A Feature Modeling Support for Non-Functional Constraints 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.
San Mateo, CA 94404
oba@ogis-international.com

Abstract

It is important in Service Oriented Architecture (SOA) to separate functional and non-functional requirements for services because different applications use services in different non-functional contexts. In order to maximize the reusability of services, a set of constraints (e.g., dependency and mutual exclusion constraints) among non-functional requirements tend to be complicated to maintain. Currently, those non-functional constraints are informally specified in natural languages, and developers need to ensure that their applications satisfy the constraints in manual and ad-hoc manners. This paper proposes a model-driven development framework, through the notion of feature modeling, to explicitly and graphically specify non-functional constraints in SOA. The proposed framework allows developers to validate non-functional constraints in their applications in an automatic and consistent way. This paper also describes how the proposed framework is implemented and effectively used for service-oriented application development.

1. Introduction

It is important in Service Oriented Architecture (SOA) to define the non-functional requirements (NFRs; e.g., security and fault tolerance) of services separately from their functional requirements because different applications use services in different non-functional contexts. For example, an application may send signed and encrypted messages to a service when the messages travel to the service through third-party intermediaries, in order to prevent the intermediaries from maliciously sniffing or altering the messages. Another application may send plain messages to the service when the service is hosted in-house. The separation of functional and non-functional requirements improves the reusability of services in different non-functional contexts.

In order to maximize the reusability of services, a set of constraints (e.g., dependency and mutual exclusion constraints) among NFRs tend to be complicated and hard to maintain because their granularity becomes finer and their number grows. Currently, those non-functional constraints are informally specified in natural languages, and developers need to ensure that their applications satisfy the constraints in manual and ad-hoc manners [1–5]. It is tedious and error-prone to consistently validate and enforce non-functional constraints in large-scale applications because a large number of NFRs and their combinations can exist.

This paper proposes a new model-driven development (MDD) framework, through the notion of feature modeling [6], to explicitly and graphically model a series of non-functional constraints in SOA (Figure 1). The framework consists of (1) a feature model that defines non-functional constraints in SOA, (2) a Unified Modeling Language (UML) profile to specify NFRs in SOA, called UP-SNFRs [1], and (3) an MDD tool, called Ark, which generates application code (program code and deployment descriptors) according to a configuration (or instance) of the proposed feature model and a UML model defined with UP-SNFRs. Feature modeling is a simple yet powerful method to explicitly model the constraints among application's *features* (e.g., functionalities and configuration policies) [6]. By modeling an NFR as a feature, the proposed framework allows developers to consistently validate non-functional constraints in service-oriented applications. Ark automatically enforces non-functional constraints in applications by transforming a feature configuration to application code with UP-SNFRs (Figure 1).

As shown in Figure 1, all modeling artifacts in this work are maintained with the metameta model of the Eclipse Modeling Framework (EMF; http://www.eclipse.org/emf/). UP-SNFRs is defined as an extension to the UML metamodel, and the proposed feature model is defined with the feature metamodel in fmp (http://gp.uwaterloo.ca/fmp/) [7].

2. An Overview of UP-SNFRs

This section gives an overview of UP-SNFRs, which is the authors' prior work [1]. UP-SNFRs is designed around

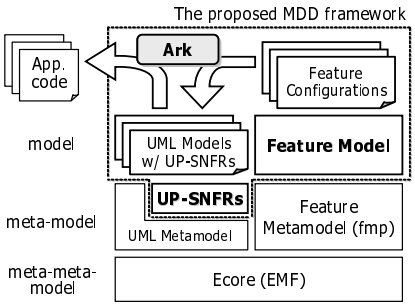

Figure 1: An Architectural Overview of the Proposed MDD Framework

two major concepts: *services* and *connections* between services. Each service encapsulates the function of a certain element in a system (e.g., subsystem). Each connection defines how services are connected with each other and how messages are exchanged through the connection.

Figure 2 shows an example model defined with UP-SNFRs. It illustrates an order and inventory management application in which a buyer places purchase orders and inventory inquiries. In this example, three services (`Buyer`, `OrderMgr` and `InventoryMgr`) exchange messages. Each service is represented by a class stereotyped with ≪`service`≫ or ≪`accessControlledService`≫. ≪`accessControlledService`≫ represents a special type of service that enforces an access control policy. (It is a stereotype extending the stereotype ≪`service`≫.) The tagged-value `securityTokens` is mandatory for this stereotype to specify security tokens (or certificates). The security tokens are used to authenticate entities (e.g., services) that access a service.

