysisService to get a recommendation to enable an informed investment/redemption decision. The FinancialAnalysisService gets periodic notifications from the StockNotificationService about the current stock values and real-time market surveillance, announcements, and quotes. Based on this information, historical records, and predictive models built into the service (for our prototype, we used very simple models), the FinancialAnalysisService informs the FundManagerService about how well certain stocks are performing. The FundManagerService makes a decision which stock to buy/sell for the monetary amount requested by the investor. Then, the FundManagerService sends the buying/selling request to the StockMarketService. The latter performs a simple trade matching between the buy orders and the sell orders. When a trade match is formed, the StockMarketService invokes in parallel the StockRegistryService to transfer the stock share ownership and the PaymentService to transfer funds.

The following scenarios illustrate the need for the functional and Quality of Service (QoS) monitoring of composite Web services to trigger timely dynamic adaptation to handle business exceptions and faults:

- Monitoring the GetAccountBalance process to examine incoming AccountBalance messages and in case of a foreign portfolio an adaptation trigger can be raised to dynamically add a CurrencyConversion Web service to convert prices of foreign stocks to a local currency.
- Monitoring the invocation of the PaymentService to check the payment amount against the trade amount. When underpayment is detected (i.e., the payment received is less than the trade amount) then a process adaptation could be triggered to calculate the amount still owing and issue a residue invoice to the customer. Whereas in case overpayment the process adaptation could dynamically add a sub-process to notify the customer, calculate the over-paid amount and refund it.
- Monitoring the Order Placement process and depending on the Trade Transaction Amount and/or the Investor's Profile a Credit Rating Web service might need to be added prior to submitting the trade order to the StockMarketService.
- Monitoring missing events such as the Trade Payment not received within a particular timeframe (e.g., NoPaymentAfter30Days event).
- Monitoring that the response time of the GetStockPrice operation is less than 10 seconds otherwise the returned price should not be considered.

From the analysis of the above scenarios we derived the key monitoring requirements to be supported by the WS-Policy4MASC language and the MASC middleware. The monitoring policies should support specifying:

- Monitoring of exchanged messages (Input, Output and Faults declared in the WSDL description of composed services) with the ability to define message filters and queries against incoming/outgoing messages to detect events of interest raise the corresponding adaptation trigger events.
- QoS monitoring to determine the health of the monitored services to trigger adaption in case of violation of QoS guarantees.
- Monitoring to detect missing or late arrival of messages.
- Monitoring logged messages and process execution traces to compute composite process performance metrics and to derive further adaptation trigger events.

3. Comparisons with Related Work

Current Web services management research and standardization efforts emphasize easing distributed Web services management via defining standards based interoperable management interfaces and metadata. For example, WS-Agreement [4] defines protocols and schema for establishing and monitoring service level agreement between independent services. However, existing approaches focus mainly on managing single services or managing the infrastructure supporting the service execution using SLA-driven configuration of managed resources. They address management concerns from resources allocation perspective to guard against violation of service levels and to optimize resources usage. For example, if the service load increases, the management infrastructure allocates additional resources to meet the service guarantees. Our work focuses on monitoring the execution of composite Web services (rather than underlying resource monitoring) in order to detect, correlate and react to business meaningful events.

The closest related work to ours is the service monitoring approach presented in [1]. The authors proposed the Web Service Constraint Language (WS-CoL) to annotate BPEL processes with monitoring rules. At deployment time, WS-CoL constraints attached to a process are translated into WSBPEL invoke activities that call external monitoring services to evaluate the monitoring rules and detect anomalous conditions. This approach is similar to ours in that monitoring rules are
specified externally rather than being embedded into the process specification. It achieves the desired reusability and separation of concerns. On the other hand, our approach supports both functional and QoS monitoring using a hybrid approach that combines synchronous and asynchronous monitoring techniques. Also, in our approach the supervision of monitoring policies can be either delegated to the SOAP messaging middleware or enacted by the process orchestration engine via dynamic adaptation of process instances.

