Policy-Driven Middleware for Self-adaptation of Web Services Compositions

Abdelkarim Erradi[1], Piyush Maheshwari[1,2], and Vladimir Tosic[1,3]

[1] Shool of Computer Science and Engineering, The University of New South Wales, Sydney, Australia
[2] IBM India Research Lab, New Delhi, India
[3] Department of Computer Science, The University of Western Ontario, Canada
aerradi@cse.unsw.edu.au, pimahesh@in.ibm.com, vladat@computer.org

Abstract. We present our policy-based middleware, called Manageable and Adaptive Service Compositions (MASC), for dynamic self-adaptation of Web services compositions to various changes. MASC integrates and extends our earlier middleware called the Web Services Message Bus (wsBus). In particular, we discuss MASC support for customization of Web services compositions to address business exceptions and wsBus support for correction (fault management) of Web services compositions to improve reliability. We have evaluated the former support on a stock trading case study and the latter support on a supply chain management case study. Our solutions are complementary to the existing approaches and provide: coordination of fault management between SOAP messaging and business process orchestration, greater diversity of monitoring and control constructs, specification of both technical and business aspects used for adaptation decisions, higher level of abstraction easier for use by non-technical people, and externalization of monitoring and adaptation actions from definitions of business processes.

Keywords: Web services middleware, Web services composition, policy-based management and adaptation, Microsoft .NET.

1 Introduction and Motivation

Web services compositions (orchestrations and choreographies) are rapidly becoming a dominant approach for implementing business processes and building open distributed systems. The widely accepted Web services technologies (the Web Services Description Language – WSDL, SOAP, and the Universal Description, Discovery, and Integration – UDDI) are not enough for implementing Web services compositions [4]. Several languages for describing Web services compositions have appeared and the Web Services Business Process Execution Language (WSBPEL or BPEL) is the most widely accepted among them. A number of additional technologies (often named 'WS-*') have been developed to address requirements such as security, reliable messaging and transactional service coordination. However, a number of important issues are not completely solved. Many of them are related to building more powerful

M. van Steen and M. Henning (Eds.): Middleware 2006, LNCS 4290, pp. 62 – 80, 2006.
© IFIP International Federation for Information Processing 2006

middleware to support creation, execution, and management of Web services compositions. Such an important research question, discussed in this paper, is how to build more powerful middleware to enable autonomous self-adaptation of Web services compositions to various runtime changes.

In preparation for this research, we had studied different types of adaptations of Web services composition and decided to classify them based on 3 dimensions, each orthogonal to the other 2. The first dimension is whether the complete class of compositions (e.g., an abstract process in BPEL) is changed or whether only a particular composition instance is changed. In this paper, we focus on the latter, because the need for such adaptations is much more frequent. The second dimension is the relative time when a Web services composition instance is changed. Adaptation is *static* when a composition instance is changed before it is started, while it is *dynamic* when a running composition instance is changed without being stopped and restarted from the beginning. In this paper, we focus on dynamic adaptation, because it is much more challenging. The third dimension describes the reason why the adaptation is done, which impacts how the adaptation is done. On this dimension, adaptation can be: a) *customization* – to add/remove/replace activities specific to the composition instance (but not to the complete class of compositions); b) *correction* – to handle faults reported during execution of this composition instance; c) *optimization* – to improve extra-functional (usually performance or billing) issues noticed during correct execution of this instance; or d) *prevention* – to prevent future faults or extra-functional issues before they occur. This classification is similar to the classification of software evolution into adaptive, corrective, perfective, and preventive [17]. In this paper we focus on customization and correction. While we have some results related to optimization, they will be discussed only in relationship to using corrective adaptation (i.e., fault management) to improve reliability of Web services compositions. A long-term goal of our research is to study and enable all identified adaptation types.

Special cases ('business exceptions') can occur relatively frequently in business processes. Such a special case has almost all activities as in a regularly occurring base business process, but some activities are removed or replaced and/or new activities are added. An example is when a company has set up a complex business process for domestic business partners (e.g., within one country), but an unexpected request comes to set up a version of this business process for some international partners with additional activities to handle payment in multiple currencies. This special case can be addressed with customization of the base business process for domestic partners. Such a customization can be performed in different ways. One way is to add into the description of the base business process (e.g., in BPEL) appropriate new exceptions, event handling constructs (e.g., timeouts), compensation activities, and/or message correlation. While this is a simple and straightforward approach, it has several drawbacks, which reduce its applicability to advanced scenarios. The most important drawbacks are that (1) it enables only static and not dynamic customization (i.e., change of running process instance), and that (2) it cannot be applied in cases when the base business process is defined by a standardization body and its description cannot be changed easily. The latter drawback can be addressed if the base process description is copied and then manually changed into a description of a new business process. However, this approach also does not address dynamic customization. In addition, it significantly reduces maintainability because if a change in the base

process occurs, descriptions of all customized processes have to be updated manually. When dynamic customization is needed, it is usually advantageous to externalize descriptions of specifics of individual cases from the description of the base process. This simplifies development, composition, and management activities (and corresponding software) and fosters reuse. Such separation of concerns is used frequently in software engineering, e.g., in aspect-oriented programming, and distributed systems and network management, particularly policy-based management [12].

Additionally, various faults can occur relatively often and unexpectedly in distributed systems. For example, remote computers can be down or unavailable (e.g., due to denial of service attacks), network links can be congested or broken, or remote applications can produce unexpected results due to semantic misunderstandings. In Web services compositions, the diversity of possible faults is particularly high because implementations of Web services have to be treated as 'black boxes', participants in business-to-business (B2B) interactions usually relinquish no or very little control to other participants, and SOAP communication mostly uses unmanaged Internet infrastructure. On the other hand, Web services compositions often implement business-critical processes whose correct and uninterrupted operation is paramount. Therefore, to achieve dependable business processes, Web services compositions have to be made reliable. Reliability can be defined as the continuity of correct service delivery. This implies zero or, at worst, relatively few failures and rapid recovery time. Reliability of Web services compositions is a complex and challenging task that has to be addressed at several layers: service provider layer (e.g., service hosting containers), transport layer, SOAP messaging layer, and business process layer. Some reliability aspects (e.g., invocation retries) can be solved at different layers with different trade-offs, but some reliability aspects are best solved only at one particular layer (e.g., influences of dependencies between activities on the reliability of the whole process can be determined only at the business process layer). In our approach, events can trigger cross-layer adaptation that could span both the process layer and the messaging layer. Among the advantages of the adaptation at the messaging layer is the potential reusability across process instances and process types. In particular, executing faults handling policies at the messaging layer shields faults from the process orchestration.