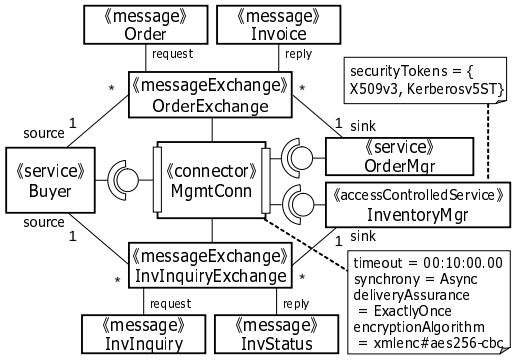

Figure 2: An Example Model with UP-SNFRs

The services in Figure 2 exchange four types of messages (`Order`, `Invoice`, `InvInquiry` and `InvStatus`), each of which is stereotyped with ≪`message`≫. Each pair of a request and reply messages is represented by a class stereotyped with ≪`messageExchange`≫. For example, a pair of `Order` (request) and `Invoice` (reply) messages is represented by `OrderExchange`.

≪`connector`≫ represents a connection that transmits messages between services. In this example, messages are delivered through the connector `MgmtConn`. Every message exchange is bound with a connector in order to specify which connector is used to deliver messages. A connector has a provided interface (represented as a ball icon) and a required interface (represented as a socket icon). Services use the provided and required interfaces to send and receive messages, respectively. In Figure 2, `Buyer` sends an `InvInquiry` message to `InventoryMgr`.

Each connector can have multiple tagged-values to specify a set of message transmission and processing semantics. Figure 3 shows the definition of the ≪`connector`≫ stereotype in UP-SNFRs. It is defined as the `Connector` metaclass that extends the UML metamodel [1]. Each field in `Connector` corresponds to a tagged-value. In Figure 2, the connector `MgmtConn` specifies the timeout of message transmissions (10 minutes), synchrony of message transmissions (asynchronous), assurance level of message delivery (exactly once), and encryption algorithm for messages (Advanced Encryption Standard).

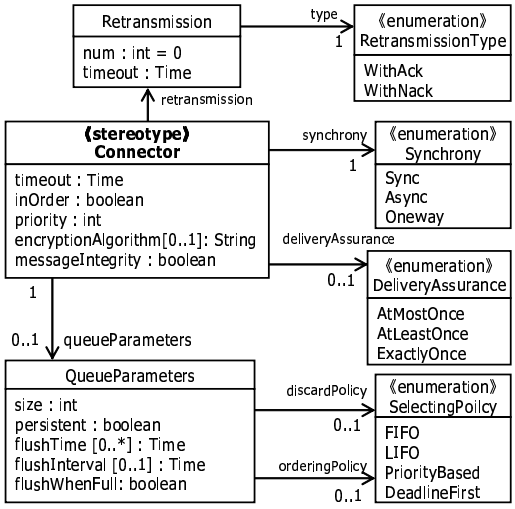

Figure 3: Definition of Connector

3. A Feature Model for Non-Functional Constraints in SOA

A series of constraints exist among the NFRs defined in UP-SNFRs. For example, when the synchrony of message transmissions is configured as asynchronous (`synchrony=Async`; see Figure 3), a timeout period should be specified with the `timeout` tagged-value. A message retransmission policy requires specifying the maximum number of retransmissions and its type: Ack-base or Nack-base (Figure 3). If it is configured as Ack-

based, a timeout period need to be specified for retransmissions. When a connector is specified as a messaging queue and configured to discard messages when it overflows (`flushWhenFull=true`; see Figure 3), `discardPolicy` should specify the order of messages to discard. Currently, UP-SNFRs defines these constraints in English, not as model elements [1]. Therefore, application developers are supposed to make sure that their application designs satisfy the constraints.

A feature model describes a set of all possible valid configurations as hierarchies of features that describe different kinds of constraints, e.g., some features are mandatory and some are optional or alternative [6]. It gives a clear view of features that a system can support and their constraints even to non-programmers. It is useful for scoping the product, i.e., deciding which features should be supported by the product and which feature should not, and encourages the requirements to be much clear in the early stage of the development process (e.g., requirement phase). By reducing the possibility of changes in requirements in the late stage of the development process, feature model gives a positive impact on application developments since the changes in requirements in the late stage can cause a huge number of modifications in existing (already implemented) application designs and code. Moreover, application developers can use feature configurations as input to an automated product derivation process (e.g., automatic model/code generation) to improve the productivity of the application development.

Figure 4 shows the proposed feature model describing NFRs in a UML profile, and Figure 5 is an example feature configuration (i.e., an instance of the proposed feature model). The proposed feature model has following features, i.e., `Message Priority`, `Synchrony`, `Retransmission`, `Timeout`, `Delivery Assurance`, `Message Encryption`, `Access Control`, `Message Integrity`, `In Order Transmission`, `Logging`, `Message Validator`, `Message Routing`, `Multicast`, `Manycast`, `Anycast` and `Queue`.