The authors in [10] propose a framework for monitoring the requirements of BPEL compositions. Their approach uses event calculus to specify the system’s behavioral properties to be monitored. Events are then observed and recorded at runtime, and then inconsistencies between the system’s expected behavior and the recorded behavior are detected using deductive and abductive reasoning algorithms. Their proposed approach is less intrusive as the monitoring is performed in parallel with the execution of the business process. This leads to less impact on performance but yields lesser degree of timely responsiveness when anomalies arise. Hence the approach is only applicable when timeliness in the detection of a deviation is not critical for a system. Also the applicability of the proposed approach is hindered by its complexity particularly for users not familiar with event calculus.

Our approach has unique characteristics. We adopt a policy-based monitoring and adaptation approach that builds on the established policy-based management principles [9] combined with Event-Condition-Action (ECA) rules. This yields decoupling between sensors that monitor and detect adaptation triggers and effectors that select and execute the most appropriate actions to handle such triggers. Additionally, our technological base is different from related work which leads to different architectural solutions. Our approach extends WS-Policy and .NET 3.5, which are under explored. MASC approach also supports adaptive monitoring by allowing the specification and execution of process instance-specific monitoring, for example, the data to be collected and the monitoring rules to be evaluated can vary based on the service invoker profile or the access channel used to access the service.

4. WS-Policy4MASC Support for Monitoring

The MASC monitoring framework uses a policy-based approach for non intrusive monitoring of composite Web services, i.e., without requiring any modification to the specification of the composite service or participating services. MASC uses a policy language, named WS-Policy4MASC [11, 12], for declaratively specifying service monitoring policies (and other types of policies). Following the review of key languages for the specification of policies and service level agreements for Web services, including WS-Policy [13], WS-Agreement and WSLA [4], our decision has been to adopt WS-Policy and extend it by defining XML schemas with new types of policy assertions to support monitoring and adaptation of composite Web services. WS-Policy was selected because it is widely used in practice and supported by most Web services platforms. WS-Policy4MASC is described in detail and illustrated in [11]. This section summarizes its key 4 assertion types:

1. Goal policy assertions specify requirements and guarantees to be met in desired normal operation (e.g., response time of a particular activity has to be less than 5 seconds). In MASC, they guide monitoring activities. The monitorable requirements are specified as one or more goal policy assertions specifying functional and QoS guarantees to be met in desired normal operation (e.g., execution of some action). They can be used by MASC for billing and for selection between alternative action policy assertions.

2. Action policy assertions specify a group of actions to be taken if certain conditions are met (e.g., some guarantees were not satisfied). Examples of these actions are removal, addition, replacement, skipping, retrying of a sub-process (or individual activity) or process termination. In MASC, they guide adaptation and other control actions such as monitored data exchange.

3. Utility policy assertions specify monetary values that indicate the business value associated with particular situations (e.g., execution of some action). They can be used by MASC for billing and for selection between alternative action policy assertions.

4. Meta-policy assertions can be used to specify which action policy assertions are alternatives and which conflict resolution strategy should be used.

In addition to these 4 new types of policy assertions, WS-Policy4MASC enables specification of additional information that is necessary for run-time policy-driven management. For example, this includes information about conditions when policy assertions are evaluated/executed, parties performing this evaluation/execution, a party responsible for meeting a goal policy assertion, monitored data items, states, state transitions, schedules, events, and various expressions (e.g., Boolean and arithmetic with unit expressions).

