Propagation of Incremental Changes to Performance Model due to SOA Design Pattern Application

Nariman Mani, Dorina C. Petriu, Murray Woodside
Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, Ontario, Canada
{nmani | petriu | cmw}@sce.carleton.ca

ABSTRACT

Design patterns for Service Oriented Architecture (SOA) provide solutions to architectural, design and implementation problems, involving software models in different layers of a SOA design. For performance analysis, a performance model can be generated from the SOA design and used to predict its performance. The impact of the design patterns is also reflected in the performance model. It is helpful to be able to trace the causality from the design pattern to its predicted performance impact. This paper describes a technique for automatically refactoring a SOA design model by applying a design pattern and for propagating the incremental changes to its LQN performance model. A SOA design model is expressed in UML extended with two standard profiles: SoaML for expressing SOA solutions and MARTE for performance annotations. The SOA design pattern is specified using a Role Based Modeling Language (RBML) and their application is automated using QVT-O. Automated incremental transformations are explored and evaluated for effectiveness on a case study example.

Categories and Subject Descriptors

H.3.4 **[Systems and Software]**: Performance evaluation (efficiency and effectiveness)

General Terms

Performance, Design, Experimentation, and Verification.

Keywords

Software performance, service-based systems, SOA pattern, model change, change propagation, LQN.

1. INTRODUCTION

Service Oriented Architecture (SOA) is an innovative software architectural paradigm for developing and deploying applications as a set of reusable composable units called services, that can be collectively and repeatedly used for the realization of specific goals [1]. In Model Driven Engineering (MDE), the performance of a SOA design can be evaluated using model transformations to generate a performance model (hereafter called *PModel*) of the SOA system from its software design model (hereafter called *SModel*) extended with performance annotations. In our work, the initial SModel to PModel transformation is performed with the PUMA transformation chain [2], previously developed in our research group, which requires the generation of an intermediate model Core Scenario Model (CSM), as shown in Figure 1.(A). CSM captures the essence of performance specification and estimation as expressed by SModel annotations, and strips away the design detail which is irrelevant to performance analysis. During the initial transformation, trace links expressing the mapping between the SModel and the corresponding PModel elements are created [3].

In this paper we propose to use performance evaluation to screen candidate SOA pattern applications, after an initial SOA design has been modeled and its PModel derived for the first time. We assume that the candidate patterns are selected by the designer in order to improve a certain quality of the design (such as maintainability, robustness, performance, security, etc.). However, the logic behind the pattern selection is not addressed in this paper, being left for future work. The focus of this paper is on the automatic application of a SOA design pattern, followed by the incremental propagation of SModel changes directly to the PModel, as illustrated in Figure 1.(B) (Model modifications are denoted using the "Δ" symbol). SOA design patterns describe generic solutions for different architectural, design and implementation problems[1, 4]. Changes due to the application of such patterns may impact performance and other non-functional properties either positively or negatively.

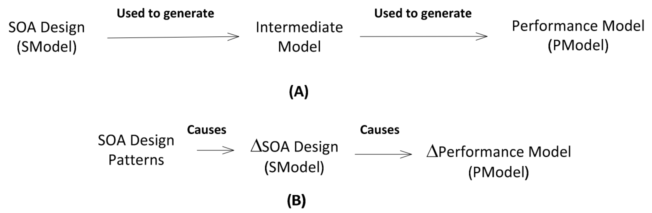


Figure 1 : (A) Initial transformation of SModel to PModel (B) Propagation of SModel changes due to design pattern application directly to PModel

Reconstructing the entire intermediate and performance models from scratch after a pattern application requires a lot of effort. To address this issue, we propose here an incremental approach that allows investigating the impact of changes due to design patterns on the system performance in a time and cost effective way, by identifying the affected performance model elements and parameters without re-constructing the entire performance model. This enables incremental studies of numerous design alternatives when applying a large number of SOA design patterns.

As already mentioned, we use the standard UML profile Service Oriented Architecture Modeling Language (SoaML) [5] to represent the design of a SOA system. SoaML defines extensions to UML 2 to support a range of modeling requirements for service-oriented architectures [6].

Another standard UML profile used in our approach is MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [7], which bridges the gap between the software and the performance domains. The MARTE annotations help us not only in transforming the SModel into a PModel, but also in propagating the modifications due to the application of design patterns.

For studying the impact of design patterns on the system and its performance, the SOA design patterns need to be formally specified. In this paper, we use the Role Based Modeling Language (RBML) [8] to formally define two parts of a SOA design pattern: the set of SModel elements and their relationships that represents the *problem* addressed by the pattern, as well as the subset that constitutes the *solution*. The pattern application will actually replace the elements from the "problem" subset with the "solution" subset, leaving the remaining SModel elements unchanged. The actual implementation of the SModel transformation that corresponds to the pattern application rules is implemented using the standard language QVT (Query, View, and Transformation) Operational (QVT-O) [9]. Trace links between SModel and Pmodel (created during the initial PModel derivation) are used to propagate the changes to the PModel. We illustrate the proposed approach with the Service Façade design pattern applied to a Shopping and Browsing system.

The paper is organized as follows: Section 2 presents related work; Section 3 shows an overview of the approach proposed in this paper; Section 4 presents briefly the derivation of a performance model from the SOA design (i.e., SModel to PModel transformation); Section 5 discusses SOA design pattern specification using RBML; Section 6 presents the pattern application to a SModel using RBML and QVT-O; Section 8 shows how the SModel changes are propagated to the performance model and Section 9 presents the performance analysis results for a case-study system before and after a design patterns is applied. Finally Section 10 concludes the paper.

2. RELATED WORK

There are a significant number of papers on the derivation of a PModel of a system from the design model (SModel). The OMG standards for UML performance annotations, SPT and MARTE, have enabled research to transform UML design specifications into many kinds of performance models, based for example on Queuing Networks [10], Layered Queuing Networks [2, 11, 12] Stochastic Petri nets , etc.

PUMA is a set of transformations which take as input different SModels and transform them into different performance models [2, 11]. PUMA uses a pivot language, the Core Scenario Model (CSM), to extract and audit performance information from different kinds of design models (e.g., different UML versions and types of diagrams) and to support the generation of different kinds of performance models (e.g., QN, LQN, Petri nets, simulation). In [12], a graph-grammar based algorithm was proposed to divide the activity diagram into activity sub-graphs, which are further mapped to LQN phases or activities. In this work we propose to create a mapping between SModel and PModel elements during the PUMA model transformation, to help us propagating incremental modifications of the design model directly to PModel.

Another category of related works studies the impact of design patterns on software performance. Beside the patterns which are recognized as best practices for software development, Smith and Williams [13] introduced general performance anti-patterns, which exclusively focus on performance concerns. Anti-patterns are defined as common design mistakes that consistently occur, causing undesirable results. Cortellessa et al. [14] present an approach based on anti-patterns for identifying performance problems and removing them. The identification of an anti-pattern suggests the architectural alternatives that can remove that specific problem. Arcelli et al. [15] used the anti-pattern detection approach in [14] and also RBML to detect and remove performance anti-patterns from the software architectural model. In their approach [15], RBML is used to present the anti-pattern problems as source and anti-pattern solution as target role models. We use a similar approach for specifying the SOA design pattern, but the difference is that we implement the pattern application in QVT-O and propagate the changes directly to the PModel.

Menascé et al [16] presents a framework called SASSY whose goal is to allow designers to specify the system requirements using a visual activity-based language and to automatically generate a base architecture that corresponds to the requirements. The architecture is optimized with respect to quality of service requirements (i.e. as measured by several performance metrics such as execution time and throughput) through the selection of the most suitable service providers and application of quality of service architectural patterns.

Parsons and Murphy [17] introduce an approach for the automatic detection of performance anti-patterns by extracting the run-time system design from data collected during monitoring by applying a number of advanced analysis techniques.

Xu [18] applied rules to performance model results to diagnose performance problems and to propose solutions for fixing them, which resulted in changes at the PModel level, which could be interpreted to suggest corresponding design changes. However, the changes were not propagated automatically to the SModel.

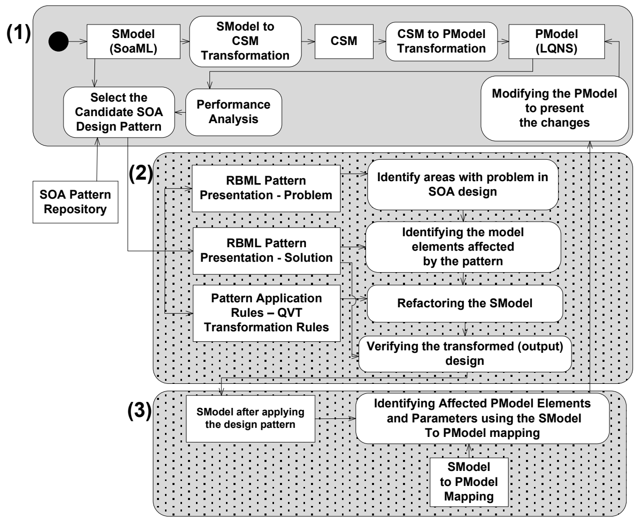
In this paper, we propose a technique for propagating the design pattern changes to the design of a service based system, which allows us to study the pattern impact on the system performance. We use RBML for the specification of the SOA design patterns and the OMG standard for model transformation, QVT-O for refactoring the SOA design according to the design pattern. To the best of our knowledge, none of the works in literature addresses the problem as proposed in this paper.