`Message Priority` and `Timeout` are optional features to specify the priority and the timeout period of messages. White circle symbols represent optional features. Each feature may have its type. For example, the type of the `Message Priority` feature is `Integer`. When a typed feature is selected in a feature configuration, its value (e.g., integer value) should be specified. In Figure 5, the value of the `Timeout` feature is ten minutes.

`Synchrony` specifies the timeout period of messages. Black circle symbols in a feture model represent mandatory features. A feature can have multiple subfeatures. For example, the `Synchrony` feature has three subfeatures: `Sync`, `Async` and `Oneway`. A fork symbol with a white sector represents an *exclusive-or* relationship among subfeatures. In Figure 4, when the `Synchrony` feature is selected, one of subfeatures (i.e., `Sync`, `Async` or `Oneway`) should be selected at the same time.

`Retransmission` specifies a policy related to message retransmissions. Message retransmission requires specifying the number of retransmissions and its type: Ack-base or Nack-base. If it is configured to be Ack-based, a timeout period of retransmissions is also need to be specified.

`Delivery Assurance` specifies the assurance level of message delivery. Three different semantics are defined. `At Least Once` means that a connector retries delivering a message until its destination receives the message. However, the message may be delivered to its destination more than once. `At Most Once` 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. A fork symbol with a black sector in a feature model represents an *or* relationship among subfeatures. The `Delivery Assurance` feature has two subfeatures, and one or two of them should be selected when `Delivery Assurance` is selected. When cardinality (i.e., the number of subfeatures to be selected) is omitted, the default cardinality, one to the number of subfeatures, will be used [8]. (Figure 4 shows the default cardinality explicitly.)

`Message Encryption` and `Access Control` specify a transport-level encryption algorithm and security tokens (or certificates) to access services respectively.

`Message Integrity` specifies whether to ensure the message integrity. Messages are checked whether changed during their transmission when this feature is selected. `In Order Transmission` specifies 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.

`Logging` specifies whether to log messages of which priorities are greater than `Priority` during their transmission. `Message Validation` validates transmitted messages against the message schemata specified in `Messages` subfeature. `Message Routing` specifies whether to use a contents-based message routing mechanism. Although the proposed feature model has no facility to specify routing rules at design time, a supporting tool generates a skeleton source code (e.g. in Java) or rule description (e.g. in XPath) that performs message routings by leveraging underlying middleware technologies.

`Multicast` specifies to transmits a message to multiple destinations (services) simultaneously (one-to-many message exchange) to improve the efficiency of message transmissions.

`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). `GroupSize` subfeature specifies how many services are deployed as a group. `Service Standby` specifies the operation of replicated

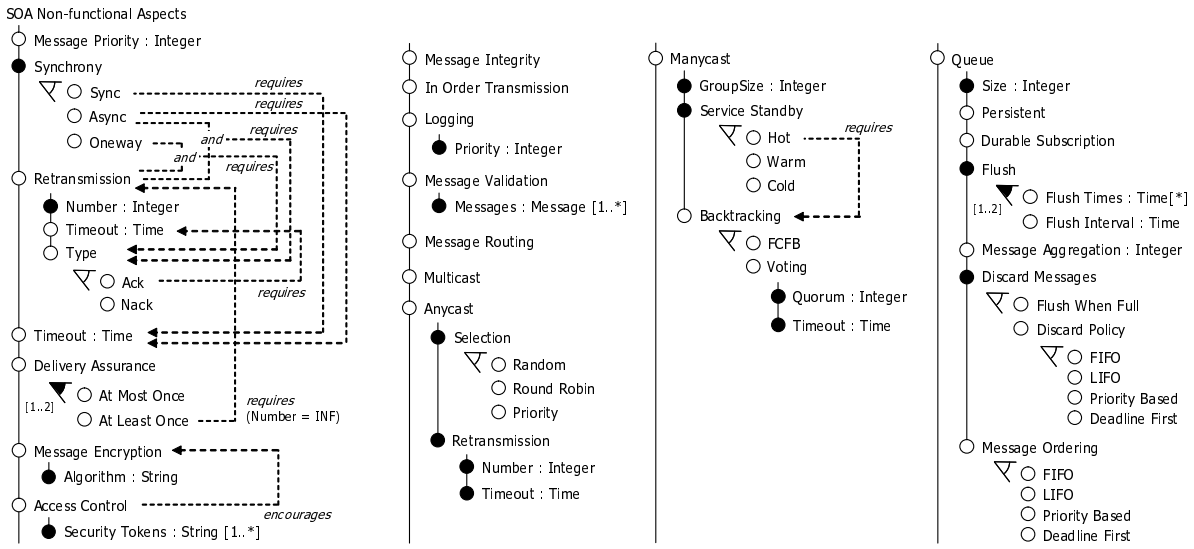

Figure 4: The Proposed Feature Model for Non-functional Constraints in SOA

services: hot standby, warm standby or cold standby. In hot standby, all services in a group remain active to receive request messages. A message is transmitted to all services in a group. Only one reply message is returned 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, the first reply among replies from destination services is returned. When `Voting` is selected, a voting process is performed. A middleware 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 reply that generates the highest voting count is returned.