The WS-Policy4MASC goal assertions used for monitoring allow specification of monitoring configuration and the specification of monitoring rules. Monitoring configuration specifies the source of the runtime data to be used for monitoring, configuration of data collectors, way in which monitoring is to be performed (synchronous vs. asynchronous), regularity and timing when monitoring rules should be checked, management party responsible for monitoring (process layer vs. messaging layer), and the recipient(s) of the monitoring results (e.g., detected violations). The monitoring rules express the conditions that should be satisfied by a composite service or its constituent services.
The conditions of the monitoring rules combine WS-Policy expressions specifying predicates on monitored data items originating either from within the process, the exchanges messages, QoS monitoring modules or external variables (e.g., context data). The monitoring conditions specify either (1) the desired service functional behavior in terms of pre-conditions and post-conditions that express constraints over exchanged messages, or (2) quality requirement in terms of thresholds on QoS metrics (e.g. service response time, throughput and mean time between assertions).

Using WS-PolicyAttachment [13] standard, the monitoring policies can be attached to monitoring points at various levels of granularity such as a composed service, a process endpoint or a service operation. For example, a monitoring policy could be attached to the GetMarketValueOfPortfolio operation exposed by the RegistryService to examine whether a foreign currency value is received and, if this is the case, to trigger an adaptation to add a call to a CurrencyConversion service. This example is illustrated in Figure 1 and 2. It illustrates a simple monitoring policy expressed using WS-Policy4MASC constructs. (Many additional examples, including various adaptations of Web service compositions, are given in [11].)

Figure 1 shows how WS-Policy4MASC can be used to specify the post-condition that the currency code in the reply message of GetMarketValueOfPortfolio operation (from the RegistryService) should be 'AUD'. Otherwise an adaptation trigger should be raised to add a currency conversion. WS-Policy4MASC policy assertions and other constructs are specified within a WS-Policy element (namespace attributes are omitted from Figure 2 for brevity). First, the <MonitoredDataItem>, <MonitoredDataCollection>, and <ActionGroup> constructs specify that the message part "CurrencyCode" is monitored. Definitions of states and events follow, but they are omitted from Figure 2. Then, a <When> construct referring to the state "Executing" and the event "GetMarketValueOfPortfolio-ReplyReceived" is defined. The subsequent action policy assertion specifies that when this event happens in this state, monitoring of the message part "CurrencyCode" is performed by the Orchestrator. This action policy assertion is used to configure MASC monitoring modules. Definition of the Boolean expression "IsForeignCurrencyExpression" is omitted from Figure 2 for brevity. This Boolean expression specifies that values of the monitored data item (the message part "CurrencyCode") must be 'AUD'. The above-mentioned <When> construct and Boolean expression are referenced in the definition of the subsequent goal policy assertion, which is also used to configure MASC monitoring rules evaluator. This goal policy assertion specifies that when the event "GetMarketValueOfPortfolio-ReplyReceived" occurs in the state "Executing", then the MonitoringRulesEvaluator should evaluate the mentioned Boolean expression. It also states that the Orchestrator is responsible for meeting this goal. In a separate file, shown in Figure 2, the standard WS-PolicyAttachment element defines that this policy (referenced in the <PolicyReference> element) is applied to the service operation specified in the <AppliesTo> element.

5. Supported Monitoring Patterns

This section discusses the key monitoring patterns learned from our experiences in designing WS-Policy4MASC and implementing the MASC middleware. We concentrate on patterns that are both useful and practical and highlight how MASC supports them:

- **Message Filters** define predicates over the content of exchanged messages message. For example a message filter monitoring a stream of trade orders may filter all high priority orders where the condition is Priority = 'High' and Amount > 20000. This example presents the simplest kind of filter where the query condition only involves the content of one message. In the stock trading...
scenario, a filter may be used to filter all trades where the volume is too small, or all trades that refer to low performing stock symbols. More complex scenarios could require specifying more complex filters, for example, filters that compare messages to other messages in the same stream, or in another stream, or compare message content to a computed metric. For instance, a filter might capture orders where the purchase amount is larger than the previous purchase amount, or purchase amounts that are larger than the average for the previous day.

- **Message Logging** pattern is used to keep a history of monitored messages and QoS metrics (in an in-memory cache or in a monitoring database). Often it is necessary to write raw or processed (filtered, aggregated or correlated) messages into a relational or XML database. This allows multiple messages to be kept and used for correlation and further data aggregation.

