Modeling Non-Functional Aspects in Service Oriented Architecture

Hiroshi Wada and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393
{shu, jxs}@cs.umb.edu

Katsuya Oba
OGIS International, Inc.
Palo Alto, CA 94301
oba@ogis-international.com

Abstract

Service Oriented Architecture (SOA) is an architectural style to reuse and integrate subsystems in existing systems for designing new applications. Each application is designed in an implementation independent manner using abstract concepts: network services and connections between network services. In SOA, the non-functional aspects of services and connections should be described separately from their functional aspects because different applications use services and connections in different non-functional contexts. This paper proposes a UML profile to graphically design the non-functional aspects in SOA and maintain them in an implementation independent manner. This paper presents the design of the proposed UML profile and describes how it is used in service-oriented application development.

1. Introduction

One of the current key issues in large-scale distributed systems is to reuse (or repurpose) and integrate existing systems to build new applications in a cost effective manner [1, 2]. Service Oriented Architecture (SOA) addresses this issue by improving the reusability and maintainability of distributed systems [3, 4]. It is an architectural style to design applications in an implementation independent manner using two major concepts: *network services* and *connections between network services*. Each service encapsulates the functions of a subsystem in an existing system and hides the subsystem's implementation details (e.g., programming languages and remoting middleware) from developers. Each connection defines how services are connected with each other and how messages are exchanged through the connection, and hides implementation details of the message exchanges (e.g., messaging protocols and message routing) from developers. Developers can reuse and combine services to build their applications without knowing the implementation details of services and connections.

In SOA, the non-functional aspects of services and connections should be defined separately from their functional aspects (i.e., business logic) because different applications use the services and connections in different non-functional contexts. For example, an application may unicast messages to a service, and another may anycast messages to replicas of the service in order to distribute workload. Also, an application may use a service via reliable connection that guarantees message delivery when the service is hosted in an unreliable network (e.g., the Internet), and another application may use the service via connection that does not guarantee message delivery when the service is hosted in a reliable network (e.g., intranet). The separation between functional and non-functional aspects improves the reusability of services and connections. It also enables the two different aspects to evolve independently, and improves the ease of understanding application architectures. This contributes to increase the maintainability of applications.

This paper proposes a Unified Modeling Language (UML) profile to graphically model non-functional aspects in SOA. A UML profile extends (or specializes) the standard UML elements (e.g., class and association) in order to precisely describe domain specific or application specific concepts [5, 6]. The proposed UML profile allows developers to describe and maintain non-functional aspects in SOA as UML models (composite structure diagrams and class diagrams) in a visual and intuitive manner. Non-functional aspects can be modeled without depending on any particular implementation technologies. Supporting tools accept the UML models defined with the proposed profile and transform them into application code using certain implementation technologies such as Enterprise Service Buses (ESBs) [7]. This paper focuses on the design details of the proposed UML profile, and describes how it is used in service-oriented application development.

2. Background and Motivation

UML is a modeling language to specify application designs as graphical diagrams. It defines the syntax (or notation) and semantics of every model element that appears in diagrams (e.g., class, interface and association). The syntax

and semantics are defined in the UML metamodel [6].

In addition to standard model elements, UML provides extension mechanisms (e.g., stereotypes and tagged-values) to specialize the standard model elements to precisely describe domain or application specific concepts [5]. A stereotype is applied to a standard model element, and specializes the semantics of the standard model element to a particular domain or application. Each stereotyped model element can have data fields specific to the stereotype, called tagged-values. Similar to data fields in a class, each tagged-value consists of a name and value. A particular set of stereotypes and tagged-values is called a UML profile. Each UML profile is defined for a specific domain or application.

For example, a UML profile for Enterprise Java Beans (EJB) [8] defines the stereotype ≪EJBEntityBean≫, which extends Class in the UML metamodel. This means the stereotype can be applied to Class. Thus, a UML class stereotyped with ≪EJBEntityBean≫ indicates that the class is designed as an EJB entity bean. The stereotype ≪EJBEntityBean≫ has a tagged-value, called EJBPersisitenceType, to specify who provides persistency to an entity bean. The tagged-value can have a value Bean or Container. Bean indicates an individual entity bean is responsible for its own persistency, and Container indicates an EJB container takes care of persistency.