In warm standby, all services in a group remain active to receive request messages. A message is transmitted 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 message is transmitted to the service. If the service does not respond within timeout, another service in the group is activated and a message is retransmitted to the service. In cold standby, `Backtracking` is not used.

`Anycast` is a variation of the hot standby policy in Manycast. A request message is forwarded to only one destination in a group of replicated services. This feature 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 a service does not respond within `Timeout`, the request message is retransmitted to another service. `Number` specifies the maximum number of retransmissions.

`Queue` is used 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. `Persistent` 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. `Flush Time` and `Flush Interval` under the `Flush` subfeature specify when and how often a queue flushes messages, respectively. `Message Aggregation` specifies a number of messages to send at once in an aggregated form. `Discard Messages` has two subfeatures: `Flush When Full` and `Discard Policy`. Either one should be selected because of a *xor* relationship among them. When `Flush When Full` is selected, queued messages are flushed from a queue to their destinations when the queue overflows. When `Discard Policy` is selected, the overflowing queue discards a message according to one of subfeatures: 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 subfeatures of `Discard Policy`. `Message Ordering` specifies how to order messages in a queue: FIFO, LIFO, highest-priority-first or earliest-deadline-first.

In addition to hierarchies of features, additional relationships can be specified between (sub)features. In Figure 4, several `requires` relationships are specified. For example, when `Async` and `Retransmission` features are selected, `Type` subfeature in `Retrasmission` and `Timeout` feature should be selected in the same feature configu-

ration. The `At Least Once` subfeature of `Delivery Assurance` requires the feature `Retransmission` selected and its `Number` subfeature configured to be infinite.

Also, `encourages` and `discourages` relationships are newly introduced in this research project. `encourages` is a relationship which is similar but weaker than `requires`. It has no mandatory power, but endorses to use the referred features at the same time. For example, in Figure 4, the `Access Control` feature *encourages* to select the `Message Encryption` feature at the same time. Selecting `Message Encryption` with `Access Control` makes systems much secure because transmitting security tokens via unsecured connections makes systems vulnerable, but in-house services may not require message encryption but need access control for the purpose of audit. `discourages` relationship works in direct contrast to `encourages` relationship, and it discourages to use referred features. `encourages` and `discourages` relationships facilitate the decision-making process in designing applications.

Figure 5 shows a sample feature configuration of the feature model in Figure 4. In this feature configuration, several features (e.g., `Async` subfeature and `Retransmission` feature) are selected. A supporting tool resolves the constraints between (sub)features. (e.g., the `Sync` and `Oneway` subfeatures are automatically deselected when the `Async` subfeature is selected.) Moreover, it reports missing properties (e.g., an alert is shown when the value of the `Timeout` is not configured even though the feature is selected.) and existence of encouraged features (e.g., a message is shown to encourage to select `Message Encryption` when `Access Control` is selected.) to application developers.

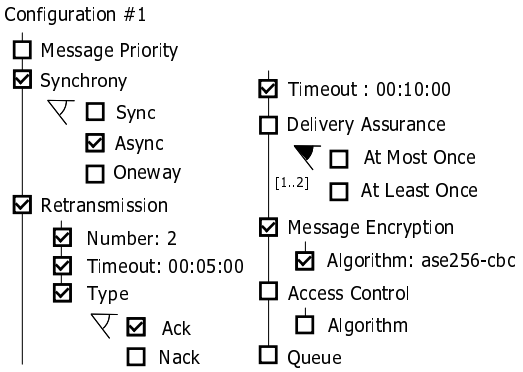

Figure 5: An Example Feature Configuration

4. Application Development with Ark

This section describes a MDD tool, called Ark. Ark accepts feature configurations and a UML model (a class or composite structure diagram in UML 2.0) describing an application design, and transforms them into a UML model designed with a UML profile for NFRs in SOA (Figure 6). Furthermore, Ark transforms the generated UML model into a skeleton of application code (e.g., source code and deployment descriptor) running on certain middleware technologies.