During the last several years, a number of academic papers (e.g., [13]), industrial standardization efforts (e.g., WS-ReliableMessaging, WS-Reliability, WS-Transaction), and industrial products have made contributions to improving reliability at different layers. However, they have limitations, particularly in the diversity of events (e.g., QoS degradations that cause faults) that they can monitor and handle, customizability and diversity of actions (apart from rollback and compensation) that they can perform in different contexts, specification of technical (e.g., performance, security) and business benefits/costs of particular actions, and cross-layer integration of reliability solutions at different layers (e.g., retries considered only at the SOAP messaging layer could cause business process timeout). One of the recent research trends to address reliability issues is augmenting Web services middleware with autonomous behavior capabilities such as self-healing and self-configuring [15]. Our work belongs to this emerging direction.

Policies can be used for representation of all types of adaptation and monitoring activities. The term 'policy' is used in different ways in the literature. A general definition is that a policy is a declarative, high-level description of goals to be achieved and

actions to be taken in different situations. There are different types of policies, but in this paper we focus on Event-Condition-Action (ECA) rules [9]. Such a rule specifies a triggering event (e.g., arrival of a message, start of a process instance, runtime fault, or performance problem), additional conditions to be satisfied (e.g., referring to process state or history), and actions to be taken (e.g., change of a process instance) when the event occurs and the conditions are satisfied. The main advantage of policies over alternatives (e.g., aspect-oriented programming) is that policies are higher-level abstractions, so humans (e.g., business analysts) can specify them more easily.

In this paper, we present our work on policy-based middleware, called Manageable and Adaptive Service Compositions (MASC) (http://masc.web.cse.unsw.edu.au), for dynamic self-adaptation of Web services compositions to various changes. While some of our previous publications, particularly [6], also discuss some aspects of our work in this area, this paper complements them by providing both an overall picture of our research and additional technical details about our recent solutions. MASC is an evolution of our previous research of middleware for Web services. It integrates and extends our previous middleware-related projects, the Web Services Message Bus (wsBus) [5] and AdaptiveBPEL [7]. In addition, we have performed a technology switch – while our previous projects were built with Java-based technologies, the new implementation of MASC is based on the novel Microsoft .NET Framework 3.0 technologies and C#. An important aspect of our work on the MASC middleware is that we aim to provide policy-based adaptation (particularly optimization and prevention) based on maximizing business metrics (e.g., profit). This complements current works on dynamic adaptation of Web services compositions, which mostly focus on maximizing technical QoS metrics (e.g., throughput), but rarely ([11]) study business metrics in detail.

This section provided an introduction to our research and summarized our motivation. The second section presents MASC middleware solutions for customization of Web services compositions. We elaborate our .NET-based architecture and implementation and explain their evaluation on a stock trading case study. The third section presents our middleware solutions for corrective adaptation of Web services compositions to improve their reliability. We discuss our Java-based architecture and implementation of wsBus and its evaluation on a supply chain management case study. wsBus is now a part of the MASC project, so their relationships are discussed in the third section. The fourth section compares our research with related work, while the last section summarizes conclusions and outlines our future work.

2 Middleware for Policy-Based Customization

To be able to perform policy-based management, it is necessary to define an appropriate machine processeable and precise format for policy specification. We have been developing a novel XML (Extensible Markup Language) format, called WS-Policy4MASC. Its goal is to enable specification of policies for monitoring of functional and QoS aspects (such as performance and reliability) and different types of adaptation for Web services and their compositions, in a way that can be used for automatic configuration of our MASC middleware presented in this section. Our language is an extension of the Web Services Policy Framework (WS-Policy) [16], an

industrial specification standardized by the World Wide Web Consortium (W3C). In WS-Policy, policies are collections of policy alternatives, which are collections of policy assertions. WS-Policy Attachment 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 very 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. Unfortunately, only extensions for security, reliable messaging, and a few other management areas that are not the focus of our project have been suggested. Therefore, we had to develop a new WS-Policy extension for use in our middleware. WS-Policy4MASC is also compatible with other Web services standards such as WSDL and BPEL, as well as Microsoft .NET 3.0 technologies. Since our MASC middleware has ambitious goals in several areas, WS-Policy4MASC offers a number of constructs for powerful and precise policy specification. Details and examples of the WS-Policy4MASC expressive power and syntax will be given in a future publication. We only provide here a short overview of the current support for customization policies.

An adaptation (including customization) policy in the current version of WS-Policy4MASC can define events which cause its evaluation, optional conditions on its relevance (e.g., a policy may be relevant only in particular contexts), a state in which the adapted system (e.g., a Web services composition) should be before the adaptation, additional conditions on the adapted system (e.g., historical values of QoS metrics), a set of actions to be taken if all previous conditions are met, a state in which the system will be after the adaptation, and change of business value (e.g., monetary payments) associated with this adaptation. The basic adaptation actions include removal, addition, and replacement. In removal and replacement, an activity or an activity block in a base business process is deleted. All business processes, including base processes and variation processes, are defined in appropriate other documents (e.g., BPEL files), so they are only referenced in WS-Policy4MASC policies. Thus, an activity block is specified using beginning and ending points. In addition and replacement, a new variation process or a single activity is inserted into a particular point in a base process. If the inserted single activity is a Web services call, the policy can specify a particular Web service or a set of criteria for dynamically selecting the best Web service from a directory. Data exchange (i.e., required parameters binding and value passing) between a base process and a variation process/activity is also described.