- **Message enrichment** pattern could be used prior to (or in response to) the evaluation of monitoring conditions. This is often achieved by accessing a database or invoking a Web service (while passing as parameters some extracted data from the incoming message) to retrieve historical or reference data to augment the incoming messages (e.g., piggybacking reference or management information into SOAP headers). Typically a message comes into the process, the monitoring engine intercepts it and issues an SQL request to a database (or call Web services interfacing external data services) and passes some extracted message data as parameters to the database query. The database returns a result that the monitoring engine combines with the data from the message, and forwards the enriched message to the next message handler for further processing. An example for this could be to look up the average price for a stock, or to lookup the Ordered Quantity for a particular order when a delivery notification message arrives to be able to compare the OrderedQuantity with the ReceivedQuantity.

- **QoS Monitoring** via measurement of basic metrics statistical metrics and calculation of derived metrics that aggregate other metrics (possibly over various time moving windows). Using the messages and QoS measurements logged in the monitoring data store (or the in-memory cache), this pattern maps individual QoS measurements into business meaningful events. The kinds of possible aggregators include running averages, sums, counts, minimum, maximum, standard deviation and user-defined aggregators. The kinds of windows that can be used to trigger the aggregation include time-based and count-based windows, sliding windows, etc. The aggregation output frequency can be either continuous or periodic. For continuous output each incoming message updates the calculated expression, and an output event is produced. With periodic output, the calculated expression is updated and published only periodically, for example every 2 minutes. An example of this pattern could be to compute the average, maximum and minimum response time of used service operations every 5-minutes interval.

- **Messages Correlation** uses correlation rules to analyze individual messages (across multiple message streams) over time and length-windows to detect patterns between these messages and derive business meaningful events from sets of lower level messages/events. This usually requires logging messages to a database and evaluating the correlation rules. For example, a WS-Policy4MASC monitoring policy can specify how to detect a speculative customer event which could be defined as an investor selling a stock immediately after buying it 3 times within a particular time interval. These correlation rules can be used to specify a pattern of events whose occurrence at runtime implies the violation of a goal policy.

The types of events relationships supported by WS-Policy4MASC are:

- Disjunction, any 1 of the m specified events happened
- Conjunction, all specified events happened in any order
- Sequencing, all events happened in the specified order
- Negation, an event did not happen within a deadline or before another event. For example a shipped order event did not occur within a specified time period or it happened before the payment event.

6. MASC Middleware Monitoring Components

   **Architecture and Implementation**

   MASC supports both synchronous and asynchronous monitoring through coordinating the activities of the monitoring components at the messaging layer and at the process orchestration layer.

   - Synchronous monitoring directly collects data and evaluates monitoring rules against the collected data while being on the process execution path. The execution of the composition process waits until the monitoring service returns the result of the check and it may continue or raise an exception if the monitored assertion has been violated. Calls to the data collectors and the monitoring rules evaluator are weaved into the running process and the messaging pipeline at runtime. This yields an instrumented process capable of detecting and reacting to condition changes. The key advantage of synchronous (active) monitoring is that it allows immediate reaction to business exceptions and faults. However synchronous monitoring is only suitable for monitoring simple conditions that do not require extensive computation. Otherwise, it could introduce unacceptable overhead as the process execution halts while the configured monitoring policies are verified.

   - Asynchronous monitoring uses data collectors and analyzers which are external to the composed service to be monitored. The monitoring data gets collected without data collectors in the message processing pipeline. Rather
the data get sourced from various data sources, deployed either on the same runtime environment or at a remote environment. Examples of such sources are process execution logs, process execution traces and QoS metrics measurements from service providers or management intermediaries. The data collection and monitoring rules evaluation is performed in parallel with the composed service execution. Thus, it does not affect the performance of the composed service. The asynchronous (passive) monitoring is more suitable when timeliness in the detection of a deviation is not critical for a system. For example, aggregating and analyzing past QoS metrics then taking process optimization actions such as lowering the score of poor performing services in way that they do not get selected if better alternative services exist.