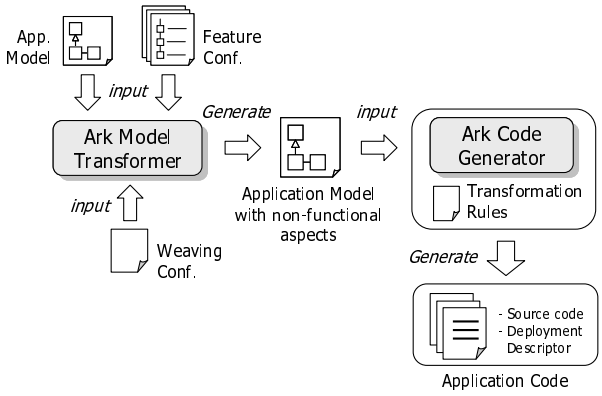

Figure 6: Development Process with Ark

The proposed feature model is defined on fmp [7], which is a feature modeling tool implemented on EMF. The tool is extended to support `encourages` and `discourages` relationships (Section 3). Figure 7 shows a feature configuration in a feature modeling tool. In this example, messages are shown in the bottom notifying the existence of `encourages` relationship and a missing value in the `Timeout` feature. Feature model and a UML profiles are defined on EMF (Figure 1), and Ark uses APIs provided by fmp and Eclipse UML2 (http://www.eclipse.org/uml2/) to read and transform models.

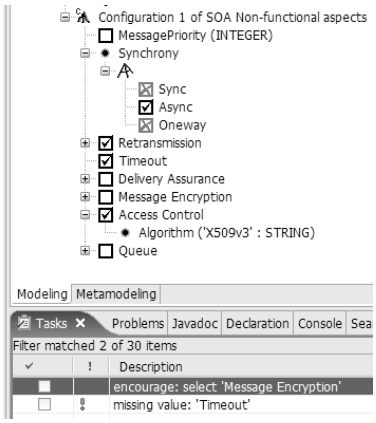

Figure 7: A Screenshot of a Feature Modeling Tool

4.1. An Example Application

Figure 8 shows an example UML model which Ark accepts. It illustrates an order processing application in which

OrderMgr service is contained in the `SalesMgmtSystem` package.

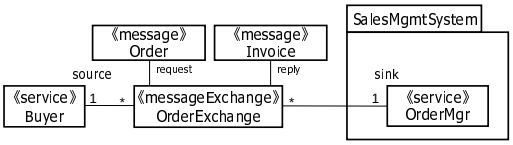

Figure 8: An Example Input UML Model

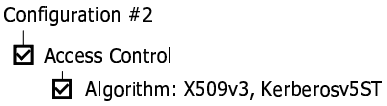

Figure 9: An Example Subset Feature Configuration

Ark accepts arbitrary number of feature configurations as its input, even a feature configuration which is based on a subset of the proposed feature model. In this example, two feature configurations in Figure 5 and 9 are applied to the application design shown in Figure 8. The feature configuration in Figure 9 configures only a subset of the proposed feature model. Ark allows application developers to specify which feature configurations to be applied to which model elements in an input application design in the form of a weaving configuration (Figure 6). The following is an example of a weaving configuration.

```
<weaving>
  <featureconfiguration name="Configuration #1">
    <model pattern=".*"/>
  </featureconfiguration>
  <featureconfiguration name="Configuration #2">
    <model pattern="SalesMgmtSystem::.*"/>
    <model pattern="ProductionMgmtSystem::.*"/>
  </featureconfiguration>
</weaving>
```

As illustrated, a weaving configuration contains a set of `featureconfiguration` tags. Each of them specifies which feature configuration is applied to which model elements using its `name` attribute and `model` tags respectively. A `model` tag has the attribute `pattern` specifying names of model elements in regular expression. Ark takes a weaving configuration, and applies feature configurations from the top to the bottom to an input UML model. In this example, the `configuration #1` feature configuration is applied to every model elements in an input UML model first (`.*` matches arbitrary model elements), and then the `configuration #2` feature configuration is applied to model elements contained in the `SalesMgmtSystem` and `ProductionMgmtSystem` packages. (Double colon '`::`' is a namespace separator in UML.)

By applying the feature configurations, Ark transforms the input UML model in Figure 8 into a UML model with a UML profile in Figure 10. The following pseudo code shows a model transformation process.

```
transform( UMLElement e, WeavingConf weavingConf ){
  if (e does not match to weavingConf)
    return;

  if (e is stereotyped with <<messageExchange>>)
    if (there is no corresponding connector)
      create a new connector;
    endif

    if (a feature configuration has 'Message Priority')
      configure connector's 'priority' tagged-value
    if (a feature configuration has 'Synchrony')
      configure connector's 'synchrony' tagged-value
    ...

  else if (e is stereotyped with <<service>>)
    if (a feature configuration has 'AccessControl')
      replace <<service>> with <<accessControlledService>>
      configure service's 'securityTokens' tagged-vale
}
```

In the transformation, a connector (`OrderExchangeConn`) is generated and several tagged-values are configured. In a UML profile, every message exchange is bound with a connector which specifies message transmission/processing semantics (see details in Section 2). Most features in the proposed feature model are mapped into connectors. The `Access Control` feature is mapped into the ≪`accessControlledService`≫ stereotype. As illustrated, `OrderMgr` is stereotyped with ≪`service`≫ in the input model (Figure 8), and the stereotype is replaced with ≪`accessControlledSservice`≫ in the output model and its tagged-value `securityTokens` is configured (Figure 10).