2.1 Architecture of MASC Support for Customization and Its Implementation

To enable different types of adaptation of Web services compositions, we have been developing the MASC middleware. It extends the new Microsoft .NET Framework 3.0 (currently in pre-release - http://www.netfx3.com/), particularly its components the Windows Communication Foundation (WCF) and the Windows Workflow Foundation (WF). For the MASC solutions presented in this section, the extensions of WF are more important. WF provides an extensible framework for building processes

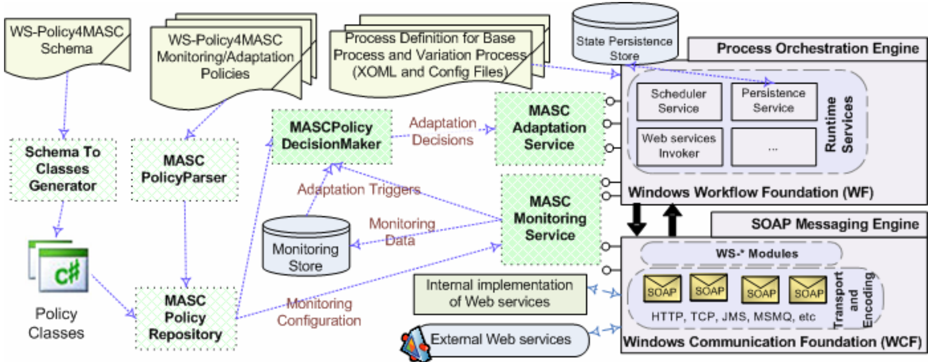

Fig. 1. Architecture of MASC support for customization of Web services compositions

(workflows) and embedding them into .NET applications to orchestrate activities of objects and services. In this respect, a WF process can represent a Web services composition (orchestration). WF processes are defined in Microsoft's Extensible Applications Markup Language (XAML, but file extension for WF is '.xoml') and not BPEL. Translation between XAML and BPEL is promised for a future version. The glue code for connecting activities, such as activity input validation, can be encapsulated into a 'code beside' .NET class. To execute a process, WF has a lightweight WF runtime engine that can be hosted in any .NET application. The WF runtime engine manages the instantiation and execution of the 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 MASCAdaptationService, for policy-based adaptation of Web services compositions implemented as WF processes. It currently enables static and dynamic customization, while its future version will provide support for static and dynamic corrective, optimizing, and preventive adaptation based on maximizing business metrics. The support for dynamic adaptation means that MASCAdaptationService can use policies to change a running process instance without any changes to process definition or implementation of activities (e.g., composed Web services). The WF runtime engine can be configured to include MASCAdaptationService and support its operation. MASC is a complex middleware with many modules (some of which are not yet implemented). For readability purposes, we will describe in this section only MASC support for customization. The overall architecture of MASC will be given in another publication.

The conceptual architecture of the MASC support for policy-based customization is shown in Figure 1. We have implemented its prototype in C#. Monitoring and adaptation policy assertions are stored in a policy repository, which is a collection of instances of policy classes. The policy classes are generated automatically from the WS-Policy4MASC schema, using an XML-schema-to-C#-classes generator (in our case, the XSD tool from .NET). When the MASCAdaptationService starts, our MASCPolicyParser imports WS-Policy4MASC files, creates instances of corresponding policy classes, and stores these instances in the policy repository. Static customization

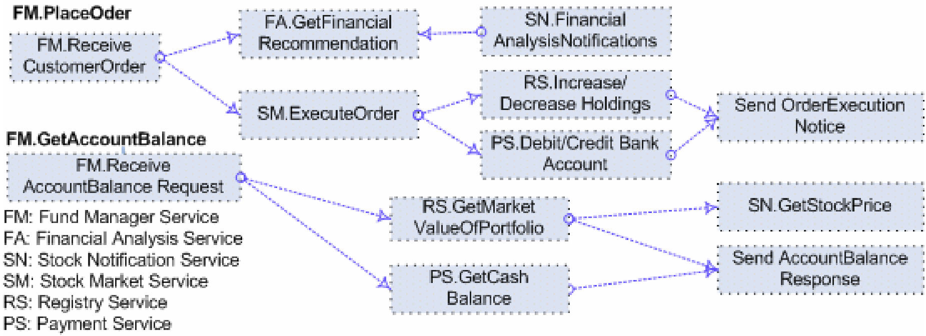

Fig. 2. Example Web services interactions in the Stock Trading case study

is started when the WF runtime raises an event that a process instance is created. Dynamic customization is started when the MASCMonitoringService module raises an event that for a particular process instance it detected (e.g., by introspecting exchanged SOAP messages and/or measuring QoS metrics such as response time) adaptation pre-conditions specified in monitoring policies. Such events can also be raised by the MonitoringStore database in situations when adaptation pre-conditions refer to several different SOAP messages. For both static and dynamic adaptation, the raised events are handled by MASCPolicyDecisionMaker, which determines adaptation policy assertions to be applied to the process instance and sends an event to MASCAdaptationService. Policy priorities are used to determine the order of execution if several policy assertions apply per event. In case of dynamic adaptation, MASCAdaptationService suspends the running process instance to be adapted. Then, it 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, MASCAdaptationService 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, our service also takes care of required parameters binding and value passing between base processes and their variation processes. When MASCAdaptationService passes the modified copy of the process' object representation back to the WF runtime, the latter applies the changes using built-in algorithms. After this, the execution of the adapted process instance is resumed.

2.2 MASC Evaluation on the Stock Trading Case Study

The MASC support for customization has been evaluated and demonstrated in various adaptation scenarios using a simplified Stock Trading case study implemented with C#, .NET 3.0, and MASC. Parts of this case study are shown in Figure 2. 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, quotes, and other information. 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. (In our prototype, this decision is very simple, e.g., buy one best stock or sell as many poorly performing stocks as needed to get the redeemed money.) 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. Note that, with the exception of the FundManagerService, there can be multiple different services of the same type in the composition. For example, there can be more than one FinancialAnalysisService, e.g., provided by different vendors and/or performing different types of financial analyses.

