A Language-based Approach to Addressing Reliability in Composite Web Services

Onyeka Ezenwoye
Electrical Engineering and
Computer Science Department
South Dakota State University
Brookings, SD 57007
Email: *onyeka.ezenwoye@sdstate.edu*

S. Masoud Sadjadi
School of Computing and
Information Sciences
Florida International University
11200 SW 8th Street, Miami, FL 33199
Email: *sadjadi@cs.fiu.edu*

Abstract

With Web services, distributed applications can be encapsulated as self-contained, discoverable software components that can be integrated to create other applications. BPEL allows for the composition of existing Web services to create new higher-function Web services. We identified that the techniques currently applied at development time are not sufficient for ensuring the reliability of composite Web services In this paper, we present a language-based approach to transparently *adapting BPEL processes to improve reliability. This approach addresses reliability at the Business process layer (i.e the language layer) using a code generator, which weaves fault-tolerant code to the original code and an external proxy. The generated code uses standard BPEL constructs, and therefore, does not require any changes to the BPEL engine.*

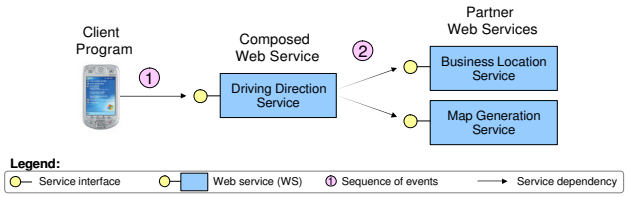
Keywords: Web service composition, Reliability, Adaptability, Business Process.

1 Introduction

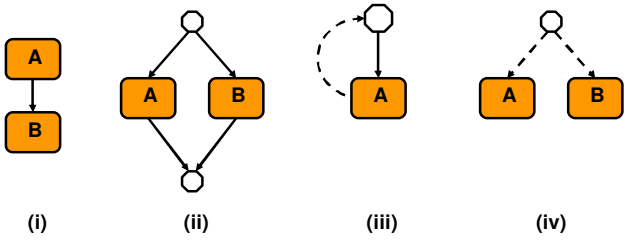
Web services are gaining acceptance as the predominant standards-based approach to building open distributed systems. With Web services, distributed applications can be encapsulated as self-contained, discoverable and Internet-accessible software components that can be integrated to create other applications. The fundamental aspects of Web services can be summarized as follows: (1) strict separation of service interface description, implementation, and binding; (2) declarative policies and Service Level Agreements (SLAs) to govern service interactions; and (3) loosely coupled, standards-based and message-centric interactions between autonomous and replaceable service components [1]. To facilitate flexibility and interoperability, Web services are described using a standard, machine-readable, XML-based language called *Web Service Description Language*. This service description provides the details necessary to interact with the service, including message formats that detail the operations, transport protocols, and location. Finally, interaction with Web services is achieved through SOAP messaging.

The family of specifications that make up the Web service standards includes a specification for service composition known as *Business Process Execution Language* (BPEL). BPEL allows for the composition of existing Web services to create new higher-function Web services [2]. BPEL is used to define workflows that represent composite services. The composite services, also known as *business processes*, contain activities that coordinate the interaction between the partner services in the composition. Figure 1(a) illustrates a business process that is a composition of two service: (1) a service that retrieves the addresses of nearby businesses; and (2) a service that gets the driving directions to a given address. Figure 1(b) depicts a basic set of workflow patterns that are supported by BPEL. In the sequence pattern (Figure 1(b)(i)), an activity in a process is enabled after the completion of another activity in the same process. Parallelism (Figure 1(b)(ii)), allows activities to be executed simultaneously. Loops (Figure 1(b)(iii)), allow for one or more activities to be executed repeatedly. In the choice pattern (Figure 1(b)(iv)), a number of branches are chosen and executed as parallel threads. Based on these basic patterns, more sophisticated constructs can be built [3].