The overall architecture of MASC middleware has been reported in previous publications, such as [2]. MASC is features rich middleware with many modules, more details as well as the prototype implementation and examples can be found in [5]. For readability purposes, this section only focuses on the details, the interactions and the inner working of key MASC middleware monitoring components shown in Figure 3 (because of limited space the full details of the interfaces exposed by each component will be posted in [5]).

![Figure 3- Architecture of MASC middleware and its Monitoring Components](image)

The MASC middleware is designed using a service-centric approach with components exposed as message-based services. The service-oriented design leads to more loosely coupled parts communicating through well-defined interfaces. MASC middleware services can be either co-located within a single domain or they can interact across multiple management domains. They support 4 key functions: policy deployment, configuration of monitored data collection, monitoring rules evaluation and the reporting of events and adaptation triggers.

We have been developing the MASC middleware prototype by extending the newly released Microsoft .NET Framework 3.5 [6], particularly its Windows Communication Foundation (WCF) and the Windows Workflow Foundation (WF). WCF is Microsoft’s next-generation platform for building secure, reliable, and transacted Web services through a simplified programming model that unifies and extends many of the previous .NET technologies. WF provides an extensible framework for building processes (workflows) and embedding them into .NET applications to orchestrate activities of objects and services. WF processes can be defined using either Microsoft’s Extensible Applications Markup Language (XAML) or BPEL. To execute a process, WF has a lightweight runtime engine that can be hosted in any .NET application. The WF runtime engine manages the instantiation and execution of workflow activities. Additionally, it takes care of different middleware concerns through an extensible set of WF runtime services (e.g., Tracking, Persistence and Transaction support are built-in). Therefore, we designed and implemented another WF runtime service, named MASCMonitoringService to manage the monitoring of Web services compositions implemented as WF processes. The WF runtime engine can be configured to engage the MASCMonitoringService.

(1) **MASCPolicyRepository**: MASC middleware stores and manages the WS-Policy4MASC policy assertions in an in-memory policy repository, which is a collection of instances of policy classes generated automatically from the WS-Policy4MASC schema, using an XML-Schema-to-C#-classes generator. The MASCPolicyRepository
exposes a service interface to load, activate, deactivate and query policies. When a policy is loaded into the repository, it gets parsed and checked for validity and conformity to the WS-Policy4MASC schema. Then it gets analyzed to extract the relevant policy assertions for each management party. The repository then distributes the loaded policy in full or in appropriate parts to each management party specified in the policy. A management party could be either a MASC middleware component (such as MASCMonitoringService), a management intermediary or a monitoring service exposed by the service provider. For example, QoS computation schedules extracted from WS-Policy4MASC policy are sent to MASC Scheduler Service. Note that the repository does not deploy the whole policy to a management party but only sends the relevant policy assertions such that the Management Party can map the received policy to a suitable configuration of its components. For example the MASCRuleEvaluation-Service only gets the GoalPolicy assertions it needs to evaluate. Similarly, when a policy gets deactivated or expires, MASCPolicyRepository notifies affected management parties. We assume that all management parties expose standard configuration interfaces to receive policy assertions from the MASCPolicyRepository. Emerging standards such as Web Services Distributed Management [7] could be leveraged for this purpose. However, for the sake of simplicity, in the MASC prototype we use simple and uniform proprietary interfaces to deploy policies to management parties.

