Studying the Performance Impact of SOA Design Patterns via Coupled Model Transformations

by

Nariman Mani

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Early performance analysis of designs for Service Oriented Architecture (SOA) can be based on performance models derived from the design models using known techniques, such as Performance from Unified Model Analysis (PUMA). When a SOA design pattern is applied to solve some architectural, design or implementation problem, it impacts the design model and its derived performance model. Conventionally, the performance model needs to be reconstructed to reflect the design pattern changes on the design model. This thesis proposes a technique to trace the causality from the design changes introduced by the pattern application to the corresponding changes in the performance model. The approach takes as input a SOA design model expressed in UML extended with two standard profiles: SoaML for expressing SOA solutions and MARTE for performance annotations. The SOA design patterns are specified using Role Based Modeling (RBM) and the performance model is expressed in Layered Queueing Networks (LQN).

To support the exploration of different patterns, the thesis proposes the following approaches: 1) Systematic identification of SOA design problem, selecting an appropriate pattern and binding the design with the RBM problem specification of the pattern; 2) Systematic recording of the SOA design changes (refactoring) using the RBM pattern solution specification; 3) Automatic derivation of the corresponding performance model changes from the design model changes using coupled transformation; 4) Automatic derivation of transformation directives from the performance model changes and annotation of the performance model with the transformation directives; 5) Automatic refactoring of the performance model by QVT model transformation.

Systematic and automated pattern exploration techniques and the tools support developed in the thesis are illustrated and evaluated with a Browsing and Shopping SOA case study. A test suite was designed and used to verify all the major functionalities of the proposed approach. Furthermore, several design patterns are applied to the Browsing and Shopping SOA to validate their effectiveness in the process of performance analysis by a system designer.

Acknowledgements

First and foremost, I would like to express my deep and sincere gratitude to my supervisors, Dr. Dorina Petriu and Dr. Murray Woodside for their earnest and devoted support, encouragement and guidance throughout this work. Accomplishing this work would have been impossible without their supervision and assistance.

Furthermore, I would also like to thank Dr. Ghizlane El Boussaidi, Dr. Jean-Pierre Corriveau, Dr. Samuel Ajila, and Dr. Liam Peyton, the thesis examination board, for revising this work and providing useful help and feedbacks.

Last but not least, I would like to thank my family: My wife Salma and my daughter Melody for all their sacrifices and support during these years, my parents Esmaeil and Fariba, my sister Labkhand, and my brother Maziar. I could not have achieved this without their unlimited help and encouragements.

This work was partially supported by the Centre of Excellence for Research in Adaptive Systems (CERAS) and by the Natural Sciences and Engineering Research Council (NSERC).

This work is dedicated to my beautiful wife

Salma Attaran

my dear daughter

Melody Mani

Table of Contents

List of Tables

List of Illustrations

List of Acronyms

BPM	Business Process Model
BPEL	Business Process Execution language
BPMN	Business Process Model and Notation
BPS	Behavioral Pattern Specification
CSM	Core Scenario Model
LQN	Layered Queueing Networks
M2M	model-to-model
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OASIS	Advancing Open Standards for the Information Society
OCL	Object Constraint Language
OMG	Object Management Group
PModel	Performance Model
PIM	Platform Independent Models
PSM	Platform Specific Models
PUMA	Performance from Unified Model Analysis
QVT	Query, View, and Transformation
QVT - O	Query, View, and Transformation - Operational
QN	Queuing Networks
RBML	Role Based Modeling Language
RUP	Rational Unified Process
SASSY	Self-Architecting Software Systems
SEAM	Service Architecture Model
SIMS	Semantic Interfaces for Mobile Service

SModel	Software Design Model
SOA	Service Oriented Architecture
SOAD	Service Oriented Analysis and Design
SoaML	Service Oriented Architecture Modeling Language
SOMA	Service Oriented Modeling and Architecture
SPS	Structural Pattern Specification
SSL	Secure Sockets Layer
TD	Transformation Directive
UML	Unified Modeling Language
WS-BPEL	Web Services Business Process Execution Language

Chapter 1 Introduction

Service-Oriented Architecture (SOA) is an architectural approach for developing and deploying software applications as a set of reusable composable services. SOA provides many architectural benefits to the design of a distributed system including reusability, adaptability, and maintainability. The quality of SOA-based systems may be improved by applying SOA design patterns [1]. However, changes due to design patterns may affect performance and other non-functional properties positively or negatively [2]. Early performance analysis of SOA designs can be based on a performance model derived from the design model by using techniques such as PUMA (Performance from Unified Model Analysis) [3], which was developed in our performance research group at Carleton. This work builds on PUMA to investigate the performance impact of SOA design patterns. This chapter presents the context, motivation, objectives and scope of this work, as well as the thesis contributions.

1.1 Motivation

Service Oriented Architecture (SOA) provides many architectural benefits to the design of a distributed system including reusability, adaptability, and maintainability. A service is a coarse-grained piece of logic providing a distinct business function, which autonomously implements the functionality promised by the contracts it exposes.

SOA raises various challenges. A set of challenges is related to issues around the architectural design of service oriented systems, such as providing service aggregation and a centralized view in an environment which promotes autonomy, encapsulation, and privacy. Another set of challenges is related to non-functional properties of distributed

systems such as availability, security, scalability, performance, maintainability, reusability, understandability, robustness, etc.