3. PROPOSED APPROACH OVERVIEW

Figure 2 shows an overview of the proposed approach. The key areas of the overview are shown in grey boxes (numbered from 1 to 3). Grey area 1 (solid grey) shows a more detailed view of what is shown in Figure 1.(A). As briefly discussed earlier, the performance model of a system (PModel) is generated from its software design model (SModel) extended with performance annotations (i.e. MARTE) using the transformation chain PUMA [2]. PUMA uses an intermediate model called "Core Scenario

Model" (CSM) to bridge the gap between different kinds of software models accepted as input and different kinds of performance models generated as output. CSM bridges the semantic gap between SModel and PModel, and captures only those software aspects that are relevant to performance models. During the initial transformation, trace links expressing the mapping between the SModel and the corresponding PModel elements are created which is used in grey area (3).

We assume that, in order to improve the quality of a SModel, the designer will apply different design patterns, which may impact performance and other non-functional properties, either positively or negatively. A designer selects a design pattern to solve a performance or non-performance problem and decides where to apply it by binding the roles defined in the pattern to specific SModel elements. In this paper we do not discuss the factors involved in selecting a candidate design pattern for SModel design issues, leaving it for future work. The grey areas (2) and (3) (dotted grey) in Figure 2 show the processes involved in applying the pattern and propagating the changes.



(1)
SModel (SoaML)
SModel to CSM Transformation
CSM
CSM to PModel Transformation
PModel (LQNS)
Select the Candidate SOA Design Pattern
Performance Analysis
Modifying the PModel to present the changes
SOA Pattern Repository
(2)
RBML Pattern Presentation - Problem
Identify areas with problem in SOA design
RBML Pattern Presentation - Solution
Identifying the model elements affected by the pattern
Pattern Application Rules – QVT Transformation Rules
Refactoring the SModel
Verifying the transformed (output) design
(3)
SModel after applying the design pattern
Identifying Affected PModel Elements and Parameters using the SModel To PModel mapping
SModel to PModel Mapping

Figure 2 : Proposed Approach Overview

Grey area (2) represents the process for refactoring the SModel by applying a pattern. The process starts when the designer selects a pattern. Next, the pattern specification is retrieved from the SOA pattern repository. As further discussed in Section 5, each design pattern consists of three main parts: problem, solution and application rules. In this paper, we use RBML to formally specify the first two, which are named "RBML Pattern Problem Specification"/"RBML Problem" and "RBML Pattern Solution Specification"/"RBML Solution". Details about RBML and the design pattern specifications are provided in Section 6. We use the QVT-O language for implementing the model transformation that corresponds to the pattern application rules. In Figure 2, area (2) "RBML Pattern Problem Specification" is used for identifying a subset of SModel elements that will be modified by the pattern (i.e., the respective model elements and their relationships must conforms to the RBML problem specification). If a match is found, the respective subset will be replaced by a "solution subset", which must conform to the "RBML Pattern Solution Specification". The actual transformation is described in the form of generic QVT rules, which consist of one or more basic actions, such as Adding, Removing, and Moving a model element or parameter. After performing the model transformation, the outcome is verified by checking whether the newly modified SModel elements conform with the RBML Pattern Solution Specification (see Section 7 for more details).

The SModel changes are propagated to PModel. As shown in Figure 2, area (3), the mapping between the elements of SModel and PModel is used to identify the required performance model changes. Finally, the flow of the processes in Figure 1 goes back to area (1) where the modified PModel after applying the design pattern is being re-analyzed to investigate the impact on the system performance. In Section 9, we discuss the performance results changes due to application of Service Façade design pattern to the SOA system case study.

4. SMODEL TO PMODEL TRANSFORMATION

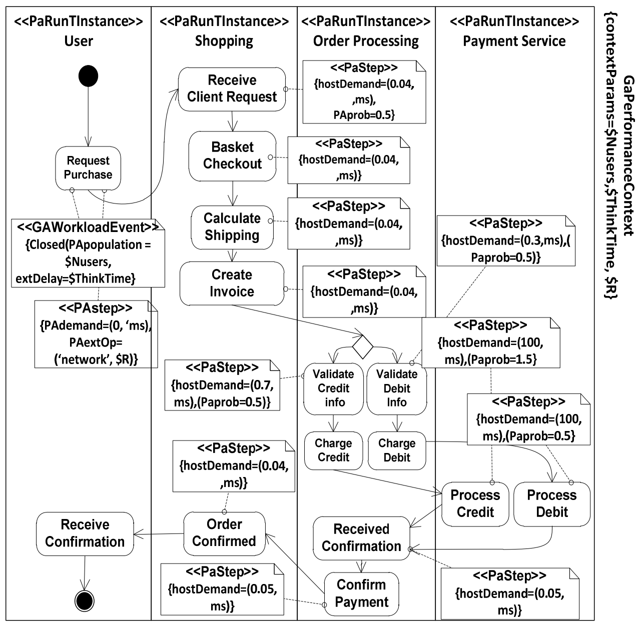
In this section we present briefly the SModel expressed in UML extended with SoaML and MARTE, and the corresponding PModel using the Layered Queuing Network formalism [2].

4.1 SoaML and MARTE annotations

The SoaML specification is an OMG standard for the design of services within a SOA system [5, 6]. It is defined as a UML profile that provides a standard way to architect and model SOA solutions. In this section we provide a brief description of the SModel of a SOA system used in our approach.

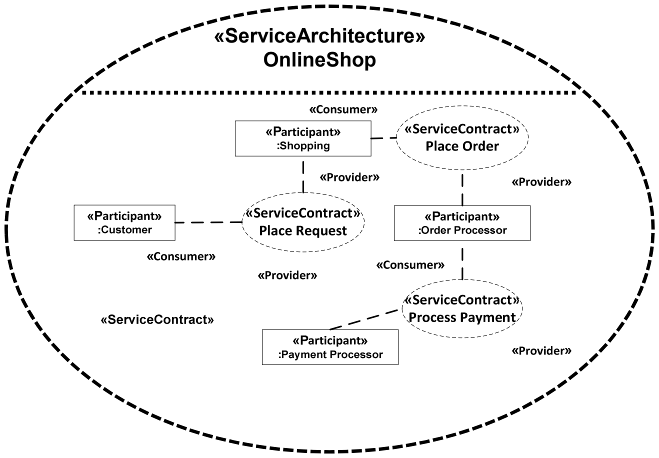
In SOA design using SoaML, the first step is to identify services by analyzing the business goals and objectives of the system, as well as the businesses processes to be implemented for meeting these objectives. Using these processes it is possible to identify other business-relevant services. The model created in this step is called Business Processes Model (BPM) diagram [8]. In this paper we specify the BPM diagram using the UML activity diagram notation [19]. An UML activity diagram representing an example of such a model for a shopping and browsing service is shown in Figure 3. Once the business processes have been identified, each will be specified in a business process diagram and refined in regard to participants, their tasks, and information flow between the participants.

The next step is to define the Service Architecture Model (SEAM) based on the existing BPMs. SEAM is a high level description of how participants work together for a purpose by providing and using services expressed as service contracts. Participants are recognized from pools, participants and lanes specified in the BPM processes. Once the participants are known, the possible interactions between the different participants must be identified and represented as service contracts. Figure 4 represents a SEAM in the form of a UML collaboration diagram with service participants and contracts. Participants are modeled as UML classes stereotyped *«Participant»* and a service contract is a UML collaboration with the stereotype *«ServiceContract»*. In SoaML, the service contracts can be refined further in a Service Contracts diagram [6]. Each participant plays either the role of Provider or of Consumer with respect to a service. For instance, in Figure 4 the Shopping participant provides PlaceRequest and consumes PlaceOrder. A participant may provide or consume any number of services. A service contract represented by a collaboration may contain nested participants and services.