(2) MASCMonitoringService: This component is implemented as a custom WF Runtime Service. It is responsible for initiating and controlling monitoring activities to continuously monitor interactions between the participating services to verify that the configured goal policies are being satisfied. As process instances get created and process activities get executed, MASCMonitoringService collects process level data such as process lifecycle events (e.g., Process Instance Started Event, Process Instance Completed Event), activity lifecycle events (e.g., Activity Started Event, Activity Completed Event) and data manipulated by the process such as the number of iterations executed by a while-loop activity. When a process is instantiated or a process activity starts execution, the MASCMonitoringService queries the MASCPolicyRepository and computes the monitoring policy relevant for the process instance and/or the process activity. Then it extracts the specification of the MonitoredDataItems to be collected from the goal policy assertions and supplies the required configuration to the SOAP Message Examination Service. It also extracts the rules to be monitored and configures the MASCRuleEvaluationService. The MASCMonitoringService also configures the selected SOAP Message Examination Service and MASCRuleEvaluationService so that they exchange monitored data directly at runtime (this is achieved using WS-Addressing ReplyTo header).

(3) SOAP Message Examination Service: This component has the responsibility of examining the exchanged messages to extract the primitive monitored data items needed populate the MonitoredDataItems defined in the policy. Various techniques are used by this component to perform data collection:

(a) Message Inspectors are message handlers configured in the WCF messaging pipeline to evaluate message filters on message streams and collect the monitored data items. This component is implemented using a custom WCF ProxyMessageInspector and a custom Dispatcher-MessageInspector to examine incoming and outgoing message streams. Primarily, the message handlers use XPath over exchanged messages to populate the appropriate MonitoredDataItems with the values contained in the header or the body of the exchanged SOAP messages (e.g., the CustomerID in a TradeOrder message). The message filters are also used to detect fault events based on fault messages returned by invoked services, as specified in their WSDL interface.

(b) MASCDataResolver is used to fill in the missing data and enrich the monitored data items by retrieving and attaching data from an external data source (e.g., accessing a customer profile database to establish whether a service requester is a Gold customer). During the evaluation of monitoring assertions, a monitoring policy might reference data from external sources, such as the MASC monitoring data store, to obtain data not available in the exchange messages. The source of such external data can be specified as Web service calls in the monitoring assertions, such as calling the QoS measurement service or querying the log of prior interactions to get some historical data. Note that data retrieval to populate missing information is initiated only if all other monitoring rule conditions that do not require additional information are true.

(4) MASCRuleEvaluationService: This component evaluates conditions specified in WS-Policy4MASC goal policy assertions to check if the monitored data items satisfy the configured monitoring rules. This is done to discover business exceptions and adaptation triggers (e.g., OrderPartiallyDelivered event when the received quantity is less than the ordered quantity, LatePayment, OverPayment and UnderPayment) and to detect any condition changes such as faults (e.g., FailedServiceInvocation, ViolatedServiceQoS such as the invoked service not responding within the time specified in a goal policy. This component also evaluates the event correlation rules discussed in Section 5. MASC implementation leverages WF Rule Engine to evaluate the monitoring rules by autotransforming the WS-Policy4MASC expressions into WF rules and submitting them to engine for execution.
When an undesirable condition is detected (e.g., a goal policy assertions is not satisfied or a fault is detected), then the MASCRuleEvaluationService raises the appropriate event to notify the MASCPolicyDecision-Point. Examples of events could be ProviderService-Unavailable, GoalPolicyViolation, ServiceInvocation-Timeout. The event is reported to the MASCPolicyDecisionPoint along with all the data required for recovery (i.e., the ProcessInstanceId and an EventData that contains data relevant for adaptation.)

Upon the receipt of a notification, the MASCPolicy-DecisionPoint takes the appropriate adaptation decisions (e.g., retry the invocation of a faulty service) to correct the problem and restore the service’s correct execution and quality of service. The adaptation decision is guided by the action policy assertions specified in WS-Policy4MASC. The decision also considers the utility policies and the business value associated to adaptation actions. More details about MASC adaptation process and the supported adaptation actions can be found in [2, 3].

(5) Monitoring Data Store: This component expose an interface to store and retrieve the monitoring data items. It also performs periodic analysis and aggregation of the monitored data and sends it to the MASCRuleEvaluation-Service. Such as, averaging the response time of invoked services every 5 minutes.