To evaluate MASC's static and dynamic adaptation capabilities, we have conducted several experiments to customize the base business process for national stock trading, described above, to support international stock trading. WS-Policy4MASC was used for policy description. Among the conducted experiments was dynamic addition of a CurrencyConversion Web service (CC_1, CC_2...CC_n) to convert stock prices of foreign stocks to a local currency. Also, depending on the country of foreign stock, a PESTAnalysis Web service (PS_1, PS_2...PS_n) was added to assess the non-financial aspects (political, economic, social and technology) that influence the trade. Additionally, monitoring policies were used to define constraints over the trade transaction amount and/or the customer's profile (e.g., personal investor vs. corporate investor) to dynamically add a CreditRating Web service (CR_1, CR_2...CR_n) before processing the trade. In terms of removing activities, we have experimented with dynamic removal of the invocation of MarketComplianceService when the trade amount is less than a particular threshold. The conducted experiments were successful and demonstrated feasibility and usefulness of the MASC approach in adding dynamic customization 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 customization without any changes to either the process definition or the constituent services implementations. 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 above scenarios will be further extended to evaluate MASC and WS-Policy4MASC support for corrective, optimizing, and preventive optimization, once they are completed.

3 Middleware for Policy-Based Corrective Adaptation

Our work addresses reliability at the business process layer and the SOAP messaging layer by specifying and enforcing monitoring policies to help in fault detection and corrective adaptation policies to guide fault correction. It is complementary to the

existing approaches and provides: (1) coordination of fault handling across these two layers, (2) greater diversity of monitoring and control constructs, (3) specification of both technical and business aspects that can be used for adaptation decisions, (4) higher level of abstraction easier for use by non-technical people, and (5) externalization of monitoring and adaptation actions from definitions of business processes.

Our main past project in the area of reliability of Web services compositions was the wsBus middleware built using Java-based technologies and an early version of our WS-Policy extensions in this area (the name 'WS-Policy4MASC' was not used at that time). As mentioned in the introductory section, our focus has recently shifted towards the more general MASC middleware built upon .NET technologies. WS-Policy4MASC grammar was also updated. We have been working on integrating wsBus solutions with other parts of MASC, including .NET and C# reimplementation and support for the new WS-Policy4MASC grammar. However, since our results are still more complete for the Java-based implementation of wsBus, we will describe it in this paper and leave discussion of recent improvement for another publication.

Policies that can be enforced by the Java-based version of wsBus are specified in a WS-Policy extension described and illustrated in [6]. The main types of actions in these policies are: invocation retries, Web services substitution, concurrent invocation of multiple equivalent services, skipping of activities, and relatively simple dynamic changes of process instances (e.g., add/remove/skip an activity, change sequence of activities, delay/suspend/resume/terminate process). Only the latter is at the business process layer, while the others are at the SOAP messaging layer. In this way, they complement the policies described in the previous section, which are all at the business process layer.

3.1 Architecture of wsBus and Its Implementation

This section presents the architecture of wsBus with emphasis on the modules that facilitate the enactment of adaptation policies. As shown in Figure 3, adaptation policies supported by wsBus work via injecting runtime inspectors and custom Message Processing Modules into a messaging pipeline at different message processing stages such as before sending a request and after receiving a response. These custom modules can be applied at different scopes such as the whole service, a particular endpoint or a particular service operation. For example, the Invocation Retry Handler places the messages that fail to be delivered in a retry queue and the queue reader tries redelivery using the pattern specified by the used recovery policy. Messages for which processing repeatedly fails are placed in a 'dead letter' queue after exhausting the maximum number of allowed retries and no further delivery will be attempted.

wsBus key architectural abstraction is the concept of a Virtual End Point (VEP). A VEP allows virtualization by grouping a set of functionally equivalent services and exposes an abstract WSDL for accessing the configured services (e.g., Web search service exposing Google, Yahoo and MSN search as one virtual service). The grouped services might have different QoS properties. The VEP acts as a recovery block and various runtime policies can be associating with it.

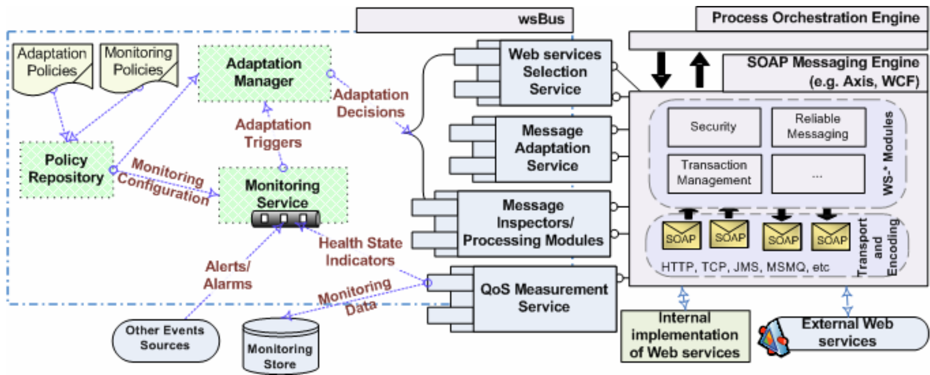

Fig. 3. wsBus Architecture

wsBus can be deployed either as a gateway to a Process Orchestration Engine or it can act as a transparent HTTP Proxy. In the first case the Process Orchestration Engine should be configured to explicitly direct service calls to the virtual endpoints configured in wsBus and the later routes request messages to the real services. The VEP takes care of the dynamic Find, Select, Bind and Invoke on behalf of the BPEL engine, using the configured selection and binding policies. The VEP does 'on the fly' selection of service provider or intermediary based on a selection criteria specified in the policy attached to the VEP, such as message content and context (e.g., requester profile), or the service provider's capabilities or QoS of prior interactions. The VEP then manages the automatic enforcement of adaptation policies (e.g., retry and substitute policies) by inspecting messages going into and out of the composed services and interposing additional Message Processing Modules along the message pipeline. To decide the relevant Message Processing Modules applicable to a given message, the VEP uses simple rules expressed as a regular expression or XPath query against the header or the payload of the message. Additionally, the VEP provides middleware services to service compositions such as QoS measurement and monitoring, conversation management and fault management. Our fault management approach is based on two models: (1) the capturing model uses assertion-based monitoring to detect faults and to notify the relevant middleware component, and (2) the handling model uses adaptation policies represents to resolve faults. For example, a policy might stipulate that for particular type of faults, the VEP should retry to the original service and if the fault persists then it should select an equivalent backup service.