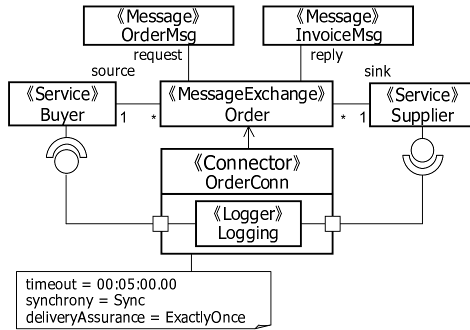

Figure 1. A Motivating Example

Figure 1 shows an example model defined with the proposed UML profile. It illustrates an order processing application in which a buyer places an order and a supplier receives it. In this example, two services (Buyer and Supplier) exchange messages. Each service is represented by a class stereotyped with ≪Service≫. These services exchange two types of messages (OrderMsg and InvoiceMsg), each of which is stereotyped with ≪Message≫. Each pair of a request and reply messages is represented by ≪MessageExchange≫. ≪Connector≫ represents a connection that transmits messages between services. In this example, messages are delivered through a connector called OrderConn. Every message exchange is bound with a connector in order to specify which connector is used to deliver messages. Each connector can have multiple filters inside. They are used to customize message transmission/processing semantics in a connector. This example uses Logger in the OrderConn connector. Logger logs message transmissions (OrderMsg and InvoiceMsg in this example). Also, each connector can have multiple tagged-values to specify additional message transmission/processing semantics. In this example, OrderConn specifies the timeout of message transmissions (five minutes), synchrony of message transmissions (synchronous) and assurance level of message delivery (exactly once). As demonstrated in Figure 1, the proposed UML profile provides an easy-to-understand abstraction to visually specify the architectures and non-functional aspects of service-oriented applications.

3. Design of the Proposed UML Profile

The proposed UML profile provides key model elements to specify service-oriented applications: *service*, *message exchange*, *message*, *connector* and *filter* (Table 1). Each of them is defined as stereotypes.

Table 1. Key Model Elements (Stereotypes) in the Proposed UML Profile

Stereotype	Description
Service	Represents a network service.
MessageExchange	Represents a pair of a request and reply messages. Specifies which services send and receive the messages.
Message	Represents a (request or reply) message.
Connector	Represents a connection between services (i.e., message source and destination). Defines the semantics of message transmission and processing. Specifies which messages (message exchange) to transmit.
Filter	Customizes the semantics of message transmission and message processing in a connector.

Figure 2 shows how the proposed UML profile defines its stereotypes by extending the UML metamodel. Each stereotype is defined as a metaclass stereotyped with ≪stereotype≫. Except Connector, four stereotypes inherit the Class metaclass in the Kernel package of the UML metamodel. Thus, they are applied to classes in user-defined models (see Figure 1). A Service can be a source or sink of each request/reply message. The source and sink are identified with source and sink, roles on two associations between a MessageExchange and Services (Figure 1). Each MessageExchange may have multiple reply messages per request message (Figure 2). Using multiplicity on two associations between a MessageExchange and Services, MessageExchange can indicate one-to-one (unicast) and one-to-many (multicast or manycast) message exchanges. For example, Figure 1 shows a one-to-one

message exchange between a Buyer and a Supplier.

Connector is a stereotype extending the Class metaclass in the InternalStructures package of the UML metamodel (Figure 2). This metaclass defines a composite class, a special type of class, which can contain other model elements (e.g., inner classes)[1] and have Ports to specify how internal model elements interact with external elements. In the proposed UML profile, a Connector can contain Filters to specify the semantics of message transmission and message processing. The Ports connected with a Connector identify the Messages it receives and sends out, using association roles input and output.

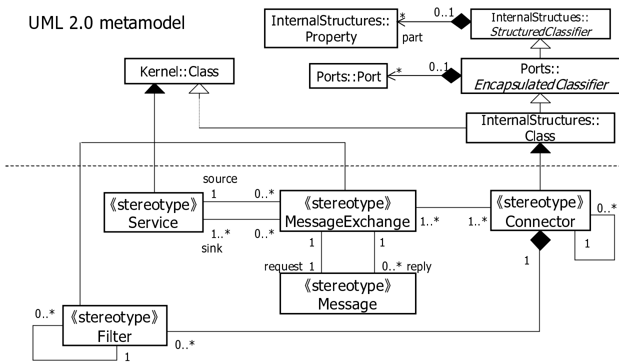

Figure 2. Metamodel Definition of Stereotypes in the Proposed UML Profile

3.1. Connector

Connector has five tagged-values (Figure 3 and Table 2). timeout is a mandatory tagged-value to specify the timeout period (in millisecond) in which a connector needs to deliver each message (see also Figure 1). If a message is not delivered to its destination (sink) within the time period, a connector discards the message.