As the use of Web services continues to grow, so has the need to deliver reliable service compositions with precise Quality of Service (QoS) attributes covering functional correctness, performance and dependability [1]. This is because current Web services standards provide limited constructs for specifying exceptional behavior and recovery



(a) A Business Process that integrates remote components to create a new composite component for providing driving directions.



(b) Selected workflow patterns supported by BPEL: (i) Sequence, (ii) Parallelism, (iii) Loop, (iv) Choice.

Figure 1.

actions. Currently, BPEL is a composition language that mainly concentrates on modeling the business process in terms of interacting Web services but does not consider the behavior of such models at runtime.

While it is relatively easy to make an individual service fault-tolerant, addressing reliability and availability of Web services collaborating in multiple application scenarios is a challenging task. This is because the integration of multiple services, which are potentially developed and maintained on autonomous heterogeneous environments, introduces new levels of complexity in management. Thus the composed service has no influence over the factors affecting QoS provision and partner services can spontaneously appear and disappear over on the Internet. Moreover, services may fail because of problems in their execution such as network faults, overload and lack of resources [1].

Given the unreliability of communication channels, the unbounded communication delays, and the autonomy of the interacting services, it is difficult for developers of composite services to anticipate and account for all the dynamics of such interactions. There is therefore a need for adaptability in composed services to make them more robust and dependable. The need for adaptability is particularly evident in complex long-running applications as is found in scientific Grid computing. In Grid computing, computational and storage resources are exposed as an extensible set of networked services that can be aggregated to create higher-function applications [4]. These highly available applications need to remain operational and rapidly responsive even when failures disrupt some of the nodes in the system.

In this paper, we present a *systematic* approach to making existing aggregate Web services more tolerant to the failure. We demonstrate how a composite Web service, defined as a BPEL process, can be instrumented automatically to monitor its partner Web services at runtime. To achieve this, events such as faults and timeouts are monitored from within the adapted process. We show how our adapted process is augmented with a proxy that dynamically replaces failed services. In doing this, we improve the fault tolerance and performance of BPEL processes by transparently adapting their behavior. By *transparent*, we mean the following; first, the adaptation preserves the original behavior of the business process and does not tangle the code that provides self-healing and self-optimization behavior with that of the business process; and second, the fault-tolerant approach does not need any modification of the BPEL engine[1]. This transparency is achieved by using a *dynamic* proxy that encapsulates the autonomic behavior (adaptive code).

The rest of this paper is is structured as follows. Section 2 provides a background in addressing reliability in composite components. Section 3 overviews our approach and gives a brief introduction to the RobustBPEL framework, we also describes the dynamic proxy. Section 4 contains some related work. Finally, some concluding remarks are provided in Section 5.

2 Addressing Reliability in Composite Components

The goal of *fault-tolerance* is to improve *dependability* in a *system* by enabling it to perform its intended functions in the presence of a given number of faults [5]. There exists several definitions of dependability. These definitions often depend on the attributes (e.g., availability, reliability and safety) of the system that are being defined as a criterion to decide whether or not a system is dependable at a given time. The attribute defined may depend on the intended use of the system [6].

In general, dependability is based on the notion of *reliance* in the context of interacting components. It associates to the relation *depends upon*, where a component *A* depends upon a component *B* if the correctness of *B's* service delivery is necessary for the correctness of *A's* service delivery [6]. This relationship is typical of composite services since they are entirely dependent on interaction with partner services. An error may propagate from a partner to the composite thereby creating new errors.

[1] A BPEL engine is a virtual machine that executes BPEL grammar

Our work focuses on the *reliability* attribute of dependability with a specialization on *robustness* as a *secondary attribute*. Avizienis [6] defines reliability as the continuity of *correct* service, it defines robustness as dependability with respect to *external* faults. Techniques for achieving dependability that are applied at development time are not sufficient enough for ensuring the reliability of composite Web services that are expected to dynamically discover and assemble components, configure themselves, and operate securely and reliably in a completely automated manner. This calls for the development of new reliability techniques that introduce *autonomic* functionality to address these challenges.