The enactment of adaptation policies is managed by the following key components:

1) *QoS Measurement Service* is responsible for management data collection and analysis either through direct computation of QoS metrics (e.g., collecting statistical metrics about the performance) or via periodic probing for management information from other management intermediaries (e.g., third QoS measurement entity). The key QoS metrics measured by this component are: (a) Reliability (calculated as a ratio of successful invocations over the number of total invocations in given period of time); (b) Response Time (the time interval between when a service is requested and when it is delivered; (c) Availability: the percentage of time that a service is available during

some time interval. Because the lack of space, the QoS measurement algorithms are not presented is this paper.

2) *Monitoring service* continuously monitors interactions with the participating services to verify that the configured monitoring policies are being satisfied and to detect any condition changes such as faults. The monitoring policies specify the desired behavior of the system in terms of (a) pre-conditions and post-conditions that express constraints over exchanged messages (b) thresholds over QoS guarantees (e.g. service response time) as stipulated in pre-established Service Level Agreements (SLAs). The monitoring policies can be attached to Monitoring Points at various levels of granularity such as a Service Endpoint or a Service Operation. For example, the monitoring policies could specify that exchanged messages between participant services must be validated to ensure conformance to the service contract expected by the service composition. The Monitoring Service also supports events-based monitoring to detect fault events and recognize their type. Various techniques are used to achieve this. First, the Monitoring Service listens to fault messages returned by invoked services as specified in their WSDL interface. Faults can also be identified based on management events coming from internal or external management systems, such as hardware or network failure faults. Also, the Web services Invoker component can use timers to raise timeout faults when the service does not respond within the set timeout interval.

The monitoring policy uses XPath to reference variables defined in the header or the body of the WSDL contract of constituent services (e.g., the CustomerID of PurchaseOrder message). During the evaluation of the monitoring assertions, the Monitoring Service might reference data from external sources to obtain data not available in the exchange messages. The source of such external data as specified as Web service calls in the monitoring assertions, such as calling a QoS measurement service or querying the log of prior interactions to get some historical data.

When an undesirable condition is detected, then the Monitoring service uses ECA rules to assign a meaningful fault type to the violation event, such as Service Unavailable Fault, SLA Violation Fault, Service Failure Fault and Timeout Fault. The fault is then passed to the Adaptation Manager along with all the data required for recovery (i.e., ProcessInstanceID of the process instance to be adapted, and a Context Collection that contains relevant data that could be needed during the adaptation.)

3) *Adaptation Manager* decides and coordinates the execution of appropriate adaptation action(s) to restore the system to an acceptable state using adaptation policies configured at the VEP. Currently our adaptation policies use a rule-based approach to specify the necessary adaptations per fault type. Such a rule-based approach is more flexible as it can handle wider variety of faults whether coming from the infrastructure or from the partner services. Also the process specification is kept simple and uncluttered through the separation of the process logic and fault handling policies. The adaptation action could be simple or composite. It could be specified to be enacted either at the SOAP messaging layer (such as retry a service call) or at the process orchestration layer (such as skip a process activity or add/remove activity) or sometimes at both layers. For example, before retrying invocation of a faulty service, the adaptation policy might stipulate that MASCAdaptationService should first suspend the calling process instance (until the execution of the adaptation actions is completed) or increase its timeout interval to avoid the calling process timing out. To

be able to decide the process instance to be adapted, MASCAdaptationService transparently adds the ProcessInstanceID of the calling process to outgoing SOAP messages (using the RelatesTo Message Addressing Header). When multiple adaptation policies are specified per fault type, policy priorities are used to determine the order of execution of the adaptation actions. For example, a policy could stipulate that the VEP should first attempt *n* retries before failover to a known backup service. The policy decision manager passes an object representation of the adaptation actions to the relevant policy enforcement point(s) to execute the adaptation policy.

4) *Web services Selection service* manages the dynamic mapping of abstract Web services defined in the composition to concrete Web services. This allows shielding the orchestration engine from changes to available services. Hence, adding, modifying and selecting among available services could be done without the need to complicate the process with the routing logic for deciding which concrete services to use. The selection of services among the equivalent services registered with a VEP is done using various selection policies. A VEP can be configured to choose between registered services in round-robin fashion, or to select the best performing service (based on the QoS metrics gathered from prior interactions or from other management entities), or to 'broadcast' the request message to multiple targets service providers concurrently and consider the first one that respond, all pending invocations are then aborted and their responses are ignored. The concurrent invocation of equivalent services is accomplished by making a copy of the message and modifying its route, then invoking multiple target services using concurrent invocation threads. This strategy is more suitable for data lookup services and freely available services such as Web search.

5) *Message Inspectors/Processing Modules* implements common handlers for enforcing typical adaptation policies. These handlers can be configured as a pipeline to manipulate and pre/post-process both request and response messages as instructed by adaptation policies. Among the handlers provided by this component is a Message Logger to log the messages as they pass through the messaging layer. This is useful for debugging problems, meter usage for subsequent billing to users, or trace business-level events, such as transaction over a certain amount. It can also be used for data inspection, or for service management.

6) *Message Adaptation Service* is a Message Processing Module that handles data transformation and enrichment to resolve incompatibilities between services registered with a particular VEP (i.e., structural, value and encoding mismatches). Various transformation patterns are supported, such as transform a message payload from the one schema to another; attach additional data from external sources, such as Web services calls or from database queries; split/merge messages; buffer multiple messages and aggregate them into a single one before sending them to the destination service. These transformation modules can be composed into a pipeline to transform and relay messages.