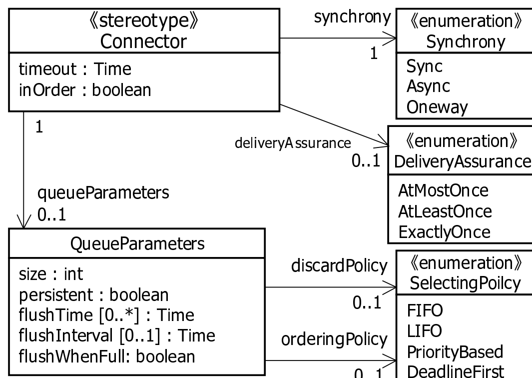

Figure 3. Tagged-Values of Connector

synchrony is a mandatory tagged-value to specify the synchrony semantics of message transmissions between a message source and destination. Synchronous, asynchronous and oneway non-blocking semantics are defined as an enumeration in Synchrony (Figure 3), and each connector chooses one of them. In Figure 1, a Buyer transmits OrderMsg messages to a Supplier synchronously.

inOrder is a mandatory tagged-value to specify whether the order of messages that a service (message destination) receives is same as the order of messages that the other service (message source) sends out. The default value of inOrder is false.

deliveryAssurance is an optional tagged-value to specify the assurance level of message delivery. Three different semantics are defined as an enumeration in DeliveryAssurance (Figure 3), and each Connector chooses one of them at a time (see Figure 1). AtLeastOnce means that a connector retries delivering a message until its destination receives the message. (A message retransmission is triggered with the timeout tagged-value.) However, the message may be delivered to its destination more than once. AtMostOnce means that a connector discards a message if the message has already been delivered to its destination; however, there is no guarantee of message delivery. ExactlyOnce satisfies the requirements of the above both semantics. It guarantees that a connector delivers a message to its destination without duplications. When inOrder is true, ExactlyOnce is implicitly (automatically) set to deliveryAssurance because duplicated or missing messages violate the inOrder semantics.

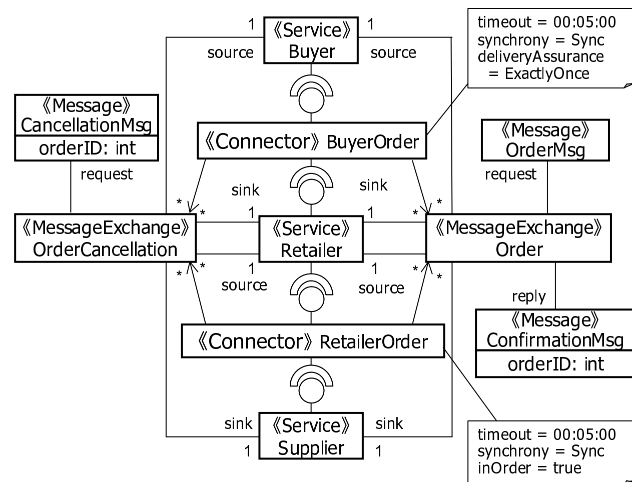

Figure 4. An Example of inOrder and deliveryAssurance

Figure 4 shows an example model using inOrder and deliveryAssurance. This example illustrates an extension to an order processing application in Figure 1. In this example, a Buyer transmits an OrderMsg to a Supplier via Retailer. After a Retailer forwards an OrderMsg from a Buyer to a Supplier, the Buyer can cancel the order by transmitting a CancellationMsg to

[1] Precisely, a composite class can contain any *classifiers*, defined in the UML metamodel.

Table 2. Tagged-Values of Connector, Service and Message

Tagged-Values	Description	Type
≪Connector≫		
timeout (mandatory)	Timeout of message transmissions in a connector	Time
synchrony (mandatory)	Synchrony of message transmissions between services	Synchrony (Enum)
inOrder (mandatory)	Automatic ordering of messages between services	Boolean
deliveryAssurance (optional)	Assurance level of message delivery	DeliveryAssurance (Enum)
queueParameters (optional)	Queuing semantics of message transmissions	QueueParameters
≪Service≫		
priority (optional)	Priority of messages that a service issues	int
timeout (optional)	Timeout period of messages that a service issues	Time
redundancy (optional)	The number of runtime instances of a service	int
≪Message≫		
priority (optional)	Priority of a message	int
timeout (optional)	Timeout period of a message	Time
schemaURI (mandatory)	URI representing the schema of a message	String

the `Retailer`, and in turn, to the `Supplier`. In this example, the order of message transmissions is important between `Retailer` and `Supplier` because an order must be delivered to `Supplier` before a corresponding order cancellation. Therefore, the `inOrder` semantics is assigned to the `RetailerOrder` connector. This semantics implicitly assigns `ExactlyOnce` to the `deliveryAssurance` semantics in the `RetailerOrder` connector.

