Abstract

MASC (Manageable and Adaptive Service Compositions)** is a policy-based middleware for monitoring of Web service compositions and their dynamic adaptation to various runtime changes. MASC policies are described in our new WS-Policy extension called WS-Policy4MASC. Compared with recent related works, MASC has several distinctive characteristics, such as coordination of adaptation on the SOAP messaging layer and the business process orchestration layer, use of both technical and business metrics for adaptation decisions, and extending the power and flexibility of the new Microsoft .NET 3.0 platform. In this paper, we focus on MASC support for adaptation to address business exceptions and manage runtime faults. For example, a sub-process (or an activity) can be added, deleted, replaced, skipped, or retried. We have been implementing a MASC proof-of-concept prototype and evaluating it on adaptation scenarios from a stock trading case study. Our performance studies of the prototype indicate that overheads introduced by MASC are acceptable.

1. Introduction and Motivation

One important trend in Enterprise Application Integration (EAI) and Business-to-Business (B2B) integration is the increasing use of composite Web services to automate cross-organization business processes, via dynamically selecting and assembling a set of autonomous and loosely-coupled Web services distributed over the Internet. One of the resulting research problems is to model and execute composite Web services that are capable of adapting themselves effectively to handle business exceptions and to manage runtime faults (failures) and quality of service (QoS) degradations of one or more constituent Web services. Building such an adaptive composite Web service is challenging as there can be many types of business exceptions and runtime faults that can occur at anytime during execution [4, 5]. Hence, adding runtime adaptation capabilities to a composite Web service can require considerable design and development effort and it can become a source of complexity in the architecture and the implementation of composite Web services. The key challenge stems from the asynchronous nature and the uncertainty about when a business exception/fault could occur. For example, an order cancellation request could be received at any stage while the process is running. Furthermore, monitoring and adaptation aspects tend to be more volatile and subject to more frequent changes than the base process.

While adaptation is a very important problem and there have been several research projects in this area, such as [3], runtime adaptability is not yet adequately supported by dominant Web service composition languages, such as WSBPEL [7]. Additionally, corresponding orchestration engines do not provide sufficient support for detecting and handling the broad range of business exceptions or faults that may occur during the process execution. Moreover, the adaptation 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 cost.

The need for both powerful and flexible dynamic adaptation can be illustrated with the following example. Assume that complex business process is implemented as a Web service composition and that one of the Web services did not respond on time. When such fault (which could also be a manifestation of a performance problem) happens, a number of things can be done. For example, one could wait for the reply a bit longer, resend the request to this Web service (if it is idempotent), cancel this request, simply skip this request (and ignore its results if they come later), replace the faulty Web service with a known alternative, search for an alternative in a Web service directory and then use this replacement, etc. Only some, but not all, of these actions can be specified in WSBPEL. In addition, these options have different technical and business trade-offs, which cannot be specified in WSBPEL. Business value (quantifying and/or qualifying these tradeoffs) often depends on run-time context that is not predictable during design-time. In real-life scenarios decision which of the options to choose will not be pre-determined, but will depend on the business need to maximize various types of tangible (e.g., profit) and intangible (e.g., customer retention) business value.

---

** MASC project is sponsored by the Australian Research Council (ARC) and Microsoft Australia.

---

1 NICTA is funded through the Australian Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council (ARC).
To address the requirements for adaptive composite Web services, we propose a policy-based approach to runtime monitoring and adaptation to handle business exceptions, fault management, as well as process versioning and customization. The central part of the approach is our lightweight Web service management middleware MASC (Manageable and Adaptive Service Compositions) that performs runtime monitoring and adaptation. The key goal of MASC is to externalize and automate the detection of adaptation triggers and to execute the adaptation policies to ensure timely resolution of business exceptions and faults. 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. To achieve it, we formally specify monitoring and adaptation policies in WS-Policy4MASC [9], our novel extension of the Web Services Policy (WS-Policy) Framework [10]. Another goal of MASC is to achieve coordination of adaptation on the SOAP messaging layer and the business process orchestration layer, instead of focusing on only one of these layers. Further, we envisioned MASC as middleware that will provide innovative use of both technical and business metrics to achieve business value-driven adaptation decisions. Another distinctive characteristic of MASC is that it leverages and extends the power and flexibility of the recently released Microsoft .NET 3.0 platform [6]. This is important because architectural differences prevent directly reusing J2EE-based solutions in the .NET world.