3.2 wsBus Evaluation on the WS-I Supply Chain Management Case Study

We conducted a series of benchmarking tests to assess effectiveness (i.e., impact on reliability) and efficiency (i.e., impact on performance) of wsBus in enhancing

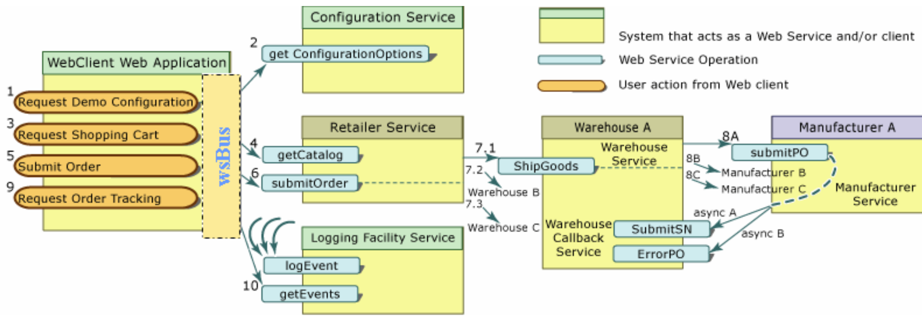

Fig. 4. WS-I Supply Chain Management (SCM) application process (adapted from [17])

reliability of Web services interactions. Our secondary aim for these tests was to discover areas of the platform that need further improvement. We used an extended Java-based implementation of WS-I (Web Services Interoperability) Supply Chain Management (SCM) application [17]. The SCM scenarios, as shown in Figure 4, are designed as Web services based interactions that simulate business activity of an online supplier of electronic goods. First a Web client calls the Retailer service's *getCatalog* operation. When the user submits the order, the Web client calls the Retailer service's *submitOrder* operation. To fulfill orders, the Retailer Web service manages stock levels in three warehouses (WA, WB, and WC). If Warehouse A cannot fulfill an order, the Retailer checks Warehouse B; if Warehouse B cannot, the Retailer checks Warehouse C. When an item in a Warehouse stock falls below a certain threshold, the Warehouse must restock the item from the Manufacturer's inventory (MA, MB, and MC). Each use case includes a logging call to a Logging Service to monitor activities of the services. A customer can track orders by using the *getEvents* operation of the Logging Facility Web service. During the SCM process enactment, participating Web services can log events by calling the *logEvent* operation of the Logging Facility Web service. Optionally, there is a Configuration Web service that lists all implementations registered in the UDDI registry for each of the Web Services in the sample application.

Our experimental setup consisted of 2 run-time configurations: 1) wsBus was not used and all invocations were direct (point-to-point) between the Web services, and 2) wsBus was placed at the client side and acted as an intermediary (broker, mediator). Both configurations used identical application logic implemented in Java. We simulated multiple concurrent Web service clients, each of which invoked deployed services multiple times. We used Apache's JMeter 2.1.1, a load generator toolset, to generate the workload and to measure the observed performance. We deployed the SCM backend Web services (Retailers, Warehouses, and Manufacturers) at a P4 2.8GHz, 1GB RAM server running Windows 2003, Tomcat 5.5 and Axis 2. JMeter stress tool (acting as the client) and wsBus were deployed at a Windows XP laptop with P4 2.8GHz and 500MB RAM. The machines were connected by a 100MB LAN.

To estimate the impact on reliability and robustness of the wsBus solution in response to QoS changes and service failures, we wrote test code that occasionally (at random times) injected exception events in the tested system. For service failures, we randomly picked some of available services and made them unavailable for a random

amount of time. For service QoS degradations, test code occasionally picked some service instances and changed their QoS values (e.g., introduced delays). We have defined monitoring policies and corrective adaptation policies for the experiments using wsBus. Monitoring policies configured messaging pipeline inspectors to intercept faults (e.g., fault message returned from the service provider, timeout fault message returned from the Web services invoker, QoS degradation event raised by the QoS constraints evaluator). When a fault was detected, the wsBus VEP used corrective adaptation policies to decide the adaptation actions. For timeout faults, these policies configured the VEP for the Retailers to first retry the invocation of the faulty services three times with a delay between retry cycles of two seconds. After exhausting the maximum number of allowed retries, the policies configured the VEP to route the request message to a different Retailer based on the response time gathered from prior interactions. (In other experiments, we have defined policies that configured concurrent invocation of the four Retailer services and considered the results coming from the first responding service.) For the Logging service we have configured a skip policy since the functionality provided by the Logging service is not business critical.

Table 1. Reliability and availability of direct interactions vs. channeling through wsBus

		Reliability	**Availability**
Direct Web services invocations without wsBus mediation	Only Retailer A used by the client	105 failures per 1000 requests	0.952
	Only Retailer B used by the client	81 failures per 1000 requests	0.992
	Only Retailer C used by the client	17 failures per 1000 requests	0.998
	Only Retailer D used by the client	91 failures per 1000 requests	0.983
Web services invocations with wsBus mediation	All 4 Retailer services exposed as 1 wsBus VEP	6 failures per 1000 requests	0.998

In a representative experiment, we compared reliability and availability of the *get-Catalog* operation in cases when a client directly calls one of the Retailer Web services (which have occasional random faults) and cases when the client calls Retailer Web services (with the same occasional random faults) through 1 VEP of the client-side wsBus. Reliability was measured as a number of failures seen by the client per 1000 requests. Availability was calculated as mean time between failures divided with the sum of mean time between failures and mean time to recover. The test results in Table 1 show that reliability and availability in cases when wsBus was used improved compared to cases when only direct interaction with individual Retailers was used. This is a simple experiment that enabled us to perform quantitative comparisons. Qualitative comparisons are more straightforward – when there are complex failures, wsBus adds useful corrective adaptation. How much useful and appropriate the adaptation is in particular circumstances, depends solely on the policies and their priorities – if a human defines an inappropriate policy, wsBus will try to enforce it.