`queueParameters` is an optional tagged-value to deploy a message queue between services (i.e., message source and destination) and specify the semantics of message queuing between them. `size` specifies the maximum number of queued messages. `flushWhenFull` specifies whether queued messages are flushed from a queue to their destinations when the queue overflows. When `flushWhenFull` is false, the overflowing queue discards a message according to `discardPolicy` (Figure 3); discarding the oldest message (First-In-First-Out), the newest message (Last-In-First-Out), the lowest priority message or the closest deadline message. These four policies are defined as an enumeration in `SelectionPolicy` (Figure 3). `flushTime` and `flushInterval` specify when and how often a queue flushes messages, respectively. `orderingPolicy` specifies how to order messages in a queue: FIFO, LIFO, highest-priority-first or earliest-deadline-first. `perisistent` specifies whether a queue stores messages in a storage (e.g., a file or database) so that the queue can recover them when it crashes unexpectedly.

Figure 5 shows an example using `queueParameters`. It illustrates an inventory management system for retail stores, warehouses and suppliers. Each `RetailStore` transmits an `OrderMsg` to an `InventoryManager` when it has no or few products in stock. The `InventoryManager` receives `OrderMsgs` from multiple `RetailStores` every two hours in a batch manner. The `OrderConn` connector implements a synchronous queue that stores and forwards `OrderMsgs`. The `InventoryManager` schedules which warehouses deliver which products to which retail stores (every two hours), and based on the shipping schedule, sends `ShippingMsgs` to `Warehouses`. If a warehouse has a small inventory of a particular product, the `InventoryManager` orders the product by sending a `PurchacingMsg` to a `Supplier`.

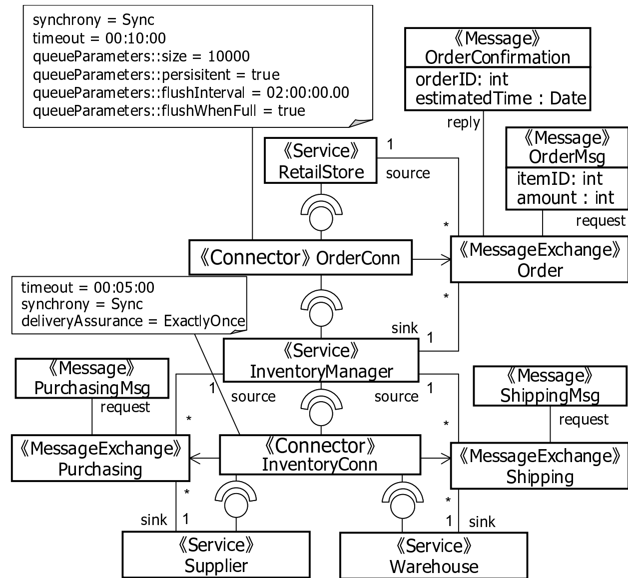

Figure 5. An Example of Queue

3.2. Filter

The proposed UML profile defines seven filters as stereotypes (Figure 6). They derive from the `Filter` stereotype. Any new filters can be defined as a subclass of `Filter`. Filters are used to customize the semantics of message trans-

mission and processing in each connector. This section shows five filters to specify message transmission semantics and two filters to specify message processing semantics.

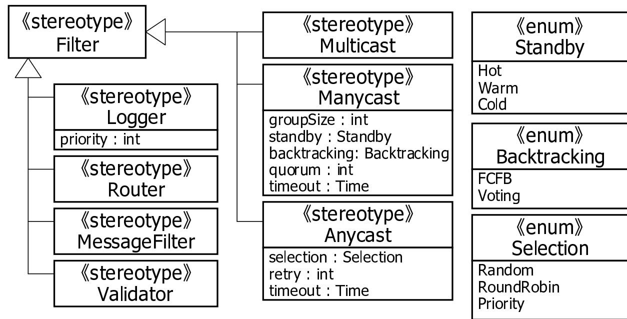

Figure 6. Tagged-Values of Filters

The stereotypes `Multicast`, `Manycast`, `Anycast`, `Logger` and `Router` are used to define the message transmission semantics in a connector (Figure 6). `Multicast` receives a request message from its source and transmits it to multiple destinations simultaneously (one-to-many message exchange). When the `Multicast` filter receives reply messages from the destinations, it sends them back to the source of the request message. `Multicast` is used to improve the efficiency of message transmissions.