This section provided an introduction to our research and summarized our motivating adaptation scenarios from a Stock Trading case study. The second section compares our research with related work. Then the paper elaborates MASC middleware architecture and its .NET implementation in Section 3 along with MASC middleware solutions for dynamic adaptation at the process layer. In Section 4 we present MASC middleware evaluation on a stock trading case study to illustrate the effectiveness of our approach while the last section summarizes conclusions and outlines our future work.

2. Comparisons with Related Work

Several ongoing academic and industrial efforts recognize the need to extend composite Web services middleware with mechanisms to provide dynamic adaptation. The closest related work to ours is the service monitoring approach presented in [1]. The authors proposed the Web Service Constraint Language (WS-CoL) for specifying client-side monitoring policies, particularly those related to security. At deployment time, WS-CoL constraints attached to a process are translated into WSBPEL invoke activities that call the Monitoring Manager to evaluate the monitoring policies and detect anomalous conditions. This approach is similar to ours in that monitoring policies are specified externally rather than being embedded into the process specification. It achieves the desired reusability and separation of concerns. However, it only provides support for monitoring and focuses mainly on security. On the other hand, our approach is more focused on adaptation (rather than just monitoring).

Another related work is [3], which suggested an aspect-oriented extension to WSBPEL to enable dynamic weaving of aspects into composite Web services to address QoS concerns such as security and reliable messaging. We find that our approach is more flexible and not limited to QoS aspects. Also the enforcement of MASC adaptation policies can be either delegated to the SOAP messaging middleware or enacted by the process orchestration engine via dynamic adaptation of running process instances. We also argue that policies offer better abstractions for specifying monitoring and adaptation needs than aspect advice.

Adaptability has been also researched in traditional workflow systems, such as eFlow [2] and others. However, most adaptive workflows approaches focused on modeling and formal verification, while our approach concentrates on the run-time execution of adaptation policies by the MASC middleware. Additionally, their solutions remain hard to integrate with recent composition languages and orchestration engines, as they are mainly based on proprietary graph or Petri net based models, making it non-trivial to map their suggested adaptation approaches to composite Web services.

To enable different types of adaptation of Web services compositions, MASC middleware significantly extends .NET Framework 3.0 [6], particularly the Windows Workflow Foundation (WF) and the Windows Communication Foundation (WCF). WF [6] provides an extensible framework for building processes (workflows) and embedding them into .NET applications to orchestrate activities of composed objects and services. In this respect, a WF process can represent a composite Web service. WF is capable of supporting multiple process specification standards by plugging-in a custom loader that transforms a process specification into an executable representation. However, at this time WF processes are defined only in Microsoft’s Extensible Applications Markup Language (XAML) and not WSBPEL. Translation between XAML and WSBPEL is promised for a future WF version. XAML addresses some of the limitations of WSBPEL by providing support for human workflows and sub-processes. To execute a process, WF has a lightweight runtime engine that manages the instantiation and execution of the process activities. Additionally, it manages different middleware concerns through an extensible set of runtime services, such as the built-in loading, scheduling, tracking, persistence and transaction support services. MASC middleware extends
WF with a set of runtime services for policy-based monitoring and adaptation of composite Web services implemented as WF processes.

3. MASC Middleware Architecture and Its Implementation

The conceptual architecture of MASC, capturing key components and their relationships, is shown in Figure 1. The MASC Middleware architecture incorporates both platform-independent components (that can be leveraged regardless of the SOAP messaging engine and the process orchestration engine used) and platform-specific ones. The latter execute the adaptation actions through transforming the WS-Policy4MASC assertions into either platform-specific commands (i.e., API calls to WF engine and custom extensions of both WF orchestration engine and the WCF SOAP messaging engine) or actions over messages exchanged between composed Web services.

3.1. Policy Specification, Run-Time Storage, Monitoring, and Decision

To be able to perform policy-driven management of Web services and their compositions, we needed a machine processable and precise format for declarative specification of policies for monitoring of functional and QoS aspects as well as specifying different types of adaptation. In particular, we wanted that this format could be used for automatic configuration of our MASC middleware. We have chosen the Web Services Policy Framework (WS-Policy) [10], an industrial specification standardized by the World Wide Web Consortium (W3C), as the basis for our policy specification. In WS-Policy, policies are collections of policy alternatives, which are collections of policy assertions. WS-PolicyAttachment defines a generic mechanism to associate a policy with subjects (e.g., WSDL elements) to which the policy applies. WS-Policy is a general and extensible framework for specification of policies for Web services and it has good properties in this respect. However, it does not contain detailed rules for specification of policies in particular areas, such as security, QoS monitoring, and adaptation. Specification of such detailed rules is left for WS-Policy extensions. Currently, only extensions for security, reliable messaging, and a few other management areas that were not the focus of our project had been published. Consequently, we had to develop a new WS-Policy extension, which we named WS-Policy4MASC.