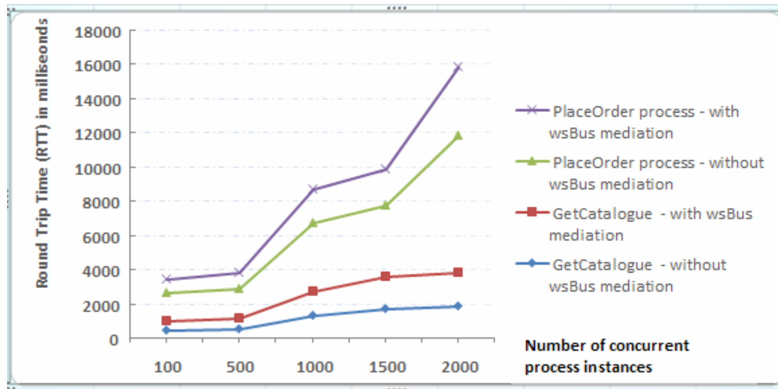

Fig. 5. Round trip time (RTT) for direct interactions vs. channeling through wsBus

To estimate the impact of introducing wsBus on performance of Web services compositions, we used the implemented SCM Web services composition to measure and examine 2 key performance metrics: round trip time and throughput. *Round Trip Time (RTT)* is defined as the period from the time a service consumer sends a request to the time when it successfully receives full reply from its service provider. It includes execution time of the service implementation, time consumed by the supporting provider-side software (e.g., application server, Web server, database server), queue waiting time (if any) inside wsBus, and network delays. *Throughput* is defined as the average number of successful requests processed in a sampling period.

Figure 5 shows round trip time for *getCatalogue* and *submitOrder* requests with varying request sizes. Each data point represents the average latency value over three independent runs of up to 2000 requests each and performed measures at different load levels. The delay between requests is set to zero to increase the load on the server. These data show that channeling of SOAP through wsBus is slower (usually about 10%, which is not drastic) than direct SOAP-over-HTTP, due to the overheads introduces by the added QoS features in wsBus. Our analysis of the main reasons of delays introduced by wsBus points to the high number of threads created to serve the requests. When a message arrives at the Listener component, a thread is created to serve the request, and this does not scale well with high number of requests. This will be avoided in our new .NET reimplementation of wsBus. Another important source of wsBus delays is the need to import, parse, and process policies. In our .NET reimplementation of wsBus we will minimize this overhead by working with object representation of policies, which is updated only when policies change.

4 Related Work

While Web services based business processes are gaining wider adoption, tools and middleware frameworks in this space do not yet provide adequate support for modeling and enacting dynamic process adaptations. Several ongoing academic and industrial efforts recognize the need to extend Web services composition middleware with

mechanisms to provide dynamic adaptation. However, our work has unique characteristics. We adopt a policy-based approach that builds on the established policy-based management principles [12], while decoupling between sensors that monitor and detect adaptation triggers and effectors that react to and handle such triggers. Our middleware performs different types of adaptation and contains solutions at different Web services middleware layers. Also, our technological base is different (extensions of WS-Policy and Microsoft. NET 3.0 have not been previously studied in detail), which leads to different architectural solutions. Furthermore, the ultimate goal of our research in this area is business-driven adaptation of Web services compositions, while related works aim at improvement of technical metrics. We briefly discuss next how our work differs from and complements the main recently published works.

Probably the closest related work 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 BPEL invoke activities that call the Monitoring Manager, the component in charge of runtime evaluation of monitoring policies to detect anomalous conditions. This approach is similar to ours in that monitoring policies are specified externally rather than being embedded into the process specification. The proposed approach 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) to customize the process to cater for special cases or to handle faults and address anomalous situations.

Another related work is [3], which suggested an aspect-oriented extension to BPEL to enable dynamic weaving of aspects into Web services compositions. In their work, a process runs inside a process container that provides middleware services to BPEL processes. However, we believe that some of the QoS aspects that they tried to address, e.g. security and state persistence, can be addressed more naturally via interception at lower-layer messaging middleware rather than augmenting a BPEL engine with the ability to call low-level middleware services. We argue that a process should focus solely on the control flow and message routing between composed services. On the other hand, enforcement of adaptation policies in our approach can be either delegated to the underlying SOAP messaging middleware that mediates the Web services interactions or enacted by the process orchestration engine via dynamic adaptation of Web services composition instances. This operation at the SOAP messaging layer can shield the process orchestration layer from the need to provide fault management.

In [8], the authors presented RobustBPEL as an approach to improve reliability of BPEL processes via automatic generation of exceptions handling BPEL constructs, as well as generation of a Web services proxy for each participating service to discover and bind to equivalent Web services that can substitute a faulty service. However, the proposed approach does not consider potential dependencies between Web service operations. Our approach is more general and controls adaptation using policies that can be checked for consistency.

Significant progress (e.g., see [14]) has been achieved in the field of dynamic composition of Web services by leveraging artificial intelligence planning and semantic Web services to obtain new Web service compositions when the measured QoS violates a Service Level Agreement (SLA). However, such approaches incur considerable

overhead and their practical applicability to business problems is still to be proven. We argue that our approach is more practical and lightweight.

Our MASC middleware can also be seen as a complement to Web services management (WSM) systems, such as the Web Service Offerings Infrastructure (WSOI) [16]. These systems provided mechanisms for measuring, evaluating, and managing Web services to ensure that QoS objectives are met. The central concept in such systems is often an XML-based contract that formally specifies QoS assurances (e.g., about response time, throughput, availability, and reliability). However, most of the proposed approaches focus on monitoring and/or QoS-based selection of individual Web services. Our work aims to go beyond the past approaches towards self-adaptive and more agile business processes implemented as Web services compositions.

The work in [2] proposed a general extension of the service oriented architecture to support autonomic behavior of Web services, but the proposed architecture does not address the requirements of adaptive business process execution.

5 Conclusions and Future Work

Dynamic adaptation of Web services compositions is an important step towards agile business processes that need to continually adapt to keep fulfilling the functional and QoS requirements of their dynamic business environment. In this paper, we presented MASC – a policy-based middleware for monitoring and adaptation of Web services compositions. The underlying design principle of our approach is the separation of concerns between the process definition and the monitoring and control, considerably simplifying Web services composition development and management. The benefits of the work presented in this paper are of twofold:

(1) A novel language, WS-Policy4MASC, is used to declaratively specify monitoring policies for detection of adaptation needs (e.g., special cases and faults) and adaptation policies that guide process reconfiguration (e.g., fault correction). The externalization and explicit definition of such policies helps in keeping the Web services composition simple and uncluttered. Further, these policies can evolve independently, while allowing potential reuse.