Figure 7 shows an example that models stock quote notification using `Multicast`. A `StockInfoClient` subscribes for the price changes of a particular stock ticker, and a `StockInfoServer` notifies (or publishes) any price changes to the `StockInfoClient`. A `StockInfoClient` transmits a `Subscription` message to a `StockInfoServer` in a synchronous and exactly-once manner. A `StockInfoServer` multicasts a `StockQuote` message to multiple `StockInfoClient`s in asynchronous and at-least-once semantics.

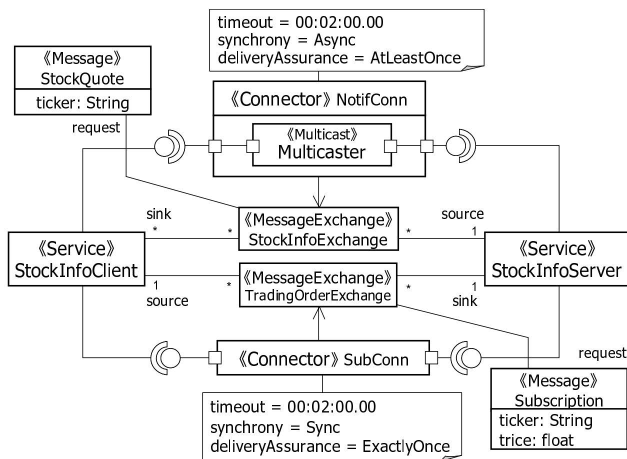

Figure 7. An Example of Multicast

`Manycast` is used to improve fault tolerance by forwarding a request message to a group of replicated destinations (i.e., to the same type of services). The tagged-value `groupSize` specifies how many services are deployed as a group. `standby` specifies the operation of replicated services: *hot standby*, *warm standby* or *cold standby*. In hot standby, all services in a group remain active to receive request messages. A `Manycast` filter sends a message to all services in a group. `Manycast` returns only one reply message to the source of a request message, out of multiple replies from services. `backtracking` defines two policies to decide which reply message to be returned. When `FCFB` (first-come-first-backtracked) is selected, a `Manycast` filter returns the first reply that it receives from destination services. When `Voting` is selected, the `Manycast` filter performs a voting process. It counts the number of reply messages and inspects their contents. If the number of replies that have the same content reaches `quorum`, the `Manycast` filter returns one of the replies. If the number does not reach `quorum` within `timeout`, the `Manycast` filter returns the reply that generates the highest voting count.

In warm standby, all services in a group remain active to receive request messages. A `Manycast` filter sends a message to all services in a group, but only one service returns a reply. In this case, `backtracking` is not used. In cold standby, only one service in a group is active, and a `Manycast` filter sends a message to the service. If the service does not respond within `timeout`, the filter activates another service in the group and sends a message to the service. In cold standby, `backtracking` is not used.

`Anycast` is a variation of the hot standby policy in `Manycast`. It forwards a request message to only one destination in a group of replicated services. This filter is used to balance workload placed on services. `selection` defines how to choose a destination from multiple services; randomly, on round robin or on destination's priority basis. (the service with the highest priority in a group is selected.) If an `Anycast` filter fails to deliver a request message within `timeout`, it retries to forward the request message. `retry` specifies the maximum number of retries. If the `Anycast` filter fails the maximum number of retries, it returns an error message to the source of the request message.

`Logger` records transmission of each message whose priority value is higher than `priority` (Figure 6). `Router` routes an incoming message to one or more destinations with certain criteria. Since UML does not provide a means to define rules, the proposed profile has no facility to specify routing rules at design time. Supporting tools transform a `Router` to a skeleton source code (e.g., in Java) or a rule description (e.g., in XPath) that performs message routing. Developers are expected to complete the skeleton code/description.

In addition to the stereotypes for message transmission semantics, the proposed UML profile provides two other stereotypes to define the message processing seman-

tics in each connector: Validator and MessageFilter. Validator validates an incoming message against the message schema specified in its tagged-value schema, and forwards only valid messages. MessageFilter filters out incoming messages with a given criteria. Since UML does not provide a means to define rules, the proposed profile has no facility to specify message filtering rules at design time. Supporting tools transform a MessageFilter to a skeleton source code (e.g., in Java) or a rule description (e.g., in XPath) that performs message filtering. Developers are expected to complete the skeleton code/description.