<<PaRunTInstance>> User
<<PaRunTInstance>> Shopping
<<PaRunTInstance>> Order Processing
<<PaRunTInstance>> Payment Service
GaPerformanceContext {contextParams=$Nusers,$ThinkTime, $R}
Receive Client Request
<<PaStep>> {hostDemand=(0.04, ,ms), PAprob=0.5}
Basket Checkout
<<PaStep>> {hostDemand=(0.04, ,ms)}
Request Purchase
Calculate Shipping
<<PaStep>> {hostDemand=(0.04, ,ms)}
<<GAWorkloadEvent>> {Closed(PApopulation = $Nusers, extDelay=$ThinkTime}
<<PaStep>> {hostDemand=(0.3,ms),(Paprob=0.5)}
Create Invoice
<<PaStep>> {hostDemand=(0.04, ,ms)}
<<PAstep>> {PAdemand=(0, 'ms), PAextOp= ('network', $R)}
<<PaStep>> {hostDemand=(100, ms),(Paprob=1.5}
<<PaStep>> {hostDemand=(0.7, ms),(Paprob=0.5)}
Validate Credit info
Validate Debit Info
<<PaStep>> {hostDemand=(100, ms),(Paprob=0.5}
<<PaStep>> {hostDemand=(0.04, ,ms)}
Charge Credit
Charge Debit
Process Credit
Process Debit
Receive Confirmation
Order Confirmed
Received Confirmation
<<PaStep>> {hostDemand=(0.05, ms)}
Confirm Payment
<<PaStep>> {hostDemand=(0.05, ms)}

Figure 3: Checkout Business Process Model



«ServiceArchitecture» OnlineShop
«Consumer»
«Participant» :Shopping
«ServiceContract» Place Order
«Provider»
«Provider»
«Participant» :Customer
«ServiceContract» Place Request
«Participant» :Order Processor
«Consumer»
«Consumer»
«Provider»
«ServiceContract» Process Payment
«ServiceContract»
«Participant» :Payment Processor
«Provider»

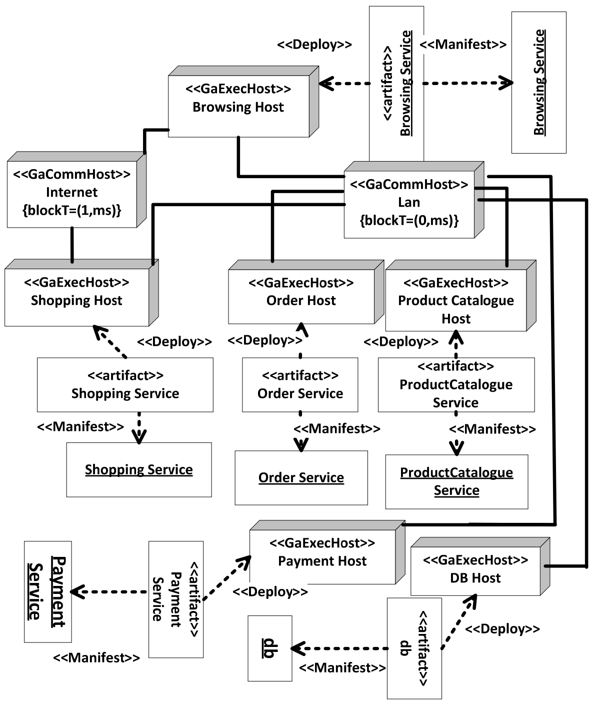
Figure 4: Service Architecture model for the Online Shop case study

Next, the specification of each service is further refined by defining the service interfaces between consumers and providers of the service. The diagram created in this step is called Service Interface diagram. It represents the provided and required interfaces, the roles that the interfaces play in the service specification, and protocols for how the roles interact. The third step, Service Realization, starts with identifying services that each participant provides and/or uses. This process has an important effect on service availability, distribution, security, transaction scopes, and coupling. The third step actually models how each service functional capability is implemented and how the required services are actually used. The forth step, called Service Composition or Components diagram, is about assembling and connecting the service participants models and then choreographing their interactions to provide a complete solution to the business requirements. Finally the last step is Service Implementation.

Among all the SoaML models mentioned in this section, our approach uses BPM (Figure 3) and SEAM (Figure 4) to propagate to PModel the changes made into the SModel by the application of design patterns. The reason is that all the information required to create the PModel can be obtained from these diagrams [3]. (See Section 8 for more details).

We add performance information to a UML+SoaML specification by adding annotations defined in the OMG standard profile MARTE [7]. Figure 3 shows an example of a SoaML behavioral diagram for the checkout operation of a shopping application. MARTE performance annotations describe the behavior as a Scenario, with steps and a workload *«GaWorloadEvent»* attached to the first step. Concurrent runtime instances *«PaRunTInstance»* are identified with swimlane roles. *«PaStep»* represents the execution of an activity or an operation invoked by a message, and has attributes *hostDemand* giving the required execution time and *PAprob* giving the probability if it is an optional step. The workload *«GaWorloadEvent»* defines a closed workload (a set of users) with a population given by the variable *$Nusers* and a think time for each user given by the variable *$ThinkTime*.

For performance analysis, the SOA specification must include the deployment of concurrent runtime component instances (see Figure 5). Where deployment is not specified, some kind of deployment is assumed, such as one host node with one concurrent process for each service. UML deployment diagrams will be used, and performance annotations are also added to this diagram. In Figure 5, the processing nodes are stereotyped as *«GaExecHost»* and the communication network nodes as *«GaCommHost»*, and the stereotypes have attributes for processing capacity, etc. Such annotations help us not only in transforming SModels into PModels, but also in propagating future modifications due to the application of design patterns.



<<Deploy>>
<<artifact>> Browsing Service
<<Manifest>>
Browsing Service
<<GaExecHost>> Browsing Host
<<GaCommHost>> Internet {blockT=(1,ms)}
<<GaCommHost>> Lan {blockT=(0,ms)}
<<GaExecHost>> Shopping Host
<<GaExecHost>> Order Host
<<GaExecHost>> Product Catalogue Host
<<Deploy>>
<<Deploy>>
<<Deploy>>
<<artifact>> Shopping Service
<<artifact>> Order Service
<<artifact>> ProductCatalogue Service
<<Manifest>>
<<Manifest>>
<<Manifest>>
Shopping Service
Order Service
ProductCatalogue Service
<<GaExecHost>> Payment Host
<<GaExecHost>> DB Host
Payment Service
<<artifact>> Payment Service
<<Deploy>>
<<Deploy>>
<<artifact>> db
db
<<Manifest>>
<<Manifest>>

Figure 5: SOA Deployment Diagram

4.2 Performance Model (PModel)

This work constructs performance models (PModels) in an extended queuing format called Layered Queuing Networks (LQNs) [2]. An LQN represents congestion in waiting for service provided by host processors and also by software servers.

Figure 6 shows the LQN model corresponding to the SoaML business process in Figure 3. For each service there is a task, shown as a bold rectangle, and for each of its operations there is an entry, shown as an attached (thin line) rectangle. The task has a parameter for its multiplicity (e.g. {1} for OrderProcessing task in Figure 6) and the entry has a parameter for its host CPU demand, equal to the *hostDemand* of the operation in SoaML (e.g. [0.3 ms]). Calls from one operation to another are indicated by arrows between entries (a solid arrowhead indicates a synchronous call for which the reply is implicit, while an open arrowhead indicates an asynchronous call). The arrow is annotated by the number of calls per invocation of the sender (e.g. (3)). For deployment, the host node is indicated by a round symbol attached to each task.

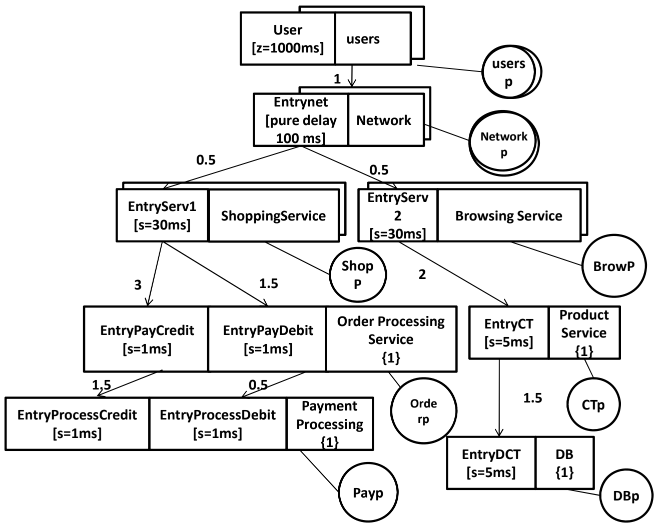


Figure 6 : LQN Performance Model (PModel) corresponding to Figures 3 and 4

We apply existing techniques [2, 11, 12] and also the technique in our previously published paper [3] to convert the UML+SoaML+MARTE SModel to a LQN model and to build the mapping between SModel and PModel elements. The mapping is further discussed in Section 8.

5. PATTERN APPLICATION RULES

A design pattern is defined as a proven design solution for a common design problem that is formally documented in a consistent manner. In the context of SOA, there are many categories of design patterns which address different aspect of a SOA-based systems including but not limited to: service messaging patterns, service implementation patterns, service security patterns, composition implementation patterns, etc. [1]. A SOA design pattern contains the description of a situation or problem where it applies, the solution for it and application rules. The RBML formal specification of these patterns and implementation of their application rules in form of QVT rules proposed in this paper helps the system designers to systematically identify the place in a SModel where a pattern can be applied and to automatically apply the pattern.

This section describes a pattern from the category of service implementation patterns called "Service Façade" [1]. Later this example will be used to show traceability issues and techniques.

Design Pattern Service Façade [1]. In general, the Service Façade addresses the way in which a service can accommodate changes to its contract or implementation while allowing the core service logic to evolve independently.

Problem: Usually a service contains a core logic that is responsible for operating its main capabilities. When a service is subject to change either due to changes in the contract or in its underlying implementation, this core service logic is also prone to modifications to accommodate that change.

Solution: Façade logic is added into the service architecture to create one or more layers of abstraction that can accommodate future changes to the service contract, the service logic, and the underlying service implementation.