(6) QoS Measurement Service is responsible for computation of QoS metrics by periodically evaluating configured WS-Policy4MASC arithmetic expressions against the data items logged by the data collectors. For example, the execution duration of a process activity is calculated using the timestamp attributes of the ActivityStartEvent and the ActivityCompletedEvent. Similarly, the end-to-end execution duration of a composite service is calculated using the timestamp attributes of the ProcessStartEvent and the ProcessCompletedEvent. The QoS computation is triggered either by a receipt of a particular monitored data item or by the occurrence of a Time Event (raised by the Scheduler Service). The key QoS metrics calculated by this component are: (a) Response Time (the time interval between when a service is requested and when it is delivered); (b) Reliability (ratio of successful invocations over the number of total invocations in given period of time); (c) Availability (the percentage of time that a service is available during a period).

(7) Scheduler Service is used to activate periodic management actions using scheduled events. The Scheduling Service can be configured to emit Time Event such as point in time reached or time period elapsed e.g., 5 minutes passed. Such Time Events are used to trigger further analysis of the collected monitoring data (e.g., aggregation of QoS metrics by updating a moving average response time every 5 minutes) or to trigger further monitoring rules evaluation. For example, an event could be raised when the purchase order from a gold customer is not shipped within two days of the order. The sequence diagram in Figure 4 depicts interactions and data flows between key MASC monitoring services.

Figure 4- Interactions between key MASC middleware Monitoring Components

7. Performance Evaluation of MASC Middleware Monitoring Components

To validate the feasibility and efficiency of our framework, we performed a series of experiments to measure the performance overhead introduced by MASC synchronous and asynchronous monitoring and to demonstrate MASC applicability to a Stock trading case study. Our secondary aim for these tests was to measure the overhead incurred by each component to discover areas of the platform that needed further improvements. The overhead was measured in terms of average response
time and memory usage. We focused in particular on scalability tests, since scaling well for a high number of process instances is important for services because they often have many concurrent requesters. Similarly, scaling well for high number and complex monitoring policies with many assertions is an important property, since monitoring and adaptation policies are often complex.

In these experiments we used a .Net 3.5 implementation of MASC middleware (available at [5]). The scenarios used for evaluation were discussed in Section 1. In particular, we used a BPEL process, called Account Balance Process (ABP), offered by the FundManager-Service (FM) to allow investors to get their account balance in their local currency. The ABP is composed of 4 Web service operations: (1) GetCashBalance operation offered by an internal investment service (IS), (2) GetNumberOfUnitsInPortfolio operation offered by the StockRegistryService (SR), (3) GetStockPrice operation offered by the StockNotification Service (SN) and (4) a Calculator Service (CS) is used for calculations such as calculating the market value of portfolio.

In the experiments, a monitoring assertion is configured and attached to the ABP to synchronously examine incoming AccountBalance messages and in case of a foreign portfolio, an adaptation trigger can be raised to dynamically add a Currency Exchange Service (CES) and a Calculator Service (CS) to convert prices of foreign stocks to a local currency. Additionally, a monitoring assertion is specified as a constraint over the response time of CES to detect QoS thresholds violations. If GetStockPrice operation does not return results within 5 seconds, then the returned price is ignored and the MASC middleware dynamically adapts the process to use an equivalent SN service. For asynchronous monitoring, a policy is configured to examine the average response time of invoked services to be able to rank partner services according to their performance.

We used simulation to generate a number of ongoing process instances with different sets of data. The ABP client and the SN service pick up randomly two currency symbols from a set of 20 symbols. When the monitoring rules evaluator detects that the requested currency differs from the currency returned by the SN service then it emits an adaptation trigger event. To simulate different response times, the invoked services sleep for a random time interval to simulate delays of exchanged messages between the composed Web services and the ABP. The sleeping interval is uniformly distributed in the range of 0 and 20 seconds. Experiments were done in Windows Vista PC with 2.8GHz Intel processor and 1GB of RAM.