3.3. Service

Service has three optional tagged-values: priority, timeout and redundancy (Figure 8). priority indicates the priority of each message that a service issues. The range of priority is from 0 to 255. (0 is the lowest and 255 is the highest.) Each Anycast filter uses priority to select the destination of each message, as described in Section 3.2. Each Connector also uses priority to order messages in a message queue when it has queueParameters (Section 3.1).

timeout specifies the timeout period (in millisecond) of each message that a service issues. If a message is not delivered to its destination within the time period, a connector discards the message.

redundancy specifies the number of runtime instances of a service. This tagged-value must be specified when Manycast or Anycast filters accesses a service.

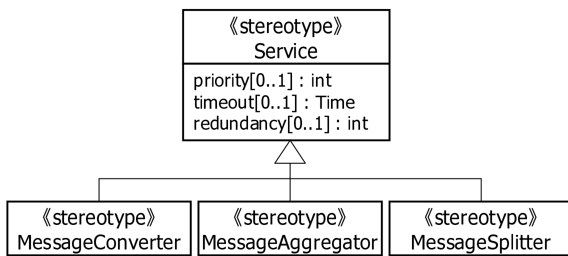

Figure 8. Tagged-values of Service

There are three special types of service, MessageConverter, MessageSplitter and MessageAggregator, to define the message processing semantics (Figure 8).

MessageConverter converts an incoming message with a given rule. Similar to Router, supporting tools transform a MessageConverter to a skeleton source code or rule description (e.g., in XSLT) that performs message conversions, and developers complete the skeleton code/description.

MessageSplitter divides an incoming message into multiple fragments with a certain rule. Similar to MessageConverter, supporting tools transform a MessageSplitter to a skeleton code or rule description that split messages, and developers complete the skeleton. In an example model shown in Figure 9, a MessageSplitter divides a request message (ReservationMsg) into two fragments, and sends them to different destinations (Hotel and AirCarrier). The destinations directly returns reply messages to Customer.

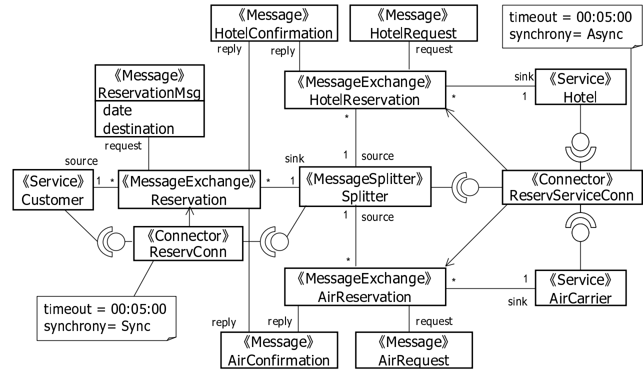

Figure 9. An Example of MessageSplitter

MessageAggregator combines multiple incoming messages to a single message. Figure 10 shows an example extending the model in Figure 1. In addition to OrderMsg, Buyer sends a message AuthReqMsg to ask the service Supervisors to authorize the order. Aggregator synchronizes and combines OrderMsg and AuthMsg (an authorization message from Supervisors), and it sends a combined message AuthedOrderMsg to Supplier.

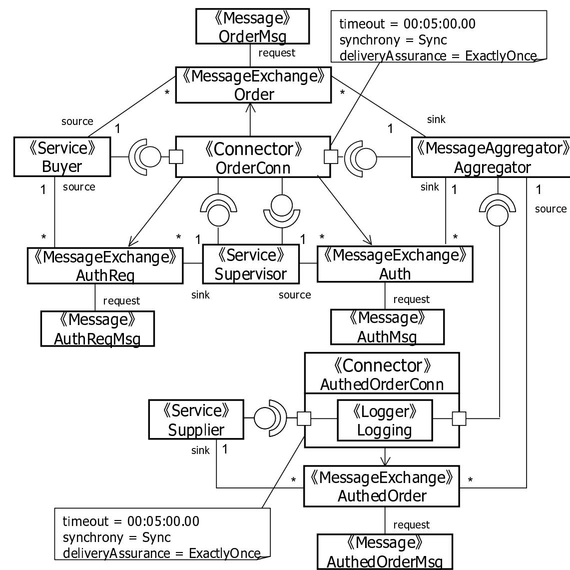

Figure 10. An Example of MessageAggregator

3.4. Message

Message has a mandatory tagged-value, schemaURI, and two optional tagged-values, priority and timeout (Figure 11). schemaURI identifies the schema of a message. The default value of schemaURI is message's quali-

fied name (a combination of a package name and message's name).

`priority` and `timeout` specify the priority and timeout period of a messages. `Connector` and `Service` also have `priority` and `timeout`. The precedence is that `Message`'s tagged-values override `Service`'s ones, and `Service`'s tagged-values override `Connector`'s ones.