Applications Rules: Service façade components can be positioned within the service architecture in different ways, depending on the nature and extent of abstraction required, such as: 1) between the core service logic and the contract to intentionally tightly couple to their respective contracts, allowing the core service logic to remain loosely coupled or even decoupled 2) between the core service logic and the underlying implementation resources to help shield core service logic from changes to the underlying implementation by abstracting backend parts.

As can be understood from the pattern description, it requires a new component to be added to the service architecture wherever there is service core logic that is tightly coupled with its service contract or underlying implementation. The application rules for this pattern are defined as:

- **Conditions:** If there is a core service logic in the SOA design which is identified as tightly coupled (considered as a design-choice by the system designer) with its associated contracts or underlying implementations.
- **Actions:** Add a new façade participant between the core service logic participant and the coupled contracts/underlying resources. Change all the communications in the service core logic and the coupled contract/underlying implementation to consume/provide services from/to the façade participant. Add processes (activities) to the newly added participant as façade to handle the responses/requests from the service core logic and also service contract/underlying implementation. Since the façade participant is created to be coupled with the service core, this newly added participant should be created as a private participant and in the same component (or even the same physical service).

As it can be understood from this pattern, new processes and participants need to be created in the SOA design. Therefore this pattern is targeting the business process in the SOA design. In future sections, we use this design pattern and apply it to the case study shown in Figure 3 and trace its impacts on the PModel of the case study.

There are several ways that the facade pattern could be applied to the SOA specification shown in Figure 3 and Figure 5. In general, if there is a service core which is prone to future changes, a service façade can help to protect it. In the SOA example in this paper, the shopping and browsing services, payment service, and product service are candidates for service façade, but we assume that the shopping service is selected as core service. The Service Façade pattern will be applied to accommodate requests on multiple channels (different types of devices, e.g. desktops, tablets, and smartphones). The capability to handle requests from multiple channels could be created either by adding new activities to each service/task (which may corrupt the core logic), or by applying the Service Façade pattern, so we consider both alternatives. In Section 7 and 8 , the change due to application of the Service Façade pattern is propagated to the PModel and then in Section 9, the system performance is evaluated for both approaches (multi-channel implementation with and without Service Façade).

6. ROLE-BASED SOA DESIGN PATTERN SPECIFICATION

Informal descriptions of the design patterns (similar to the one discussed in section 5) are useful for communicating proven solutions to the development team, but they lack the formality needed to support precise specification of design patterns and the development of automated techniques and supporting tools. Although formal pattern specification languages using a mathematical notation (e.g., see [20, 21]) provide the concepts needed to precisely describe design patterns, applying them requires sophisticated mathematical skills. Therefore in this research, we decided to deploy pattern specification languages that are based on familiar software modeling concepts (e.g. UML) to specify SOA design patterns in our proposed technique in this paper. The notation used is called Role-Based Modeling Language (RBML) [8] , which it was used for a similar purpose (i.e., specifying performance anti-patterns) in [22].

A Role Model is a structure of meta-roles (hereafter called roles), where a role defines properties that determine a family of UML model elements [8]. A UML model element conforms to a role if it is an instance of the role's base and has the properties specified in the role. A Role Model is thus a characterization of UML diagrams and its profiles such as SoaML. Therefore a Role Model realization is a model (e.g., a static structural diagram, sequence diagram) that consists of realizations of the roles in the Role Model. The concept of role in RBML is very helpful to capture the various aspects of the knowledge in the design patterns.

In model specification using RBML, the specification consists of a Static Role Specification (SRMs) that specifies the static structural models, and a set of Interaction Role Models (IRMs) that specifies interactions (i.e. behaviors) [8]. Respectively, we use SRM to represent the static and IRM for interactive (behavior) aspects of SOA design patterns.

In [8], authors provided the SRM notation and IRM samples for UML sequence and collaboration diagrams. Since in this research we use UML+SoaML to present the SOA system under study, we extended the concepts proposed in [8] with IRM for activity diagrams using UML extended with profiles.

6.1 Static Role Models (SRMs) in SOA

Figure 7.(A) shows the structure of a single SRM role. A SRM role characterizes a set of UML static modeling constructs (e.g., class, and association constructs). The top section has three parts: 1) a role base declaration in form of "«Base Role»", where Base is the name of the role's base (i.e., the name of a metamodel class) 2) a role name declaration in form of "/RoleName", where RoleName is the name of the role; and 3) a realization multiplicity that specifies the allowable number of realizations that can exist for the role in a realization of the SRM that includes the role. The second section contains metamodel-level constraints and the third, optional, section contains feature roles that determine a family of application-specific properties (e.g., properties represented by attributes and operations defined in application-specific classes).

In RBML, there are two types of feature roles:

(1) Structural roles that specify state of the properties that are realized by attributes or operations in a SRM role realization.

(2) Behavioral roles specify behaviors that are realized by a single operation or method.

In this paper, we use SRM to represent the static aspects of a SOA design pattern. In pattern specification using SRM, each SOA participant (e.g., providers and consumers) can be considered as a role. An example of service façade SRM is shown in Figure 7.(A) SRM presents a role for coupled core service logic. In Figure 7.(B), the "Shopping" service is playing that role (assuming that this was the decision of the designer).

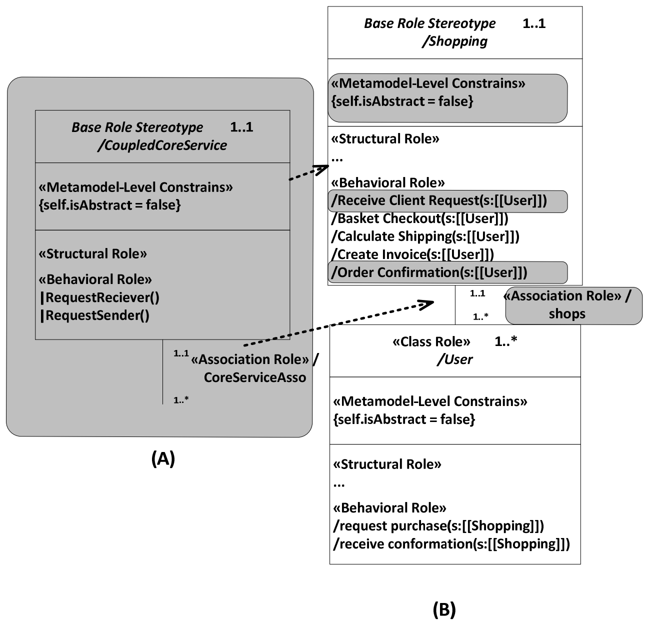

Base Role Stereotype 1..1
/CoupledCoreService
«Metamodel-Level Constrains»
{self.isAbstract = false}
«Structural Role»
«Behavioral Role»
|RequestReciever()
|RequestSender()
1..1 «Association Role» /
CoreServiceAsso
1..*
(A)
Base Role Stereotype 1..1
/Shopping
«Metamodel-Level Constrains»
{self.isAbstract = false}
«Structural Role»
...
«Behavioral Role»
/Receive Client Request(s:[[User]])
/Basket Checkout(s:[[User]])
/Calculate Shipping(s:[[User]])
/Create Invoice(s:[[User]])
/Order Confirmation(s:[[User]])
1..1
1..*
«Association Role» /
shops
«Class Role» 1..*
/User
«Metamodel-Level Constrains»
{self.isAbstract = false}
«Structural Role»
...
«Behavioral Role»
/request purchase(s:[[Shopping]])
/receive conformation(s:[[Shopping]])
(B)

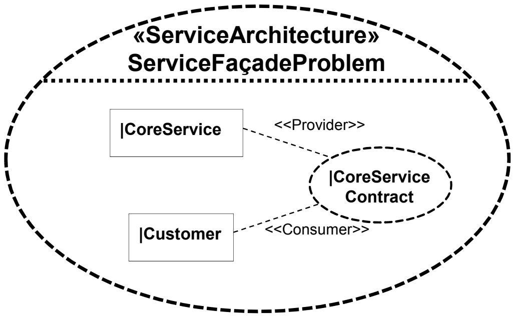
Figure 7 : (A) SRM of a coupled core service in Service Façade (B) Conformance to coupled core service in Shopping example

In SOA systems, the architecture is represented not only with class diagrams, but also with collaboration diagrams, as for example the service architecture. Therefore, we also use SRM specifications in the form of collaborations (see Figure 8).

6.2 Role Relationships

In RBML, a role can be associated with another role in a class diagram, indicating that the realizations of the roles are associated in a manner that is consistent with how the bases of the roles are related in the UML model. RBML uses the UML form of association to represent relationships between roles. Role associations can be named and can have multiplicities associated with their ends. An example of relationship between roles in SoaML is shown in Figure 7.(B). In this figure, the "Shopping" service, which plays the role of Coupled Core Service in the

Service Façade design pattern is in "shop" relationship with "User". The multiplicity on the role relationship shows that more than one user can be in "shops" relationship with the "Shopping" service.