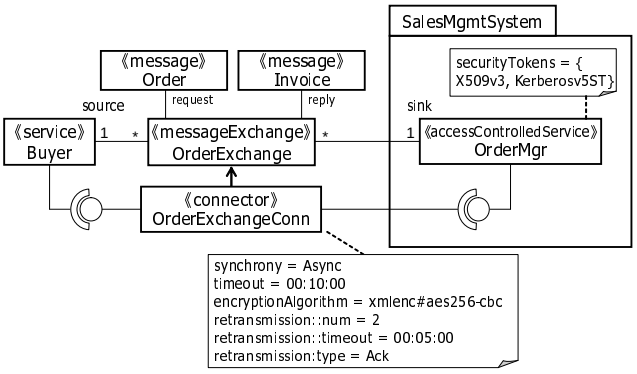

Figure 10: An Example Output UML Model

As demonstrated, the feature modeling technique gives a clear view of features and their constraints, and it allows application developers to design NFRs without knowing the details of a UML profile. (e.g., which stereotypes and tagged-values to be used to configure certain NFRs.) Moreover, by leveraging a weaving configuration, Ark separates the application design and its NFRs well and improves the reusability of feature configurations. For example, a feature configuration can be applied to all elements in an input UML model as a default setting, or can be applied to only specific elements (e.g., elements in a certain package)

by only changing a weaving configuration. It makes easy to configure applications in typical situations (e.g., services hosted in-house, or accessed via the Internet) by reusing existing feature configurations.

4.2. An Extended Example Application

Figure 11 shows an example UML model extending the model in Figure 8. In this example, an inventory management system (in package `InventoryMgmtSystem`) is connected to an order processing application. `OrderMgr` service exchanges `Purchasing` message with `InventoryMgr`, and the `InventoryMgr` exchanges `Shipping` message with `DistributionMgr`.

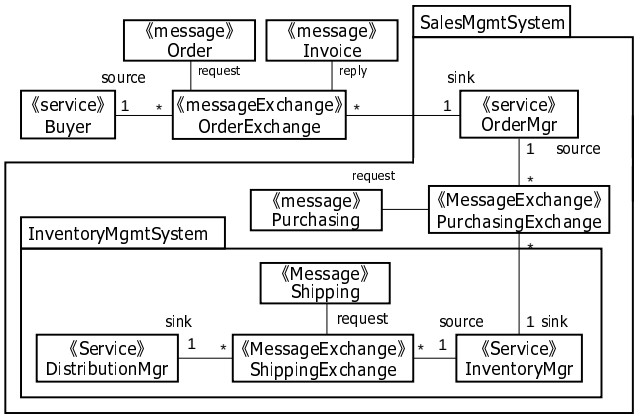

Figure 11: An Example Input UML Model

New feature configuration is defined as Figure 12. The feature configuration intends to be applied to in-house services which exchange plain messages in a synchronous manner via unsecured (the `Message Encryption` feature is no selected) but reliable (both `At Most Once` and `At Least Once` features are selected) connections. The following is a weaving configuration.

```
<weaving>
  <featureconfiguration name="Configuration #1">
    <model pattern=".*"/>
  </featureconfiguration>
  <featureconfiguration name="Configuration #3">
    <model pattern="SalesMgmtSystem::.*"/>
    <model pattern="ProductionMgmtSystem::.*"/>
  </featureconfiguration>
  <featureconfiguration name="Configuration #2">
    <model pattern=".*::OrderMgr"/>
  </featureconfiguration>
</weaving>
```

The `Configuration #1` feature configuration is applied to all model elements as a default, and then `Configuration #3` is applied to model elements in the package `SalesMgmtSystem`, which supposed to be hosted in-house. Finally, `Configuration #2` is applied to only the `OrderMgr` service because it exchanges messages with the service hosted in outside (`Buyer`) and required to have an access control. Ark accepts the input UML model (Figure 11), feature configurations (Figure 5, 9 and 12) and the weaving configuration, and generates a UML model illustrated in Figure 13. (Model elements outside the `SalesMgmtSystem` package are omitted. They are the same as ones illustrated in Figure 10.) Since feature configuration `Configuration #3` overrides `Configuration #1`, tagged-values in `ShippingExchangeConn` and `PurchasingExchangeConn` are different from ones in `OrderExchangeConn`. (`timeout` and `type` tagged-values are identical because corresponding features are not configured in `Configuration #3`.)