WS-Policy4MASC is described in detail and illustrated with examples in [9], but this paragraph summarizes its main characteristics. It extends WS-Policy by defining XML schemas with new types of policy assertions. 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 1 second). They guide monitoring activities in MASC. Action policy assertions specify actions to be taken if certain conditions are met (e.g., some guarantees were not satisfied). For example, these actions can be removal, addition, replacement, skipping, or retrying of a sub-process (or individual activity) or process termination. They guide adaptation and other control actions in MASC. Utility policy assertions specify diverse business values (e.g., tangible vs. intangible, agreed payments vs. estimates) assigned to particular situations (e.g., execution of some action). They can be used by MASC for billing and for selection between alternative action policy assertions. Meta-policy assertions can be used to specify which action policy assertions are

Figure 1- MASC Middleware Architecture
alternative and which conflict resolution strategy (e.g., maximization of intangible profits) should be used. WS-Policy4MASC supports many different conflict resolution strategies, maximizing different business values. WS-Policy4MASC also 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 units). Within MASC, WS-Policy4MASC policy assertions are stored 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 SOAPMessageLoggingService and the QoSMeasurementService monitor the message exchanges between the composed Web services and collect monitoring information, such as values of message parts and measured QoS metrics. This information is stored in the Monitoring Database. Using this information, the MASCMonitoringService evaluates goal monitoring policies to detect adaptation triggers and events of interest. When they happen, the MASCMonitoringService generates an event (with all necessary information, such as process instance ID and monitored data values) to the MASCPolicyDecisionPoint. The latter determines adaptation action policy assertions to be applied and submits them to the MASCAdaptationCoordinator for execution. If there are several alternatives, then MASCPolicyDecisionPoint chooses the one to execute, based on WS-Policy4MASC meta-policy and utility policy assertions and built-in functions for maximization of business value. The MASCPolicyDecisionPoint is also responsible for evaluation of the pre-conditions and constraints associated with the adaptation actions to ensure the correctness of both the adaptation actions and the state of the adapted process. The MASCAdaptationCoordinator coordinates the execution of the adaptation actions between the SOAP messaging layer and the process orchestration layer. Management actions can adapt a single process instance, several instances of the same process schema, or instances of different process schema.

3.2. Supported Adaptation Actions

MASC middleware provides built-in support for common adaptation actions exposed through MASC’s management interfaces. We mainly concentrated on actions handling frequently occurring adaptation needs, such customization (incl. versioning) and corrective adaptation for fault management. The key adaptation actions supported by MASC are classified in three categories: process instance structural adaptation, process instance execution adaptation, and data flow adaptation.

**Process instance structural adaptation actions are:**
- **Add** an activity or a sub-process. This action takes as input the details to invoke the added service (such as the service’s endpoint address) or a search query to dynamically locate the service or the sub-process to be added. An activity can be added before, after or in parallel to a particular process activity.
- **Remove** an activity or an activity block. This action takes as input the identifier of activity or the activity block to be removed, in the form of a qualified activity name or an XPATH over the process definition. For removing an activity there must be no data dependencies between the removed activity and the successor activities otherwise data flow adaptation may become necessary.
- **Replace** an activity or an activity block. This allows replacing a service S by dynamically binding to another functionally equivalent service S’, potentially from a different company. This strategy requires the availability of replacement services having equivalent input messages, pre-conditions, output messages, and effects. This means that after any of the replaceable services is invoked, the process instance must be in the same state. Alternatively, faulty Web service can be replaced by a sub-process that has equivalent functionality as the replaced service. In most cases, some input/output message transformations are required to address syntactical, structural, and business protocol mismatches between services.