In the first test T1, the ABP process with embedded monitoring (i.e., the data collection and monitoring rules evaluation were defined in the process) was deployed on the WF engine without MASC middleware. In the second test T2, the base ABP process was deployed on the WF engine with MASC middleware services. In test T2, the data collection and the evaluation of monitoring rules were defined in separate WSPolicy4MASC assertions. The dynamic composition of the base process with the monitoring services was managed by MASC middleware. Tests were repeated 500 times and the average time elapsed between the creation of the process instance and the completion of that instance was measured in milliseconds. The results of these two tests are shown in Figure 5. As expected, the embedded monitoring clearly gives better performance than external monitoring, but the tradeoff is lose of flexibility, reusability and maintainability. As shown in Figure 1, the use of MASC increased the process execution time by 8% and memory usage by 10%, compared to response time and memory usage without MASC mediation. We argue that this would be acceptable in most cases (compared with the cost of interactions with partner Web Services). Moreover, this overhead decreases (from 8% to 5%) when the process contains more messaging activities.
in Figure 6. The figures on the vertical axis show the overhead percentage of MASC monitoring. As we can see from the graph, increasing the number of monitoring assertions causes only a linear increase in the average monitoring time. Similarly, increasing the number of process instances causes a linear increase in the average monitoring time. This shows that the monitoring strategies of MASC scales well both for a high number of instances, and for complex and high number of monitoring policies with many assertions.

The main source for the performance overheads of the MASC monitoring is the time taken to transform WS-Policy4MASC expressions into WF Rules, which is about 15% of the total overhead. This includes the overhead of parsing WS-Policy4MASC policies. We consider implementing caching mechanisms to offset the performance degradations introduced by policy parsing. The conducted experiments were successfully achieved because demonstrated the feasibility and the usefulness of the MASC monitoring approach.

The conducted experiments illustrated WS-Policy4MASC support for externalization and modularization of the monitoring crosscutting concerns by encapsulating all activities, variables, and algorithms that implement them in separate modules. Additionally, MASC monitoring approach supports flexible monitoring as the monitoring policies can be deployed/undeployed at runtime. The dynamic composition of monitoring aspects and processes has several benefits such as improving the adaptability of BPEL processes, providing more flexibility to switch monitoring on and off as needed, allowing an easier support of instance-based monitoring and adaptation that depend on runtime data. In addition to these benefits, the runtime overhead induced by the approach is small and acceptable. The performance measurements presented above confirm two claims. First, the additional overhead incurred for the evaluation of WS-Policy4MASC monitoring policies is acceptable when compared with the execution time of the messaging activity such as Invoke, which carries out a costly interaction with a partner services over the network. Second, the policy-based monitoring based on MASC middleware compares well to static monitoring embedded in the process definition.

8. Conclusions and Future Work

In this paper, we have presented WS-Policy4MASC and the MASC middleware to enhance composite Web services with policy-driven monitoring capabilities to detect business exceptions and faults and to autonomously adapt to various changes to provide uninterrupted service and satisfaction of quality of service (QoS) guarantees. MASC’s key advantage that it uses declarative policies (described in the new WS-Policy4MASC language) to provide a high degree of separation of concerns between the base process flow and the monitoring and adaptation logic. This results in improved flexibility and reusability compared to implementing monitoring and adaptation concerns as part of the process. Further, MASC coordinates the monitoring and the enforcement of run-time adaptation decisions across both the business process orchestration layer and the SOAP messaging layer. We have been evaluating the MASC proof-of-concept prototype on a stock trading case study. Our performance studies conducted with the prototype indicate that the scalability and the overhead introduced by MASC are acceptable.

For future work, we are investigating possibilities of integrating our framework with existing standards, such as WS-Agreement, to monitor and manage service level agreements. Additionally, we are considering a separate paper to report our experiences in building manageable composite Web services using .NET vs. J2EE particularly the different monitoring and adaptation policies and mechanisms supported by the two platforms.

References