(2) The new MASC middleware architecture has been designed and implemented to autonomously make and coordinate enforcement of runtime adaptation decisions across both the business process orchestration layer and the SOAP messaging layer. Currently, MASC supports both static and dynamic customization of Web services composition instances, as well as corrective adaptation at the messaging layer.

The paper reports the progress on MASC middleware design and implementation and highlights how our previous work on the wsBus and adaptation strategies fits into the overall MASC architecture. To demonstrate feasibility and evaluate effectiveness of our adaptation techniques at the SOAP messaging layer, wsBus was deployed in a supply chain management Web services composition. The preliminary measurements confirmed improved availability and reliability at an acceptable increase in latency. Also, feasibility of our process-level static and dynamic customization was assessed using scenarios from the stock trading domain.

Our ongoing work is on providing support for other types of adaptation, i.e., corrective adaptation at the business process orchestration layer to handle process-level

faults, optimizing adaptation to improve extra-functional properties, and preventive adaptation to avoid future faults and/or QoS degradations before they occur. We are also extending our middleware to enable making and enacting adaptation decisions (e.g., optimal configuration of running Web services compositions) based on not only event-condition-action rules, but also more abstract utility/goal policies describing how to determine business benefits/costs and maximize business value by performing adaptations. These ambitious extensions aim to position MASC as a middleware for autonomic business-driven management of Web services compositions.

Acknowledgments. This work is a part of the research project "Building Policy-Driven Middleware for QoS-Aware and Adaptive Web Services Composition" sponsored by the Australian Research Council (ARC) and Microsoft Australia. We also thank A/Prof. Boualem Benattallah for insightful discussions and his comments.

References

1. Baresi, L., Guinea, S., Plebani, P.: WS-Policy for Service Monitoring. Proc. of the 6th International Workshop on Technologies for E-Services (TES 2005, Trondheim, Norway), Lecture Notes in Computer Science (LNCS), Vol. 3811. Springer (2005) 72-83
2. Birman, K., Van Renesse, R., Vogels, W.: Adding High Availability and Autonomic behavior to Web Services. Proc. of the 26th International Conference on Software Engineering (ICSE 2004, Edinburgh, Scotland, UK). IEEE-CS (2004) 17-26
3. Charfi, A., Mezini, M.: An Aspect-Based Process Container for BPEL. Proc. of the First Workshop on Aspect-Oriented Middleware Development (AOMD 2005, Grenoble, France). ACM (2005) #4
4. Curbera, F., Leymann, F., Storey, T., Ferguson, D., Weerawarana, S.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-ReliableMessaging and More. Prentice Hall (2005)
5. Erradi, A., Maheshwari, P.: A Broker-Based Approach for Improving Web Services Reliability. Proc. of the IEEE International Conference on Web Services 2005 (ICWS'05, Orlando, Florida, USA). IEEE -CS (2005) 355 - 362
6. Erradi, A., Maheshwari, P., Tosic, V.: Recovery Policies for Enhancing Web Services Reliability. Proc. of the IEEE International Conference on Web Services 2006 (ICWS'06, Chicago, Illinois, USA). IEEE-CS (2006)
7. Erradi, A., Maheshwari, P.: AdaptiveBPEL: Policy-Driven Middleware for Flexible Web Services Composition. In Proc. of the EDOC 2005 Middleware for Web Services Workshop (MWS'05, Enschede, The Netherlands). IEEE-CS (2005) 5-12
8. Ezenwoye, O. Sadjadi, S.M.: Enabling Robustness in Existing BPEL Processes. Proc. of the 8th International Conference on Enterprise Information Systems (ICEIS-06, Paphos, Cyprus). INSTICC (2006)
9. Geppert, A., Tombros, D., Dittrich, K.: Defining the Semantics of Reactive Components in Event-Driven Workflow Execution with Event Histories. Information Systems, Vol. 23, No. 3/4. Elsevier (1998) 235-252.
10. Mens, T., Buckley, J., Zenger, M., Rashid, A.: Towards a Taxonomy of Software Evolution. Proc. of the Workshop on Unanticipated Software Evolution (Warsaw, Poland). (2005)

11. Salle, M., Bartolini, C.: Management by Contract. Proc. of the IFIP/IEEE International Symposium on Network Operations and Management (NOMS'04, Seoul, South Korea). IEEE (2004) 787-800
12. Sloman, M.: Policy-Driven Management for Distributed Systems. Journal of Network and Systems Management, Vol. 2, No. 4. Kluwer (1994) 333-360
13. Tai, S., Mikalsen, T., Wohlstadter, E., Desai, N., Rouvellou, I.: Transaction Policies for Service-Oriented Computing. Data & Knowledge Engineering, Vol. 51, No. 1. Elsevier (2004) 59-79
14. Verma, K., Doshi, P., Gomadam, K., Miller, J., Sheth, A.: Optimal Adaptation in Web Processes with Coordination Constraints. Proc. of the IEEE International Conference on Web Services 2006 (ICWS'06, Chicago, Illinois, USA). IEEE-CS (2006)
15. Verma, K., Sheth, A.P.: Autonomic Web Processes. Proc. of the Third International Conference Service-Oriented Computing (ICSOC'05, Amsterdam, The Netherlands), Lecture Notes in Computer Science (LNCS), Vol. 3826. Springer (2005) 1-11
16. Tosic, V., Lutfiyya, H., Tang Y.: Extending Web Service Offerings Infrastructure (WSOI) for Management of Mobile/Embedded XML Web Services. Proc. of the 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06, San Francisco, California, USA). IEEE-CS (2006) 87-95
17. The Web Services Interoperability Organization (WS-I): Supply Chain Management Sample Application Architecture. Web resource (version Dec. 9, 2003; accessed Sept. 1, 2006). On-line at: http://www.ws-i.org/SampleApplications/SupplyChainManagement/2003-12/SCMArchitecture1.01.pdf (2003)