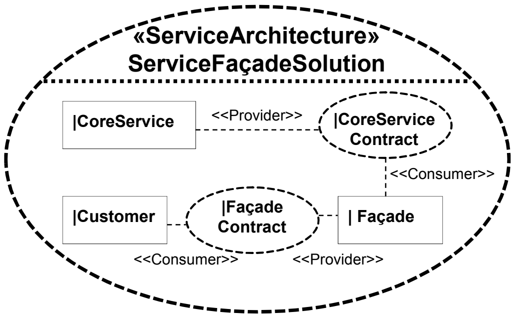


Figure 8: SRM specification for the Service Façade pattern: problem (top) and solution specification (bottom)

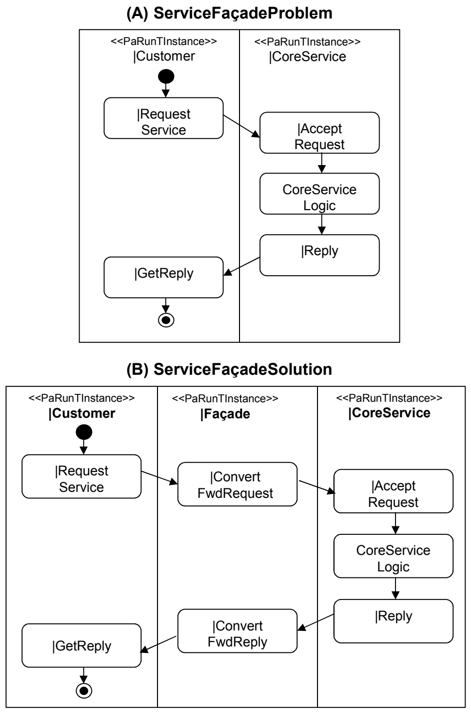


Figure 9: IRM specification for the Service Façade behavior view: (A) pattern problem specification; and (B) pattern solution specification

The RBML specification of the service architecture view of the Façade Pattern is illustrated in Figure 8: the problem specification on the top and the solution on the bottom. The roles played by the participants and the service contracts are represented as formal parameters (by convention the names start with '|'). When the pattern will be applied, the formal parameters |Customer, |CoreService, |CoreServiceContract will be matched and bound to actual participants and service contracts from the SModel service architecture, as described in Section 7.

6.3 Interaction Role Models (IRM)

We use Interaction Role Model (IRM) to present the interactions between participants in SOA behavior diagrams (e.g., a BPM diagram). IRM is defined using the roles specified as formal parameters in the SRM for specifying the participants and may add other roles to specify some of their actions. Figure 9 illustrates the IRM behavior view of the Façade Pattern: (A) the pattern problem and (B) the pattern solution. When the pattern is applied, a new participant |Façade is added. Its activities are located in a new swimlane in the activity diagram in Figure 9.(B). The activities from the Service Façade Problem with names starting with '|' (i.e., formal parameters) will be matched and bound to activities from a concrete BPM diagram, which will be then modified according to the pattern solution. For instance, a participant playing the role of |Customer will need to perform an activity that matches |RequestService and another that matches |GetReply. The activity sending the request will be followed by an edge crossing the swimlane to the |CoreService, while the activity receiveing the reply has an incoming edge from the |CoreService swimlane. Similarly, the activities execute by a participant playing the role of |CoreService will have to be matched with |AcceptRequest and |Reply, respectively. Figure 10 illustrates the refactored BPM after applying the Service Façade; its activities shaded in grey are bound to formal parameters from Figure 9.(B).

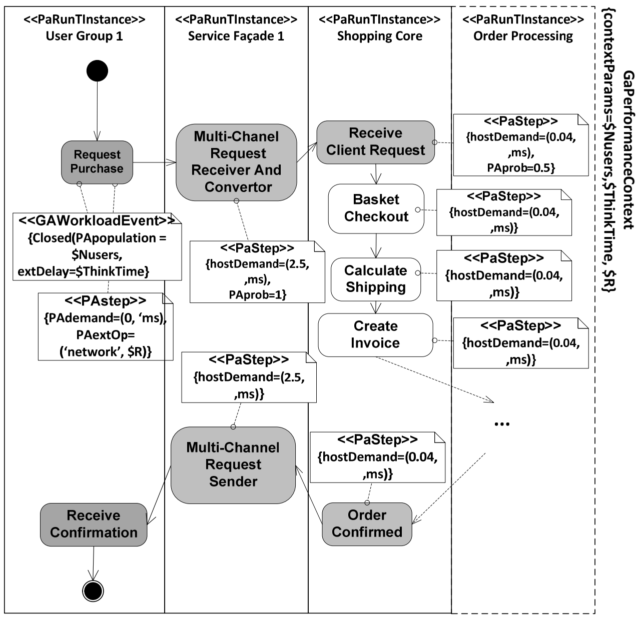
6.4 Conformance with Pattern SRM and IRM

An SRM is a generic presentation of the SOA model structure. Each role in a SRM specifies a specialization of its base class in the UML model. In this paper, we consider that a SModel diagram or sub-diagram structure conforms to a pattern SRM if each role defined in the pattern SRM can be played by at least one model element from the SModel. This involves the following:

(i) Each role defined in the pattern SRM has at least one matching SModel element and for each role there is at least one match for metamodel-level constraints specified in SRM.

(ii) Each attribute/behavior defined for the role in pattern SRM has a matching attribute/behavior among the SModel elements.

Also we consider that a SModel behavior diagram conforms to a pattern IRM if: (a) it conforms structurally to the associated pattern SRM and (b) the behavior of roles specified in IRM match the behavior of elements playing SRM roles. The definition of behavioral conformance is dependent to the UML behavioral diagram used to illustrate the scenario. In SoaML, a BPM which is an UML activity diagram is one of the main behavioral diagrams. An activity diagram used in a SOA SModel conforms to IRM if the model elements matching the SRM roles are following the same execution flow and are located in matching swimlanes as the roles in IRM.



<<PaRunTInstance>> User Group 1
<<PaRunTInstance>> Service Façade 1
<<PaRunTInstance>> Shopping Core
<<PaRunTInstance>> Order Processing
GaPerformanceContext {contextParams=$Nusers,$ThinkTime, $R}
Request Purchase
Multi-Chanel Request Receiver And Convertor
Receive Client Request
<<PaStep>> {hostDemand=(0.04, ,ms), PAprob=0.5}
Basket Checkout
<<PaStep>> {hostDemand=(0.04, ,ms)}
<<GAWorkloadEvent>> {Closed(PApopulation = $Nusers, extDelay=$ThinkTime}
<<PaStep>> {hostDemand=(2.5, ,ms), PAprob=1}
Calculate Shipping
<<PaStep>> {hostDemand=(0.04, ,ms)}
<<PAstep>> {PAdemand=(0, 'ms), PAextOp= ('network', $R)}
Create Invoice
<<PaStep>> {hostDemand=(0.04, ,ms)}
<<PaStep>> {hostDemand=(2.5, ,ms)}
...
Multi-Channel Request Sender
<<PaStep>> {hostDemand=(0.04, ,ms)}
Receive Confirmation
Order Confirmed

Figure 10 : Refactored BPM after applying the service façade design pattern

7. PATTERN APPLICATION USING RBML SPECIFICATIONS AND QVT-O

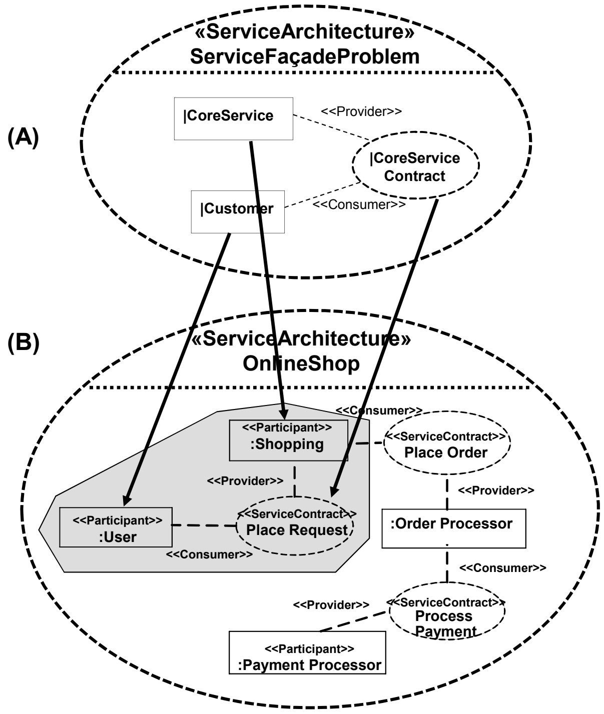
In Section 6, we discussed the importance of using a formal UML-based pattern definition language for the specification of the design patterns. Furthermore we discussed how we used RBML to specify the design pattern problem and solution description. In this section we discuss the way that the RBML specification for both solution and problem is being used to apply the design patterns to a SModel. First, the "RBML problem" is used for identifying the subset of SModel elements that will be affected by the pattern, then the "RBML solution" is used to identify the affected model elements due to application of the pattern . Once the impacted elements in the SOA are identified, the QVT-O transformation rules are executed to apply the changes (Section 7.2). Finally, the refactored SModel is being verified by checking its conformance with "RBML solution". Due to lack of space, in this section we only show the application of design patterns to SModel BPM and SEAM. Using the same techniques, the design pattern changes can be reflected to SModel in Deployment Diagram.