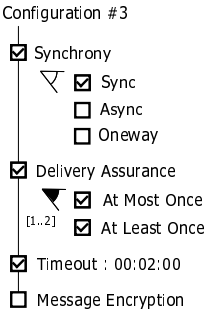

Figure 12: A Feature Configuration for In-house Services

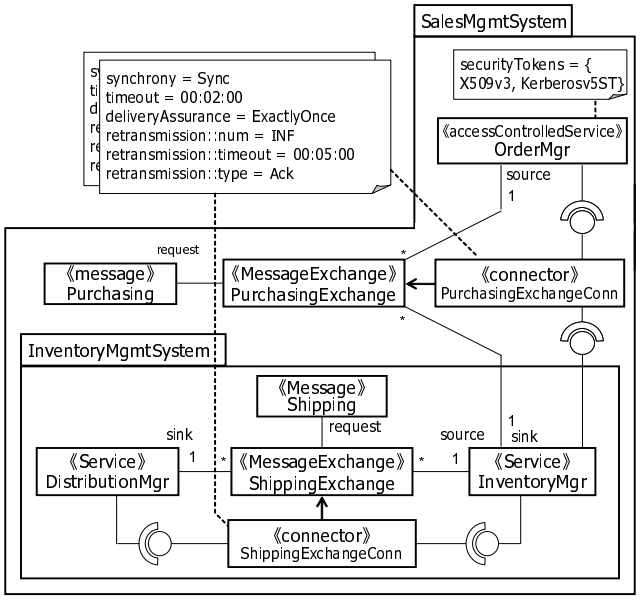

Figure 13: An Output UML

As demonstrated, a weaving configuration improves the separation of concerns between application designs and their NFRs, and enables the two different aspects to evolve independently.

4.3. Ark Code Generator

Ark code generator accepts a UML model designed with a UML profile and transforms the model into a skeleton of application code (Figure 6). Currently, Ark code generator implements a transformation mapping between a UML profile and Mule ESB[1] and GridFTP[2]. Ark takes a UML model in the XML Metadata Interchange (XMI) format, validates against the UML metamodel and a UML profile, and transforms to Java programs and deployment descriptors for Mule ESB. Ark Code Generator is implemented based on openArchitectureWare[3], a model transformation engine.

Table 1 shows some of the Java classes and deployment descriptors that Ark generates from a UML model with a UML profile for SOA NFRs when Mule ESB is selected as middleware to operate applications. Ark maps a UML class stereotyped with ≪`message`≫ to a Java class that has the same class name and the same data fields. The Java class implements the interface `Serializable`. This is required to implement messages exchanged with Mule ESB.

Table 1: Mapping Rule between a UML Profile and Mule ESB

A 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

A UML class stereotyped with ≪`service`≫ is mapped to a Java class that has the same class name and the same data fields. Ark inserts several operations to the Java class, depending on whether its association role is `source`/`sink` against a message exchange. The operations are used to send and receive messages: `_sendX()` to send messages where `X` references the name of a message exchange, and `receiveX()` to receive messages. For example, generated code from Figure 13, `InventoryMgr` class has `_sendShippingExchange()` and `receivePurchasingExchange()` to send `Shipping` and receive `Purchasing` messages respectively.

UML classes stereotyped with ≪`messageExchange`≫ and ≪`connector`≫ are not mapped to particular Java classes. The message transmission/processing semantics specified in a UML model is implemented in Java classes of message sender and destination. For example, in Figure 13, an `InventoryMgr` sends a `Shipping` message to a `DistributionMgr` synchronously. Therefore, Ark generates program code to send the message synchronously using Mule ESB's API[4], and embeds the code in `_sendShippingExchange()` of `InventoryMgr`. Ark also generates program code to handle timeout using Mule ESB's API, and embeds the code in `_sendShippingExchange()` of `InventoryMgr`.

5. Related Work

Several modeling languages (e.g., UML profiles and domain-specific languages) have been proposed to specify NFRs in SOA, such as security, data integration, service discovery and service orchestration [1–5]. However, they do not focus on modeling a series of constraints among those NFRs. This work is the first attempt to allow developers to explicitly model the non-functional constraints in SOA and automatically enforce the constraints in their applications.

The notion of feature modeling has been used to, for example, configure the non-functional policies in embedded operating systems (e.g., concurrency and interruption policies) [10], configure the functionalities of Eclipse plugins (e.g., multi-windows) [11] and select services in PBX systems (e.g., call request and call forwarding services) [12]. This work is the first attempt to leverage feature modeling for managing the non-functional constraints in SOA. Moreover, the proposed feature model supports new model elements that the exiting feature models do not have, such as `encourages` and `discourages` relationships. This work also investigates a new mechanism to weave feature configurations to different parts of a UML model. No exiting work has worked on this feature weaving mechanism.