New reliability techniques for service compositions can be developed at four layers. Figure 2 shows the different layers at which reliability techniques can be applied.

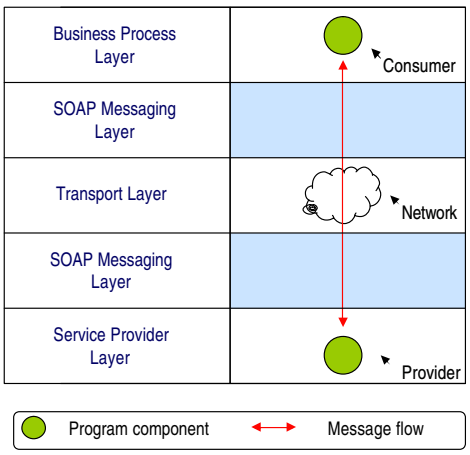


Figure 2. Layers to apply reliability techniques

Service provider layer: At this level, reliability focuses on the service hosting environment. Here, reliability can be achieved by techniques that provide redundancy of computation and data, load sharing to improve performance and fault tolerance, and clustering which interconnects multiple servers to avoid single point of failure [1].

Transport layer: At this level, the focus is on implementing reliable messaging for Web services at the transport layer. Since the reliability of SOAP messaging is dependent on the underlying transport layer, techniques in this layer center on using message-oriented middleware (MOM) [7] to ensure reliability and robustness of message traffic.

SOAP messaging layer: Addressing reliability at this layer focuses on extending SOAP messages to include reliability properties that allow messages to be delivered reliably between services in the presence of component, system, or network failures.

Business process layer: Reliability at this layer aims to provide dependable composition of Web services through advanced failure handling and compensation-based transaction protocols [1]. Efforts in this layer can be categorized into two groups; *language-based* and *non language-based* approaches. *Language-based* techniques provide advanced failure handling and adaptability by augmenting the process logic with additional language constructs while *non-language* based approaches focus specifically on the process supporting infrastructure such as the execution engine.

Our work fits into this category by enabling adaptability in BPEL process to address the concerns raised above. One might argue that BPEL should be extended with constructs to handle those concerns. However, this would increase the complexity of the language and it is also against the principle of separation of concerns. Constructs for specifying exceptional behavior and recovery actions should be modularized and externalized and not scattered and tangled with the service implementation. Entangling the logic for exceptional behavior and recovery actions with the business logic of the application negatively impacts maintainability and adaptability.

3 Overview of Our Approach

We developed RobustBPEL [8] as part of the transparent shaping programming model. Using RobustBPEL, we can automatically generate an *adapt-ready* version of an existing BPEL process. In a typical composed Web service (see Figure 1(a)), a request is first sent by the client program, then the composite Web service interacts with its partner Web services and responds to the client. If one of the partner services fails, then the whole process is subject to failure. To avoid such situations, adapt-ready version of the original composed service monitors the behavior of it partners and tries to tolerate their failure. As monitoring all the partner Web services might not be necessary, the developer can select only a subset of Web service partners to be monitored. The adapt-ready process monitors selected Web services and in the presence of faults it will forward the corresponding request to a *proxy*. The proxy is generated specifically for this adapt-ready process and provides the same interface as those of the monitored Web services. The proxy in its turn forwards the request to a *substitute* Web service.

In this work, we make the following assumptions: (1) two services are *substitute*, if they implement the same interface; (2) Web service partners are *stateless* and *idempotent*. It is possible for two applications to be functionally equivalent without necessarily having the exact same interface. When this occurs, a wrapper interface/service can be used to harmonize the differences in their interfaces.