7.1 Conformance with RBML Specification

The first step toward applying a design pattern to a SOA design is to identify the places in the SModel where the design pattern can be applied. This can be done by finding a diagram or subset of it which conforms to the RBML Pattern Problem Specification.

In our proposed approach, the place in the SModel where a design pattern should be applied is identified by the software designer. Figure 11.(B) shows the SoaML service architecture diagram for the case study system, and the subset of model elements for applying the Service façade pattern shown in grey does conform to the RBML Pattern Problem Specification shown in Figure 11.(A).

As shown in Figure 11, there might be more than one place in the SModel which conforms to the RBML Pattern Problem Specification, but since Service Façade is supposed to be applied to the core service logic only, the software designer must indicate which service is considered as core service. This is another reason for the direct intervention of software designer in the selection of design patterns and the place where they should be applied.



(A)
«ServiceArchitecture» ServiceFaçadeProblem
|CoreService
<<Provider>>
|CoreService Contract
|Customer
<<Consumer>>
(B)
«ServiceArchitecture» OnlineShop
<<Consumer>>
<<Participant>> :Shopping
<<ServiceContract>> Place Order
<<Provider>>
<<Provider>>
<<Participant>> :User
<<ServiceContract>> Place Request
:Order Processor
<<Consumer>>
<<Consumer>>
<<Provider>>
<<ServiceContract>> Process Payment
<<Participant>> :Payment Processor

Figure 11: (A) RBML Facade Pattern Problem Specification" (B) SModel Service Architecture diagram

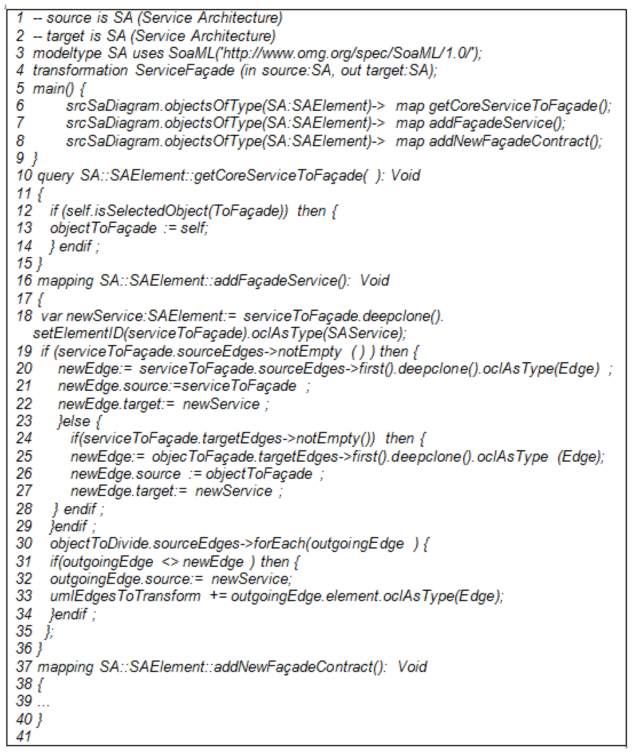
Once the subset of SModel elements corresponding to the pattern problem was identified, the following changes need to take place:

- *<u>Adding</u> a new contract to service architecture associated to the new communication between the service façade participant and service (in this case, "Format Multi-Channel Request" is added as the Façade contract)*
- *<u>Moving</u> all the communication to core service logic contract to the newly added service contract (in this case the communication from participant " User1" is moved to "Format Multi-Channel Request")*
- *<u>Adding</u> a new contract participant as Service Façade ("Multi Channel Façade 1" is added)*
- *<u>Moving</u> all the communications from the core service logic contract to the newly added contract participant (all the communications from "Place Request" are moved to "Multi Channel Façade 1").*

All the above steps are defined as generic steps for adding a service Façade to an existing SOA design. The above steps become specific when they are bound to model elements of a specific pattern application. Therefore similar to RBML Pattern Specifications of the problem and solution, the above can be translated into a form of model transformation rules and be re-used for future application of the Service Façade design pattern to any SModel. In the Section 7.2 we discuss this in more details.

7.2 Applying the Design Pattern using QVT-Operational

The RBML graphical presentation of the design patterns helps the software engineer to identify the elements that need to be added or removed from a SModel, but it does not help in applying the changes to the model. Therefore in this research, we define the model transformation rules corresponding to the application of a pattern in OMG's standard language QVT-Operational [9]. The rules are able to transform (more specifically, refactor) a SModel by adding or removing model elements. QVT defines three domain-specific languages: Relations, Core and Operational. In this research, we chose QVT Operational (QVT-O) because of its flexibility in comparison with the other two languages, due to the fact that it includes constructs commonly found in imperative languages, such as loops and conditions (i.e. Figure 12).



```
1  -- source is SA (Service Architecture)
2  -- target is SA (Service Architecture)
3  modeltype SA uses SoaML("http://www.omg.org/spec/SoaML/1.0/");
4  transformation ServiceFaçade (in source:SA, out target:SA);
5  main() {
6         srcSaDiagram.objectsOfType(SA:SAElement)->  map getCoreServiceToFaçade();
7         srcSaDiagram.objectsOfType(SA:SAElement)->  map addFaçadeService();
8         srcSaDiagram.objectsOfType(SA:SAElement)->  map addNewFaçadeContract();
9  }
10 query SA::SAElement::getCoreServiceToFaçade( ): Void
11 {
12   if (self.isSelectedObject(ToFaçade))  then {
13   objectToFaçade  := self;
14   } endif ;
15 }
16 mapping SA::SAElement::addFaçadeService():  Void
17 {
18  var newService:SAElement:=  serviceToFaçade.deepclone().
   setElementID(serviceToFaçade).oclAsType(SAService);
19  if (serviceToFaçade.sourceEdges->notEmpty  ( ) ) then {
20     newEdge:=  serviceToFaçade.sourceEdges->first().deepclone().oclAsType(Edge)  ;
21     newEdge.source:=serviceToFaçade  ;
22     newEdge.target:=  newService ;
23     }else {
24       if(serviceToFaçade.targetEdges->notEmpty())  then {
25       newEdge:=  objecToFaçade.targetEdges->first().deepclone().oclAsType  (Edge);
26       newEdge.source  := objectToFaçade  ;
27       newEdge.target:=  newService ;
28    } endif ;
29   }endif ;
30   objectToDivide.sourceEdges->forEach(outgoingEdge  ) {
31   if(outgoingEdge  <> newEdge ) then {
32   outgoingEdge.source:=  newService;
33   umlEdgesToTransform  += outgoingEdge.element.oclAsType(Edge);
34   }endif ;
35   };
36 }
37 mapping SA::SAElement::addNewFaçadeContract():  Void
38 {
39 ...
40 }
41
```

Figure 12: QVT-O rules for Service Façade Pattern

By examining the types of SModel changes needed to apply a design pattern, we introduced the following three refactoring primitives: 1- Adding a model element, 2- Removing a model element, and 3- Moving a model element. Using these refactoring primitives, more complicated refactoring actions (e.g. Merge and Divide) can also be implemented. In the model transformation we use in this paper, it's assumed that the model elements which are not impacted by the patterns are being copied into result model.

Figure 12 shows the QVT-O rules that have been created for the solution statement of the Service Façade pattern. The main function starts in line 5 with *getCoreServiceToFaçade()*. This function retrieves the selected core service logic by the system designer that the Service Façade needs to be applied for. Then a new Service Façade and also an associated service contract are being created by *addFaçadeService()* and *addNewFaçadeContract()* and all the connecting edges (i.e. incoming and outgoing) are being updated to accommodate the new model nodes.

addFaçadeService() adds the new Service Façade node to the target diagram by making a copy of the provided core service logic. Line 18 shows that the new node is being created using the *deepclone()* that QVT-O provides. Lines 19-29 contain conditional statements checking if there exist any incoming or outgoing edges that connect to the core service logic. When found, the first edge in either set is copied with the QVT-O *deepclone()* operation and then connected to the core service logic and the Service Façade (cloned node) by setting the source and target properties appropriately.

In lines 29-36 all outgoing edges from the core service logic are moved to the Service Façade by setting the source property of each edge to point to the new element. *addNewFaçadeContract()* function will also works very similarly to *addFaçadeService()* to create a new service contract for the newly created Service Façade.

The result of executing the QVT-O transformation rules for Service Façade design pattern (i.e. which are partially presented in Figure 12) is shown in Figure 13.(B). The Façade service (i.e. called contract participant in SoaML service architecture diagram) and the Façade contract are both added and all the communications are moved to accommodate the design pattern changes.