Feature-based model templates are designed to transform feature configurations to UML models [13]. Each template is a UML class or activity diagram that defines a *presence condition* for each model element (e.g., class, association and action). A presence condition specifies when a corresponding model element appears in a UML diagram. For example, a class may have its presence condition `Async & Ack` in a class diagram template. The template generates the class in an output class diagram if the template accepts an input feature configuration that selects both `Async` and `Ack` features. This way, NFRs scatter both in feature configurations and model templates. This makes it complicated

[1] A major open-source ESB implementation. http://mule.codehaus.org/

[2] An extension to FTP for transmitting files of large size [9].

[3] http://www.openarchitectureware.org/

[4] Mule ESB provides three different APIs to send messages in synchronous, asynchronous and oneway (non-blocking) manner.

to maintain NFRs; the changes in NFRs (e.g., addition, removal or customization of NFRs) always require developers to change both a feature model and model templates. In contrast, the proposed framework modularizes NFRs in feature configurations; NFRs never appear in input UML models. As discussed in Section 1, the clear separation of functional and non-functional requirements is critical in SOA, in which NFRs changes more often in application lifecycle than functional requirements [14].

POSAML (Pattern Oriented Software Architecture Modeling Language) is a domain specific language to visually configure NFRs in realtime CORBA middleware (e.g., concurrency and queuing) [15]. In POSAML, non-functional constraints are specified in a textual manner with the Object Constraint Language (OCL) [16]. Although OCL provides higher expressiveness as a constraint specification language than feature models, the proposed framework employs feature models by trading visual intuitiveness for expressiveness. In addition, a feature weaving mechanism is not available in POSAML. In POSAML, a single feature configuration is globally applied to the entire input UML model. In contrast, the proposed framework allows developers to flexibly customize which feature configuration is applied to which model elements in an input UML model.

6. Conclusion

This paper proposes a new MDD framework to explicitly and graphically model the constraints (e.g., dependency and mutual exclusion constraints) between non-functional aspects in SOA. Through the notion of feature modeling, the proposed framework allows developers to consistently validate non-functional constraints in service-oriented applications. An MDD tool, called Ark, automatically enforces non-functional constraints in applications by transforming a feature configuration to application code with the use of a UML profile for SOA. This frees developers from manual consistency checks between their models and non-functional constraints.

7. Acknowledgement

This work is supported in part by OGIS International, Inc.

References

[1] H. Wada, J. Suzuki, and K. Oba. Modeling Non-Functional Aspects in Service Oriented Architecture. *IEEE Int'l Conference on Services Computing*, September 2006.

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

[3] G. Ortiz and J. Hernández. Toward UML Profiles for Web Services and their Extra-Functional Properties. *IEEE Int'l Conference on Web Services*, September 2006.

[4] L. Wang and L. Lee. UML-based Modeling of Web Services Security. *IEEE European Conference on Web Services Poster session*, 2005.

[5] Y. Nakamura, M. Tatsubori, T. Imamura, and K. Ono. Model-Driven Security Based on a Web Services Security Architecture. *IEEE Int'l Conference on Services Computing*, July 2005.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[7] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-in for Eclipse. *ACM OOPSLA Workshop on Eclipse Technology eXchange*, 2004.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice*, 2005.

[9] W. Allcock, J. Bresnahan, R. Kettimuthu, C. Dumitrescu M. Link, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. *Super Computing*, November 2005.

[10] D. Lohmann, F. Scheler, W. S Preikschat, and O. Spinczyk. PURE Embedded Operating Systems - CiAO. *IEEE Int'l Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.

[11] M. Antkiewicz and K. Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. *ACM/IEEE Int'l Conference on Model Driven Engineering Languages and Systems*, October 2006.

[12] K. Kang, S. Kim, J. Lee, and K. Lee. Feature-oriented engineering of PBX software. *Asia-Pacific Software Engineering Conference*, December 1999.

[13] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach based on Superimposed Variants. *Int'l Conference on Generative Programming and Component Engineering*, September 2005.

[14] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, and R. Shah. *Service-Oriented Architecture (SOA) Compass : Business Value, Planning, and Enterprise Roadmap*. IBM Press, October 2005.

[15] Dimple Kaul, Arundhati Kogekar, Aniruddha Gokhale, Jeff Gray, and Swapna Gokhale. POSAML: A Visual Modeling Framework for Middleware Provisioning. *Hawaiian Int'l Conf. on System Sciences*, January 2007.

[16] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, second edition, 2003.