Given the rapid uptake of the service oriented program-

ming model, we expect the emergence of numerous services that are functionally equivalent and thus can be substituted. For instance, in our driving-direction example (Figure 1(a)), if the default map generation service provided by Google fails, it should be possible to substitute this service with that of MSN, Yahoo! or Mapquest. Also, in Grid programming environments where scientific applications are run on computational Grids, a failed (or slow) Grid service can be replaced by another service on the Grid. Thus, in our approach, we associate an adapt-ready composed service with a *dynamic proxy* (which is also a Web service) and its job is to discover and bind to *substitute* Web services.

3.1 High-Level Architecture

Figure 3 illustrates the architectural diagram of an application using an adapt-ready BPEL process augmented with its corresponding dynamic proxy. This figure shows the steps of interactions among the components of a typical adapt-ready BPEL process. Similar to a static proxy, the interface for the generated dynamic proxy is exactly the same as that of the monitored Web service. Thus, the operations and input/output variables of the proxy are the same as that of the monitored invocation. When more than one service is monitored within a BPEL process, the interface for the specific proxy is an aggregation of all the interfaces of the monitored Web services. For example, the dynamic proxy in Figure 3 has pt_i and pt_j, which are the port types of the two monitored Web services (namely, WS_i and WS_j). At runtime, if a monitored service fails (or an invocation timeout occurs), the input message for that service is used as input message for the proxy. The proxy invokes the equivalent service with that same input message. A reply from the substitute service is sent back to the adapted BPEL process via the proxy.

Although the adapt-ready BPEL process remains a functional Web service and the proxy is an autonomic Web service (encapsulates autonomic attributes), functional Web services can behave in an autonomic manner by virtue of their interaction with autonomic Web services. By replacing failed and delayed services with substitutes, the proxy service provides self-healing and self-optimization behavior to the BPEL process, thereby making the BPEL process autonomic.

3.2 Incorporating Generic Hooks inside the Adapt-Ready BPEL Processes

Following the Transparent Shaping programming model [9], we first need to incorporate some generic hooks at sensitive *joinpoints* in the original BPEL process. These joinpoints are certain points in the execution path of the program at which adaptive code can be introduced at run

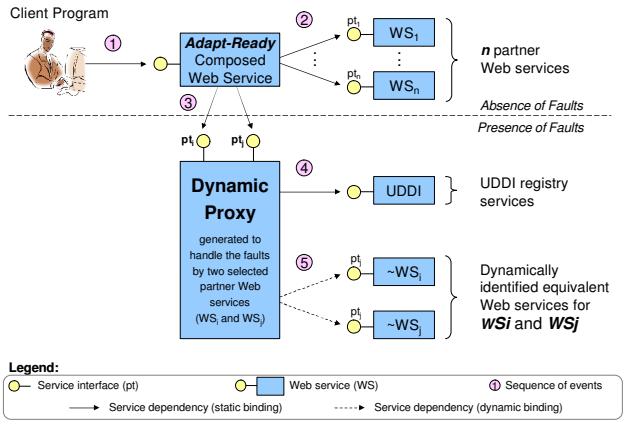


Figure 3. Architectural diagram showing the sequence of interactions among the components in an adapt-ready BPEL process augmented with its corresponding *dynamic* proxy.

time. Key to identifying joinpoints is knowing where in the BPEL process *sensing* and *actuating* are required and inserting appropriate code (hooks) to do so. Because a BPEL process is an aggregation of services, the most appropriate place to insert interception hooks is at the interaction joinpoints (*i.e.*, the *invocation* instructions). The monitoring code we insert is in the form of standard BPEL constructs to ensure the portability of the modified process.

We adapt the BPEL process by identifying points in the process at which external Web services are invoked and then wrapping each of those invocations with a BPEL `scope` that contains the desired fault and event handlers. A fault can be a programmatic error generated by a Web service partner of the BPEL process or unexpected errors from the Web service infrastructure. The unmonitored invocation is first wrapped in a `scope` container which contains fault and event handlers. The fault handlers detect any faults generated as a result of the invocation of the partner Web service. A fault-handling activity is defined, which basically forwards the request to the dynamic proxy. When a fault is generated by the partner service invocation, this fault is caught and the proxy service is invoked to substitute for the unavailable or failed service.