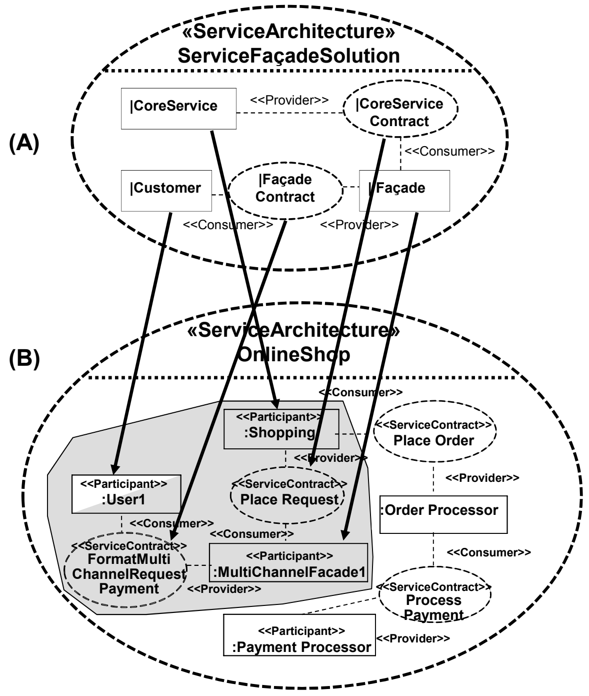


Figure 13: (A) "RBML Pattern Solution Specification" (B) Refactored SoaML Service Architecture diagram

7.3 Verifying the Transformation Output

After executing the model transformation implemented in QVT-O the transformed model can be verified to make sure that it represents the changes correctly. This not only verifies the content of the generated SOA design, but it also verifies the transformation rules which are used to generate the transformed diagrams. The application of the design pattern is implemented in the transformation and thus if the verification detects an error, this can be considered as a bug in the transformation and the transformation rules can be corrected. The step is part of the research (as an end-to-end verification of the transformation) rather than part of the proposed methodology. In this work, we verify the generated SOA design by checking the conformance of the transformed areas with the "RBML Pattern Solution Specification". Figure 13 shows the generated model (Figure 13.(B)) using the QVT-O transformation rules and also the conformance of the modified area with the "RBML Solution" (Figure 13.(A)).

8. IDENTIFY PMODEL CHANGES

In Section 7, we discussed how a design pattern can be applied to an SModel and how the modifications are identified. As a PModel is being generated from the SModel, modifications to SOA design lead to changes into the associated PModel of SOA and the performance analysis results. The LQN performance model is derived from the SOA Service Architecture (Figure 13.(B)), SOA BPM (Figure 10) and the SOA Deployment Diagram (Figure 5). Due to lack of space, in this paper we do not discuss the changes made to deployment diagram due to application of design pattern but for the sake of case study in this paper, the process is performed based on discussions in Sections 6 and 7 and results are used in performance evolution provided in Section 9. The changes to the LQN performance model due to the impact of design pattern are passed through these three models. The key elements of the LQN model are LQN Tasks, Processors, Entries and RequestArcs. The LQN performance model of the SOA under study before applying the Service Façade design pattern is shown in Figure 6 and its structure is discussed in Section 4.2. Therefore, a very straightforward mapping between SModel and PModel (which is determined automatically) is established as follows [3]:

- Mapping deployment diagram Executing Host Nodes to LQN hosts (processors)
- Mapping Service Architecture diagram Contract Participant (i.e. services) to LQN tasks
- Mapping activities in the BPM swimlane associated to the service presented in Service Architecture diagram to LQN entries
- Mapping Swimlane-crossing Edges to LQN Requests Arcs.

Some swimlane-crossing Edges associated to a service in BPM represent outgoing requests for the service and some represent the incoming responses. Since in LQN diagram, the LQN Requests Arcs represent both requests and responses from the LQN task, only the BPM Swimlane-crossing Edges that make the request are mapped into LQN Requests Arcs. A more detailed discussion about mapping Swimlane-crossing Edges to LQN requests can be found in [12]. Based on the SOA design changes discussed in Section 7 and also the mapping showed in this section (see the grey area 3 in Figure 4 approach overview), the newly added Service Façade in service architecture diagram (Figure 13.(B)) in and the activities (annotated by «PaStep») in its associated swimlane (annotated by «PaRunTInstance») in BPM (Figure 10) cause the creation of the "Service Façade" LQN task and its LQN entries, "EntryFaçade" shown in grey in the LQN model in Figure 14. Also, since a new processing node is added to the deployment diagram for the newly added "Service Façade" component, a LQN processor is created in the LQN model and attached to the corresponding LQN task.

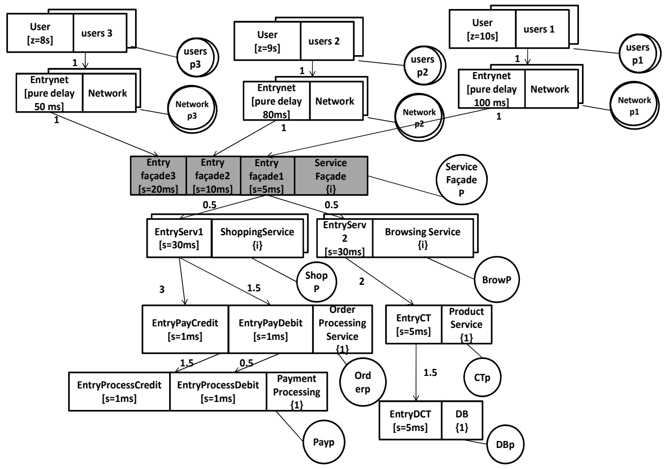


Figure 14: SOA Performance Model after applying the façade design pattern

9. PERFORMANCE ANALYSIS

This section discusses the impact of the pattern application on the performance results of the case study system. We evaluate the performance results of Shopping and Browsing SOA before and after applying the service façade design pattern. In this case study, it is assumed the core service logic is the Shopping and Browsing service and the service façade design pattern is applied to help the core service handle requests from more than one user type by adding a multi-channel service façade to SOA. We evaluate four scenarios named A, B, C and D, to find the system throughput and response time of all three user groups for a range of numbers of users in each group. For N users in "User Group 1", we defined 2N in "User Group 2", and N/2 in "User Group 3". N was varied from 2 to 220, so the total number of the system users (i.e. N+2N+N/2=3.5N) ranged from 7 to 770. In scenario A, the service façade pattern is not applied and instead the multi-channel capability is implemented by adding to each of the shopping and browsing services in the system. The performance model for scenario A is shown in Figure 15. In scenario A, a separate task entry is created for each service as used by each user group (1, 2 and 3). In scenario B the service façade pattern is applied, giving the performance model shown in Figure 14. The new service façade task added to the system has three entries, each responsible for the interface with one group of users. Also the service facade task has its own dedicated processor. Figure 16 compares the system throughputs and Figure 17 compare the system response times of scenarios A and B for each user group of the system. Figure 16 and Figure 17 show that scenario B has better throughput and response time than A for all three user groups. The underlying reason is that in scenario B a new processor is dedicated to the Service Façade task, while in scenario A the new functionality related to the multi-channel interface is running on the existing processors for the Shopping and Browsing services. To make a more fair comparison between the "before" and "after" scenarios, we use our knowledge of how the resources are used in the scenarios A and B to experiment with scenarios C and D.

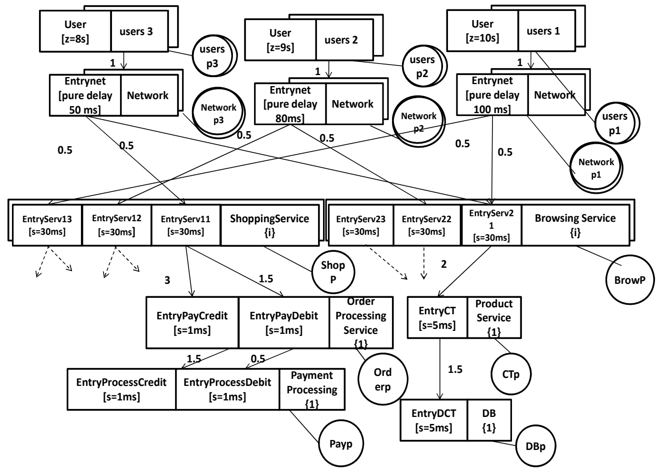


Figure 15: Performance Model after applying the multi-channel capability to each service without applying any Service Façade Design Pattern

Scenario C is obtained from A by changing the processors for the browsing and shopping services from single to dual-core. The reason is that these processors become the bottleneck in case A at high loads. Scenario D is obtained from B by changing the processors for the shopping and browsing services to dual processors and by allocating the service façade task to one of the existing processors that are under-utilized (in this case the processor of the Order Service task). So, the hardware resources for C and D are identical. The results for C and D are shown in Figure 18 and Figure 19.