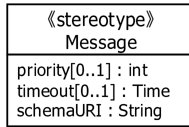

Figure 11. Tagged-values of Message

4. Application Development with the Proposed UML Profile

This section describes a model-driven development (MDD) tool, called Ark, which accepts a UML model designed with the proposed profile and transforms the model into a skeleton of application code (source code and deployment descriptor). Currently, Ark implements a transformation mapping between the proposed UML profile and MuleESB[2]. Ark takes a UML model in the XML Metadata Interchange (XMI) format. It has been tested with MagicDraw[3], a visual UML modeling tool that can serialize UML models to XMI. An input UML model (XMI file) is validated against the UML metamodel and the proposed profile (Figure 2), and transformed to Java programs and deployment descriptors for MuleESB.

A mapping rule between the proposed profile and MuleESB is implemented as a set of Velocity[4] transformation templates, which define how to transform UML model elements to application code elements. Table 3 summaries the mapping rule. Following this mapping rule, Ark transforms a stock quote example model (Figure 7) to application code shown in Figure 12.

Table 3. Mapping Rule between the Proposed UML Profile and MuleESB

The proposed UML profile	Mule ESB
≪Service≫	Java class with the same name
sink (Service's role)	Service's operations sending messages
source (Service's role)	Service's operations receiving messages
≪Message≫	Java class implementing `Serialiable` interface
synchrony	Different types of Mule ESB's operation used to send a message
timeout	An operation's parameter to specify message's timeout period
deliveryAssurance	A pair of interceptors to manage messages' IDs and timestamps
≪Multicast≫	Multicast filter provided by Mule ESB

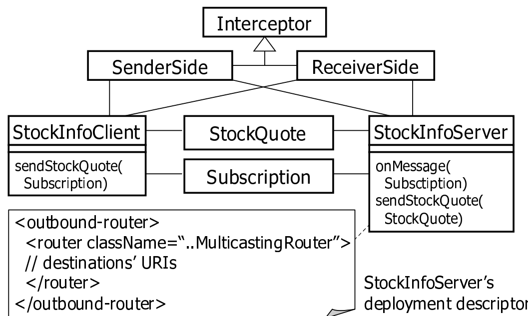

Figure 12. Generated Application Code

5. Related Work

There are several UML profiles proposed for SOA. [9] and [10] propose UML profiles to specify functional aspects in SOA. Both profiles are defined based on the XML schema of Web Service Description Language (WSDL). Each of the profiles provides a set of stereotypes and tagged-values that correspond to elements in WSDL, such as `Service`, `Port`, `Messages` and `Binding`[5]. Since WSDL is designed to define only functional aspects of web services, non-functional aspects are beyond of the scope of [9] and [10]. The proposed profile focuses on specifying non-functional aspects in SOA.

[11] proposes a UML profile to describe both functional and non-functional aspects in SOA. The stereotypes in this profile are generic enough to specify a wide range of applications. However, their semantics tend to be ambiguous. For example, the stereotypes for non-functional aspects include ≪`policy`≫, ≪`permission`≫ and ≪`obligation`≫, and ≪obligation≫ is intended to specify the responsibility of a service. [11] does not precisely define what developers have to (or can) specify with this stereotype and how to represent service responsibility (e.g., using natural languages or parameter values). In contrast, the proposed profile carefully defines its stereotypes and tagged-values in an unambiguous manner so that supporting tools can interpret and transform models to code.

[12] describes a UML profile for data integration in SOA. It provides data structures to specify messages so that users can build data dictionaries that maintain message data used in existing systems and new applications. This profile separates a non-functional aspect in data integration from

[2] A major open-source ESB implementation. http://mule.codehaus.org/
[3] http://www.magicdraw.com/
[4] A template-based code generation engine. jakarta.apache.org/velocity

[5] In WSDL, `Service` defines an interface of a web service. `Port` specifies an operation in a `Service`, and `Messages` defines parameters of a `Port`. `Binding` specifies communication protocols used by `Ports`.

functional aspects, and enables data integration in an implementation independent manner. The proposed profile focuses on non-functional semantics in message transmission, message processing and service deployment (e.g., service redundancy), rather than data integration.

[13] proposes a UML profile to facilitate dynamic service discovery in SOA. This profile provides a set of stereotypes (e.g., ≪uses≫, ≪requires≫ and ≪satisfies≫) to specify relationships among service implementations, service interfaces and functional requirements. For examples, users can specify relationships in which a service *uses* other services, and a service *requires* other services that *satisfy* certain functional requirements. These relationship specifications are intended to effectively aid dynamic discovery of services. The proposed profile and [13] focus on different issues in SOA. Service discovery is beyond of the scope of the proposed profile, and [13] does not consider non-functional aspects in message transmission, message processing and service deployment.