For the event handler, an alarm clause is used to specify a timeout. A timeout can be used, for instance, to limit the amount of time that a process can wait for a reply from an invoked Web service. If the partner service fails to reply within the time stipulated in the timeout event, a generated fault forces the monitored invocation to terminate and the proxy service is invoked as a substitute.

4 Related Work

Since Web services technology is still emerging, most of the work that aim to address the requirements for reliable and fault tolerant Web services execution are still in their infancy. These efforts can be distinguished by their focus on different layers (see Figure 2) of the Web services infrastructure . We note that our work is focused on the business process layer and as a result the work in the other layers are complementary to ours.

4.1 SOAP Messaging Layer

Some works aim to address the reliability of Web services from the *SOAP messaging layer* by addressing the issues concerning reliable transport-independent messaging. To this end, SOAP-based protocols like WS-ReliableMessaging [10] and WS-Reliability [11] strive to standardize message delivery by specifying rules for acknowledgment, message correlation, ordered delivery and so on. Such a protocol does however contribute to inefficiency if the underlying transport layer does use protocols that address reliable message delivery [1].

4.2 Transport Layer

Other approaches and technologies focus on implementing reliable massaging for Web services at the transport layer. The reliability of SOAP messaging largely depends on the underlying transport chosen. Since SOAP-over-HTTP is not reliable, attempts are being made to build messaging middleware that accept messages from sending processes and delivers them reliably to receiving processes. Reliable messaging implementations communicate across a network on behalf of senders and receivers, and have built-in transactional support to manage message conversations in the context of a larger business process [1]. Examples of message-oriented middleware are IBM WebsphereMQ [12] and Microsoft Message Queuing (MSMQ) [13]. These implementations support their own proprietary messaging APIs and protocols, as well as the standard Java Message Service (JMS) API [14]. These approaches however do not guarantee reliability for multi-hop messaging over different protocols as they assume that reliable transport protocols will be available for the entire path of the message [1, 15].

4.3 Service Provider Layer

At this layer, approaches focus on the service hosting container. Here, approaches aim to achieve reliability by using techniques that provide redundancy of computation and data, load sharing to improve performance and fault tolerance, and clustering to avoid single point of failure [1].

Dialani et al. [16] provide an approach to enabling fault tolerance in *stateful* Web services by requiring the developer to implement an interface for rollback and checkpoint.

Birman et al. [17] propose extensions to the Web services architecture to support mission-critical applications. They propose some extensions to track the health of individual Web service.

4.4 Business Process Layer

We further categorize works that focus on the business process layer into two groups: *language-based* and *non language-based*.

Non-language based approaches focus specifically on the process supporting infrastructure such as the execution engine. They include wsBus [18], which is a lightweight service-oriented middleware for transparently enacting recovery action in service-based processes. This approach is modular and separates the business logic of the process from the QoS requirements; however, this approach requires the installation of additional middleware.

Charfi et al. [19] use an aspect-based container to provide middleware support for BPEL. The process container is the runtime environment for the BPEL process. All interactions go through the container which plugs in support for non-functional requirements. This framework is different from ours because it requires a purpose built BPEL engine.

Language-based techniques provide advanced failure handling and adaptability by augmenting the process logic with additional language constructs. These approaches include BPEL for Java (BPELJ), which combines the capabilities of BPEL and the Java programming language. This combination is achieved by extending the BPEL to allow for sections of Java code to be included in BPEL process definitions. BPELJ, however, requires an extended BPEL engine that understands the additional constructs. Also, exception handling logic in BPELJ often gets tangled with the process logic, thus hampering maintainability.

Other language-based techniques include the work done by Baresi et al. [20]. In their approach, BPEL processes are monitored at run-time to check whether individual services comply with their contracts. Monitors are automatically defined as additional services and linked to the service composition via annotations in the composition. This approach achieves the desired separation of concern, however, it requires manually modifying the original BPEL process and the monitoring code is entangled with the process logic. The manual modification of BPEL code is not only difficult and error prone, but also hinders maintainability.