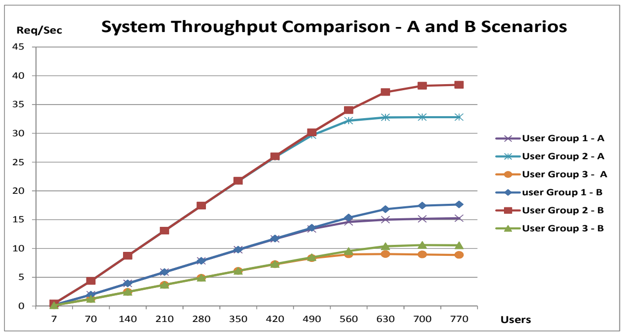


Figure 16: System Throughput for Scenarios A and B

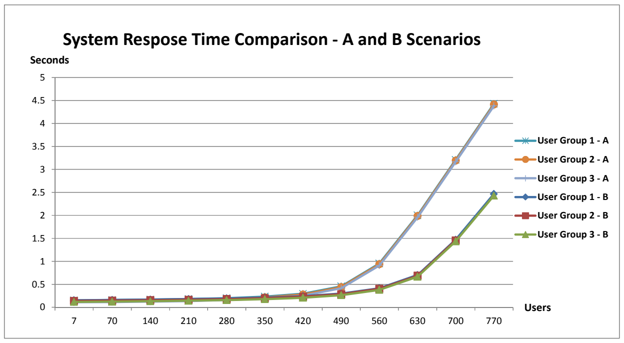


Figure 17: System Response Time for Scenarios A and B

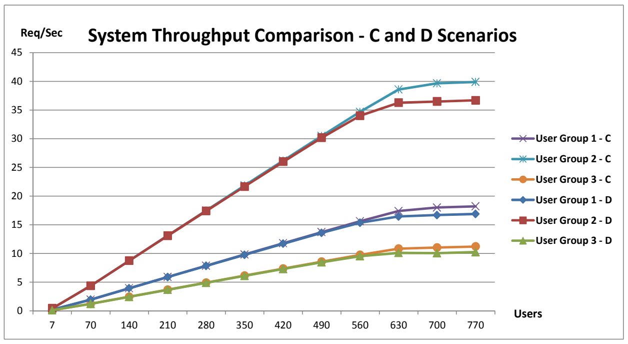


Figure 18: System Throughput for Scenarios C and D

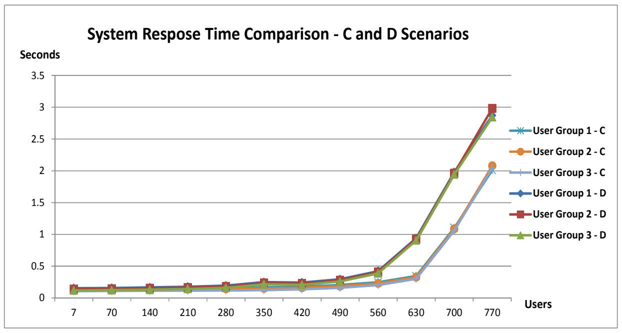


Figure 19: System Response Time for Scenarios C and D

Firstly, the throughput of C is better by about 20% than A due to the extra processing power added to the system. Secondly, the throughput and response time for D (the modified version of B) are only slightly worse than scenario C (the modified version of A) even though now C and D are running on identical hardware resources. The underlying reason is that we made use of our understanding of the resource utilization in the system and deployed the service façade in D on one of the existing but under-utilized processors. In conclusion, we could minimize the negative performance effect of the façade service pattern with the help of the LQN model, while taking advantage of its benefits to the system architecture (i.e. creating one or more layers of abstraction that can accommodate future changes to the service).

10. CONCLUSIONS

This paper traces the change propagation due to applying a SOA design pattern, from the SOA design model (created using SoaML) to its performance model. In illustration and using our approach, an example of SOA design patterns called Service Façade was applied to a Shopping and Browsing SOA case study, to equip it with a multi-channel request handler. In Sections 6 and 7 we discussed how the SOA design patterns can be applied to the SOA design using the RBML pattern presentation and QVT-O application rules. In Section 8 we showed that how the changes to SModel due to the application of a design pattern can be traced into performance model of the SOA using the mapping between SModel and PModel elements.

The overall possibilities of the technique are demonstrated in Section 9 by showing the results of the performance analysis. In Section 9, we evaluate the performance of the Shopping and Browsing service using four different scenarios that represent the Shopping and Browsing SOA with multi-channel capability before and after applying the service façade. The experiment shows the system designer the impact of the design pattern on the performance results of system in a time and cost effective way (without constructing a new performance model for the modified SOA design) and helps to make decisions about applying the design pattern. The results also indicate that the impact of the design pattern is partly determined by how the system is deployed. We are planning on developing QVT rules to also modify the LQN model to reflect the SOA design pattern changes. Furthermore, we are planning to integrate the proposed approach in a methodology for improving the quality of SOA systems, which will address the issue of selecting appropriate design pattern candidates. This will give a complete technique for SOA design improvement.

11. ACKNOWLEDGEMENTS

This work was supported by the Ontario Centers of Excellence through its Centre of Excellence for Research in Adaptive Systems (CERAS), by the Natural Sciences and Engineering Research Council (NSERC) through its Discovery Grant program, and by the Healthcare Support through Information Technology Enhancements (hSITE) Strategic Research Network.

REFERENCES

[1] Erl, T. *SOA Design Patterns* Prentice Hall PTR, Boston, MA, 2009.

[2] Woodside, M., Petriu, D. C., Petriu, D. B., Shen, H., Israr, T. and Merseguer, J. Performance by Unified Model Analysis (PUMA). In *Proceedings of the WOSP '05 Proceedings of the 5th international workshop on Software and performance*. ACM New York, NY, USA, 2005.

[3] Mani, N., Petriu, D. C. and Woodside, M. Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)* (Oulu, Finland). IEEE Computer Society Washington, DC, USA, 2011.

[4] Rotem-Gal-Oz, A., Bruno, E. and Dahan, U. *SOA Patterns (Early Access Edition)*. Manning Publications, 2007.

[5] Object Management Group, *Service oriented architecture Modeling Language (SoaML) URL: http://www.omg.org/spec/SoaML/1.0.1/ [Last time accessed Oct 5th, 2012]*. Version 1.0, formal/2009-11-02.

[6] Elvesæter, B., Carrez, C., Mohagheghi, P., Berre, A., Johnsen, S. G. and Solberg, A. *Model-driven Service Engineering with SoaML*. Springer Vienna, City, 2011.

[7] Object Management Group, *A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems)*. Version 1.0, formal/2009-11-02 URL: http://www.omg.org/spec/MARTE/1.0/PDF/ [Last time accessed Oct. 5th ,2012].

[8] France, R. B., Kim, D.-K., Ghosh, S. and Song, E. A UML-Based Pattern Specication Technique. *IEEE Trans. Software Eng.*,Vol 30 (3), 2004, 193-206.

[9] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) URL : http://www.omg.org/spec/QVT/1.1/ [Last time accessed Oct 5th, 2012]*. Version 1.1, formal/January 2011.

[10] Woodside, M., Petriu, D. C., Petriu, D. B., Xu, J., Israr, T., Georg, G., France, R., Bieman, J. M., Houmb, S. H. and Jürjens, J. Performance analysis of security aspects by weaving scenarios extracted from UML models. *Journal of Systems and Software*,Vol 82 (1), 2009, 56-74.

[11] Petriu, D. C. *Software Model based Performance Analysis*. ISTE Ltd and John Wiley & Sons Inc., City, 2010.

[12] Petriu, D. C. and Shen, H. Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. In *Proceedings of the TOOLS '02 Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*. Springer-Verlag, London, UK, 2002.

[13] Smith, C. U. and G.Williams, L. *Performance Solutions : A Practical Guide to Creating. Responsive, Scalable Software*. Addison Wesley, Boston, MA,, 2002.

[14] Cortellessa, V. and Mirandola, R. Deriving a Queueing Network based Performance Model from UML Diagrams. In *Proceedings of the WOSP '00 Proceedings of the 2nd international workshop on Software and performance*. ACM New York, NY, USA, 2000.

[15] Arcelli, D., Cortellessa, V. and Trubiani, C. Antipattern-based model refactoring for software performance improvement. In *Proceedings of the Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures* (Bertinoro, Italy). ACM New York, NY, 2012.

[16] Menascé, D. A., Ewing, J. M., Gomaa, H., Malex, S. and Sousa, J. P. A framework for utility-based service oriented design in SASSY. In *Proceedings of the WOSP/SIPEW '10 Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM New York, NY, USA, 2010.

[17] Parsons, T. and Murphy, J. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*,Vol 7 (3), 2008.

[18] Xu, J. Rule-based automatic software performance diagnosis and improvement. In *Proceedings of the Proceeding 7th Intl Workshop on Software and Performance*, Princeton, NJ, 2008.

[19] IBM, *Modeling with SoaML, the Service-Oriented Architecture Modeling Language*. January 7th , 2010 URL: http://www.ibm.com/developerworks/rational/library/09/modelingwithsoaml-1/ [Last time accessed Oct 5th, 2012].

[20] Eden, A. H., Yehudai, A. and Gil, J. Y. Precise specification and automatic application of design patterns. In *Proceedings of the ASE '97 Proceedings of the 12th international conference on Automated software engineering* (Incline Village, NV). IEEE Computer Society Washington, DC, USA, 1997.

[21] Lano, K., Bicarregui, J. C. and Goldsack, S. Formalising Design Patterns. In *Proceedings of the Proceeding of BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science*. British Computer Society Swinton, UK, 1996.

[22] Cortellessa, V., Marco, A. D., Eramo, R., Pierantonio, A. and Trubiani, C. Digging into UML models to remove performance antipatterns. In *Proceedings of the Proceeding of 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems* (Cape Town). ACM New York, NY, 2012.