[14], [15] and [16] define UML profiles to specify service orchestration in UML and map it to Business Process Execution Language (BPEL) [17]. These profiles provide a limited support of non-functional aspects in message transmission, such as messaging synchrony. The proposed profile does not focus on service orchestration, but a comprehensive support of non-functional aspects in message transmission, message processing and service deployment.

There are several specifications and research efforts to investigate implementation techniques for non-functional aspects in SOA [17, 18, 19, 20, 21, 22]. Each specification and technique provides a means to implement non-functional requirements in, for example, performance, reliability and security and to enforce services to follow the requirements. Rather than providing specific implementations of non-functional aspects in SOA, the proposed UML profile is intended to provide a means for users to model and maintain non-functional aspects in an implementation independent manner so that they can be mapped on different specifications or implementation technologies.

6. Conclusion

This paper proposes a UML profile to graphically specify and maintain non-functional aspects in SOA in an implementation independent manner. This paper presents design details of the proposed profile, and describes how MDD tools can use it in service-oriented application development. As an example of MDD tools, this paper demonstrates a tool, called Ark, which accepts a UML model defined with the proposed profile and transforms the model into application code for MuleESB.

7. Acknowledgement

This work is supported in part by OGIS International, Inc. and Electric Power Development Co., Ltd.

References

[1] S. Vinoski. Integration with Web Services. *IEEE Internet Computing*, November/December 2003.

[2] Z. Zhang and H. Yang. Incubating Services in Legacy Systems for Architectural Migration. *Asia-Pacific Software Engineering Conference*, December 2004.

[3] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling. *Patterns: Service-Oriented Architecture and Web Services*. IBM Red Books, 2004.

[4] G. Lewis, E. Morris, L. Brien, D. Smith, and L. Wrage. SMART: The Service-Oriented Migration and Reuse Technique. Technical report, Software Engineering Institute, Carnegie Mellon University, September 2005.

[5] L. Fuentes and A. Vallecillo. An Introduction to UML Profiles. *The European journal for the Informatics Professional*, April 2004.

[6] Object Management Group. UML2.0 Super Structure Specification, October 2004.

[7] D. Chappell. *Enterprise Service Bus*. O'Reilly, June 2004.

[8] Java Community Process. UML Profile for EJB, May 2001.

[9] E. Marcos, V. de Castro, and B. Vela. Representing Web services with UML: A Case Study. *the Int'l Conference on Service Oriented Computing*, December 2003.

[10] IBM. *UML 2.0 Profile for Software Services*. developerWorks, April 2005.

[11] R. Amir and A. Zeid. A UML Profile for Service Oriented Architectures. *ACM OOPSLA Poster session*, 2004.

[12] M. Vokäc. Using a Domain-Specific Language and Custom Tools to Model a Multi-tier Service-Oriented Application–experiences and challenges. *ACM/IEEE Int'l Conference on Model Driven Engineering Languages and Systems*, October 2005.

[13] R. Heckel, M. Lohmann, and S. Thöne. Towards a UML Profile for Service-Oriented Architectures. *Workshop on Model Driven Architecture: Foundations and Applications*, 2003.

[14] T. Gardner. UML Modeling of Automated Business Processes with a Mapping to BPEL4WS. *ECOOP Workshop on OO and Web Services*, July 2003.

[15] IBM. *UML 1.4 Profile for Software Services with a Mapping to BPEL 1.0*. developerWorks, July 2004.

[16] Object Management Group. Business Process Definition Metamodel, January 2003.

[17] OASIS. Web Services Business Process Execution Language, April 2003.

[18] OASIS. Web Service Reliable Messaging, September 2004.

[19] OASIS. Web Service Reliability 1.1, November 2004.

[20] F. Baligand and V. Monfort. A Concrete Solution for Web Services Adaptability Using Policies and Aspects. *Int'l Conf. on Service Oriented Computing*, December 2004.

[21] G. Wang, A. Chen, C. Wang, C. Fung, and S. Uczekaj. Integrated Quality of Service (QoS) Management in Service-Oriented Enterprise Architectures. *IEEE Enterprise Distributed Object Computing Conference*, 2004.

[22] N. Mukhi, R. Konuru, and F. Curbera. Cooperative Middleware Specialization for Service Oriented Architectures. *ACM Int'l World Wide Web Conference*, 2004.