5 Conclusion

Techniques that are applied at development time are not sufficient enough for ensuring the reliability of composite Web services that are expected to dynamically discover and assemble components, configure themselves, and oper-

ate securely and reliably in a completely automated manner. This calls for the development of new reliability techniques that introduce *autonomic* functionality to address these challenges. New reliability techniques for service compositions can be developed at four layers, namely; (1) Service provider, (2) SOAP messaging, (3) Transport and (4) Business process layers. We presented a language-based approach to transparently adapting BPEL processes to improve reliability. This approach addresses reliability at the Business process layer.

References

[1] A. Erradi, P. Maheshwari, and V. Tosic, "A policy-based middleware for enhancing web services reliability using recovery policies," in *Proceedings of the 2006 IEEE International Conference on Web Services*, Chicago, USA, September 2006.

[2] O. Ezenwoye and S. M. Sadjadi, "Composing aggregate web services in BPEL," in *Proceedings of The 44th ACM Southeast Conference*, Melbourne, Florida, March 2006.

[3] D. Cybok, "A grid workflow infrastructure: Research articles," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1243–1254, 2006.

[4] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid services for distributed system integration," *Computer*, vol. 35, no. 6, pp. 37–46, 2002.

[5] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts." *IEEE Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 01, no. 1, pp. 11–33, 2004.

[7] S. Goel, H. Sharda, and D. Taniar, "Message-oriented-middleware in a distributed environment." in *Third International Workshop on Innovative Internet Community Systems*, June 2003, pp. 93–103.

[8] O. Ezenwoye and S. Sadjadi, "RobustBPEL2: transparent autonomization in business processes through dynamic proxies," in *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems*, Sedona, Arizona, March 2007.

[9] S. M. Sadjadi, P. K. McKinley, and B. H. Cheng, "Transparent shaping of existing software to support pervasive and autonomic computing," in *Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005*, St. Louis, Missouri, May 2005.

[10] "Web services reliable messaging," http://www.ibm.com/developerworks/library/specification/ws-rm/.

[11] "WS-Reliability 1.1," November 2004, http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability%-1.1-spec-os.pdf.

[12] L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*. Wiley, 1996.

[13] "Microsoft. microsoft message queuing MSMQ," http://www.microsoft.com/windowsserver2003/technologies/msmq/default.ms%px.

[14] M. Hapner, R. Burridge, and R. Sharma, "Java message service specification," Sun Microsystems, Tech. Rep., Nov. 1999.

[15] S. Tai, T. Mikalsen, and I. Rouvellou, "Using message-oriented middleware for reliable web services messaging." in *Second International Workshop of Web Services, E-Business, and the Semantic Web*, 2003, pp. 89–104.

[16] V. Dialani, S. Miles, L. Moreau, D. D. Roure, and M. Luck, "Transparent fault tolerance for web services based architectures," in *Eighth International Europar Conference*. Padeborn, Germany: Springer-Verlag, aug 2002.

[17] K. P. Birman, R. van Renesse, and W. Vogels, "Adding high availability and autonomic behavior to web services." in *Proceedings of the 26th International Conference on Software Engineering*. Edinburgh, United Kingdom: IEEE Computer Society, May 2004, pp. 17–26.

[18] A. Erradi and P. Maheshwari, "wsBus: QoS-aware middleware for relaible web services interaction," in *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service*, Hong Kong, China, 2005.

[19] A. Charfi and M. Mezini, "An aspect based process container for BPEL," in *Proceedings of The First Workshop on Aspect-Oriented Middleware Developement*, Genoble, France, November 2005.

[20] L. Baresi, C. Ghezzi, and S. Guinea, "Smart monitors for composed services," in *Proceedings of the 2nd international conference on Service oriented computing*. ACM Press, 2004, pp. 193–202.