**Process instance execution adaptation includes** process-level control actions such as suspending, resuming or terminating a process instance. It also includes activity-level adaptation actions such as:
- **Cancel** an activity. This action stops execution of a running activity (or requests a cancellation of an asynchronous Web service request). For asynchronous service options, this adaptation operation requires that the invoked Web service provides a cancel operation. Often, there are temporal and/or cost constraints associated with cancelling a service. For example, a stock buying order in some stock trading systems can be cancelled only within the several hours after the order placement and this cancellation might also incur a fee. In some cases, cancelling a Web service invocation requires starting a separate sub-process that manages cancellation of all related running activities.
- **Skip** an activity. This action moves the process instance execution to the immediate next process activity. There must be no data dependencies between the skipped activity and the successor activities. This constraint can be alleviated by supplying some default values, expected by subsequent activities, from historical/cached values. A variation of this adaptation action is the **SkipTo** construct.
that allows specifying the next process activity. Another useful valuation is Skip and reschedule an activity viz. postponing an activity until further point in the process execution or until a particular time interval is passed. An example is postponing a customer notification activity until the end of the process execution.

- **Compensate** an activity. This action removes effects of a successfully completed activity. The compensation can take place either immediately after the activity completion or at an arbitrary point in the process instance. There might be temporal and/or cost constraints required by the service provider.

- **Retry** an activity. This action repeats invocation of a faulty Web service operation, according to a policy-defined retry pattern. For example, the adaptation policy can specify a retry of the service invocation three times with 1 minute delay between retries. This is useful for transient faults, such as a transient database deadlock or transient overloaded server. However, this strategy is only applicable to idempotent service operations where only one of multiple requests is executed or all are executed without state changes (such as getting a stock quote). One possible constraint is that retries might cause timeout in other paths of the process.

- **Delay** an activity. This action pauses a running activity until a particular event occurs, such as expiration of a timeout interval.

- **Modify** activity parameters. This action allows modifying the values of process/activity variables. An example could be to change the number of iterations for a loop.

All above adaptation actions are executed by the process orchestration layer. MASC also supports data flow adaptation actions at the SOAP messaging layer without an intervention of the process orchestration engine. These actions, discussed in our previous work [5], manipulate the exchanged messages through a set of composable adaptation actions, such as merging, splitting, transforming, enriching and buffering of messages. Additionally, we are exploring policy adaptation with activation, deactivation, addition and deletion of policy assertions. We plan to also provide support for middleware configuration adaptation, such as changing of service bindings, transport protocols, or parameters of used WS-* protocols.

In addition to a description of actions to be executed as a group, a WS-Policy4MASC action policy assertion also contains information about events that trigger execution of these actions and, optionally, filtering conditions. If more than one MASC action policy assertion is triggered at the same time, meta-policies are used to determine which actions to execute. For example, WS-Policy4MASC utility policies can specify that skipping of some desirable but non-essential faulty activity within some business process does not incur any immediate tangible cost, but in the long run reduces customer satisfaction, which results in intangible loses estimated at $200 per year. On the other hand, replacing this faulty activity with a more expensive alternative activity (e.g., Web service) provided by a different vendor brings immediate tangible loss of $30 to pay penalties for breaking QoS constraints (since the replacement process requires time), but no long-term intangible losses. If the meta-policy conflict resolution strategy is to maximize only tangible business values, the MASCPolicyDecisionPoint will choose skipping this activity, but if the conflict resolution strategy also considers intangible business values, then the faulty
activity will be replaced. Note that several actions can be specified for execution as a group, many different types of business values can be specified in utility policies, and conflict resolution strategies can contain tiebreaker rules for comparing similar amounts. This all makes that the MACPolicyDecisionPoint has more complicated task than in the above simple example. The business value-driven selection of adaptation actions is an important unique characteristic of our work. While some additional explanations are given in [9], we are preparing a separate publication on this topic.

3.3. Components for Adaptation at the Process Orchestrator Layer

The sequence diagram in Figure 2 depicts interactions between key MASC middleware components to execute dynamic adaptation actions. The adaptation at the process orchestration layer is managed by the MACAdaptationService. It is implemented as a WF runtime service and it exposes a set of management interfaces that abstract interactions between the MASC’s decision making components and the WF runtime. The MACAdaptationService’s management interfaces correspond to the process-level adaptation actions discussed in the previous section. Their implementation maps those actions to WF extensions commands and API calls to .NET 3.0 libraries. When a process instance is created, MASC middleware intercepts this event and uses its custom MACProcessLoader to modify the created process instance by traversing the base process activity tree and wrapping activities, enclosed in each composite activity, into a MACActivity (here “MAC” stands for “MACAdaptationControl”) to make the process instance adaptation-ready (while preserving the process and the enclosed activities). Composite activities can be iterative control flow constructs such as while loops, conditional control flow constructs such as if-then-else, constructs for parallel execution of activities or custom composite activities. The process instance activities run within a MACActivity which acts as a runtime container responsible for listening for and executing dynamic adaptation actions for both customization and fault management. The key functions of MACActivity are:

- Enable alternative process path to be followed. MACActivity acts as a navigator with the capability of arbitrary redirecting execution to any child activity positioned either before or after the currently executing activity. As a navigator, the MACActivity maintains a history of all activities previously executed and allows going back to particular process point and re-executing particular activities.
- Start the process execution at any child activity. When the process instance starts, the MACAdaptationService can configure the MACActivity to start the process execution at any child activity by enquiring a StartWith adaptation action that names the child activity that will be the first to execute. This pattern allows controlling the first activity to execute while avoiding expensive structure changes to the running process instance.
- Dynamically call a sub-process, such as an order cancellation or compensation sub-process.
- Apply structural changes to a running process instance. First, the MACAdaptationService asks the WF runtime engine for a description of the process to be adapted and gets back a transient copy of the process’ object representation. For this copy, MACAdaptationService performs the changes specified in the policies, using primitives built into the WF runtime. If data exchange is required between the base process and the variation processes/activities, this service also takes care of required parameters binding and value passing between base process and the added process variation. Then, MACAdaptationService passes the modified copy of the process to the MACActivity, the latter suspends the running process instance, applies the changes and resumes the execution of the adapted process instance.
- Rearrange the process activities without incurring the performance penalty of the dynamic changes of the process instance structure. For examples, if a composite activity defines a sequence of activities in the following order (A-B-C) the MACAdaptationService could instruct the MACActivity to execute the sequence as (C-A-B). Such change is only allowed if these activities have no data dependencies between them.

The implementation of the MACActivity relies on WF API to create and manage (subordinate) execution contexts found in the ActivityExecutionContextManager type, which we refer to as AECManager. The capabilities of AECManager are the basis for a wide range of MASC adaptation actions, ranging from the familiar activity retry pattern to more advanced patterns including activity compensation and changing the sequence of the process instance activities.

Communication between MACAdaptationService and MACActivity relies on the Queuing Service built into WF. The Queuing Service facilitates communications between the process instances and the outside world. When the MACActivity is initialized (this happens when the process instance is started), it registers a queue with the WF Queuing Service to listen for adaptation requests. The MACActivity subscribes for notifications from this adaptation listener queue. During the execution of the base process, when an adaptation trigger is detected by the MASCMonitoringService, the MACAdaptationService en-queues adaptation actions (decided by MACPolicyDecletionPoint), for a particular process instance. The listener queue notifies the MACActivity by raising an event. Then, the control is passed to the MACActivity, which dequeues the requested adaptation actions and executes them through
the invocation of MASC management interfaces. In the case when the destination process instance is unloaded from memory (e.g., persisted when idle or waiting for external event), the WF runtime reloads the process instance and delivers the incoming adaptation request.

4. Functional and Performance Evaluation of MASC Middleware

To evaluate the functionality and performance overheads of MASC middleware components, we have conducted several experiments that dynamically adapt the base business process for national stock trading to support international stock trading. The need for dynamic adaptation 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 key considered scenario is the Order Placement process (exposed via the PlaceOrder operation of the FundManagerService), which is initiated when an investor places an investment or redemption order with their FundManagerService. The latter invokes the FinancialAnalysisService to get a recommendation about which stocks to buy/sell. Then, the FundManagerService sends the buying/selling request to the StockMarketService, which performs trade matching between the buy orders and the sell orders. When a trade match is formed, it invokes in parallel the StockRegistryService to transfer the shares ownership and the PaymentService to transfer funds. An investor can request their Account Balance, by invoking the GetAccountBalance operation of the FundManagerService, which in turn composes the GetMarketValueOfPortfolio from the StockRegistryService and the GetCashBalance operation from FundManagerService.

The conducted experiments were successfully achieved and demonstrated the feasibility and the usefulness of the MASC approach in adding dynamic adaptation capabilities to existing Web services compositions, guided by declarative policies specified in WS-Policy4MASC. MASC has provided a solution for policy-based static and dynamic adaptation without any changes to the base process definition, implementations of the used Web services, or the implementation of .NET 3.0 technologies. All that is needed is a WS-Policy4MASC document describing monitoring and adaptation policies to be enforced. When a WS-Policy4MASC document changes, these changes are automatically enforced the next time adaptation is needed, with no need to restart any software component. The evaluation scenarios were further extended to evaluate the effectiveness of MASC corrective adaptation. We periodically injected random exception events at various stages of the stock trading process to study the behavior of the system in response to faults or QoS changes of constituent services. For service faults, we randomly picked some of the available services and made them unavailable for a random amount of time. For service QoS degradation, we periodically picked some service instances and introduced a random number of seconds delay before sending a reply back to the process. The results of the analysis from 1000 repetitions show improved process reliability since 97% of process faults were handled effectively by MASC and the process was able to resume execution past the point of failure. Only a small number of exception scenarios could not be managed by the framework and thus required human intervention, such as the absence of a replacement Web service to substitute a faulty service. Under identical conditions, disabling MASC mediation resulted in a process termination whenever faults were injected.

To evaluate the performance of MASC middleware, we conducted a series of benchmarking tests to measure the overhead introduced by each component. Our secondary aim for these tests was to discover areas of the platform that needed further improvements. We used the GetAccountBalance Web services composition from the Stock Trading case study. Our experiments measured the average process execution using 1000 runs with and without adaptation. Note that the evaluation runs used exactly same implementation and runtime environment and for all tests we ignored the first 50 process executions to allow some 'warm-up' time. The performance results were averages of measurements taken from 10 repetitions of experiments executed during different times of the day to minimize impact of background processes. In summary, with dynamic adaptation using MASC middleware, the process execution was about 20% slower than without adaptation. The dynamic adaptation from within the process performed worst, with an average execution duration almost two times slower than no adaptation. This is due to the fact that the dynamic process update from within the process has to get the running process definition, modify it and apply the changes only when the execution reaches the adaptation point. On the other hand, in the case of external adaptation, the MASCAdaptationService gets the running
process definition, modifies it and queues up the changes while the process is running. The MACActivity only applies the pending adaptations (without wasting any time on modifying the process definition). Note that MASC middleware uses external adaptation techniques and internal adaptation was only used to compare performance overheads of the two adaptation modes. Figure 3 depicts the results of the discussed performance comparisons.

The main reason for the performance overheads of the dynamic adaptation support in MASC is that it has to suspend and then resume execution of the process. Also, the overhead of parsing WS-Policy4MASC policies is about 20% of the total overhead. We plan to implement caching mechanisms to offset the performance degradations introduced by policy parsing. Note that the addition of MACActivity to control the process execution resulted in negligible performance increases.

5. Conclusions and Future Work

Dynamic adaptation of composite Web services is necessary because change is very frequent both in businesses that use these composite Web services and in computing infrastructures that implement them. Many diverse changes can happen, so the range of adaptation actions that can be undertaken is broad. While there has been a considerable body of research on related issues, such as adaptive workflows, dynamic adaptability is not yet completely addressed by Web service middleware.

The objective of the MASC middleware is to enhance the ability of composite Web services to autonomously (with minimal human intervention) adapt to various changes to provide uninterrupted service, satisfaction of quality of service (QoS) guarantees, and maximal business value. MASC builds upon the power and extensibility of the new Microsoft .NET 3.0 platform. Another advantage of MASC is that it uses human-understandable 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 policies handling exceptional aspects. This results in improved flexibility and reusability. Next, MASC was designed to address several types of adaptation, such as customization, fault management, performance optimization, and proactive adaptation. Further, MASC coordinates enforcement of run-time adaptation decisions across both the business process orchestration layer and the SOAP messaging layer. Last, but not the least, MASC uses both technical and business metrics for adaptation decisions. For example, selection of alternative adaptation policies is guided by maximization of diverse types of business value specified in utility policies. We have been implementing a MASC proof-of-concept prototype and evaluating it on case studies, particularly, stock trading scenarios. Our performance studies conducted with the prototype indicate that overheads introduced by MASC are acceptable.

Our approach has unique characteristics. We adopt a policy-based approach that builds on the established policy-based management principles [8], while decoupling between sensors that monitor and detect adaptation triggers and effectors that react to and handle such triggers. Additionally, our technological base is different which leads to different architectural solutions. Our proposed approach extends .NET 3.0 (which has not been previously studied in this area) and WS-Policy. Our middleware performs different types of adaptation and coordinates adaptation actions at different Web services middleware layers. Furthermore, the ultimate goal of MASC is business-driven adaptation of composite Web services, while related works mainly aim at improving technical metrics related to computational efficiency.

For future evaluation work we plan studies of scalability and sensitivity of the MASC middleware to various parameters, the number of deployed policies, the number of concurrent process instances, and the number of activities within an adapted composite Web service.

References