Quality of SOA-based systems may be improved by applying SOA design patterns, which provide generic solutions for different architectural, design and implementation problems [1, 4]. However, besides the intended effects, changes due to the applied patterns may affect other non-functional properties, positively or negatively. This work investigates the performance impact of SOA design patterns. 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 (Figure 1.A) . In our work, the initial SModel to PModel transformation is performed with the PUMA transformation chain [3]. An outcome of the transformation is the mapping between the SModel and the corresponding PModel elements [5].

After applying some pattern(s), usually a new performance model needs to be generated by re-applying the PUMA techniques to observe the impact of the design pattern changes on the design model. However, this has the following drawbacks:

- It masks the causal connections between the design changes and the performance impact, which can provide insight to the designer if the resulting design is unsatisfactory.
- It is a substantial waste of execution cost, which could be significant if the cycle of choosing a pattern, applying and evaluating it, is repeated many times during the development process of large systems.

- It requires significant effort from the designer, possibly repeated many times if a pattern has many variations.

Successive applications of patterns may require dozens or hundreds of evaluations (including alternatives for some patterns). The transformation effort increases rapidly with SModel size (with complex traversals of the entire design), and may become excessive and even impractical. In this work we are using incremental transformation techniques, which only process the SModel changes, propagating them to PModel with less effort (not affected by the SModel scale). Most importantly, it establishes useful traceability links. This enables incremental studies of numerous design alternatives when applying a large number of SOA design patterns.

1.2 Objectives and Scope

The goal is to rapidly and systematically evaluate the performance impact of alternative design patterns, by closely coupling the refactoring of the performance model with the design model, and by automating the evaluation as much as possible.

In this work we are making the following assumptions:

- The selection of the candidate patterns for improving a certain quality of the design (such as maintainability, robustness, performance, security, etc.) is done by the designer, and is not included in the scope of our approach. Furthermore, the locus of the pattern application is also decided by the designer.
- The initial design is assumed to be expressed in UML extended with two standard profiles, SoaML and MARTE. A partial formalization of the SModel refactoring process based on Role-Based Modeling [6-8] is used in the thesis to express the

correct role bindings and to create derived bindings that are needed for the coupled transformation.

- The initial performance model is derived from the annotated design model using the PUMA transformation chain (as illustrated in Figure 1.A). An outcome of PUMA is the mapping between the SModel and the corresponding PModel elements used in our approach.

The focus of this research is on the systematic application of a given SOA design pattern, followed by automatic incremental propagation of SModel changes to the PModel, as illustrated in Figure 1.B (Model modifications are denoted using the "Δ" symbol).

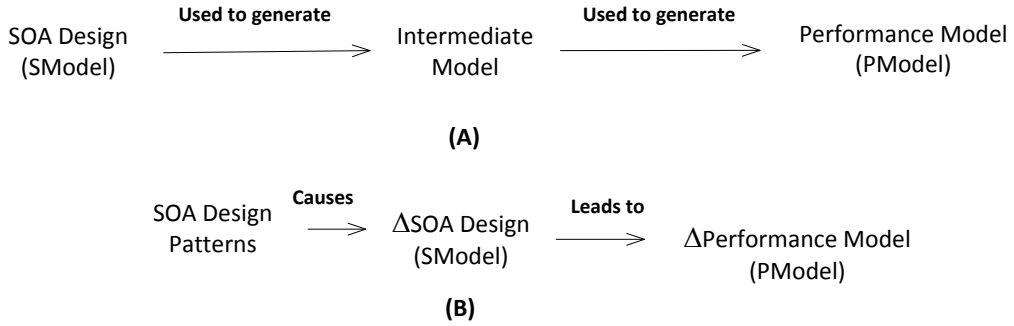

Figure 1 : (A) Initial transformation of SModel to PModel; (B) Propagation of SModel changes due to pattern application directly to PModel

The standard UML profile Service Oriented Architecture Modeling Language (SoaML) [9] is used 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 [10]. Although the context of the research is SOA design patterns, the proposed approach can be generalized to other software design patterns. Another standard UML profile used in this approach is MARTE [11](Modeling and Analysis of Real-Time and Embedded

Systems) , which bridges the gap between the software and the performance domains. The MARTE annotations help us in transforming the SModel into a PModel, and also in propagating the modifications due to the application of design patterns.

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. For studying the impact of design patterns on the system and its performance, the SOA design patterns need to be formally specified. In this research, three views are specified for each SOA design pattern: structural, behavioural and deployment views. The Role Based Modeling Language (RBML) [6] is used to formally define the two parts of each SOA design pattern view: the set of SModel elements and their relationships *before* applying the pattern which are referred to as the *problem specification*, as well as the subset of elements after applying the pattern, that constitutes the *solution specification*. The pattern application will replace some elements from the "problem" subset with elements from the "solution" subset, leaving the remaining SModel elements unchanged. The implementation of the PModel refactoring transformation that corresponds to the pattern application rules is implemented by design and implementation of a universal transformation engine using the standard language QVT (Query, View, and Transformation) Operational [12, 13] (called QVT based Transformation Engine). A mapping between SModel and PModel (created during the initial PModel derivation using PUMA) is used to propagate the changes to the PModel and establish the trace links between corresponding model elements from the SModel and PModel. The proposed approach is illustrated with the Service Façade, Service Decomposition and

SSL design pattern applied to a Shopping and Browsing system and is applied to ten SOA design patterns for evaluation.

1.3 Proposed Approach Overview

The high-level view of the SOA pattern-application process developed in the thesis, which exploits the incremental transformations is shown in Figure 2. There are two main inputs to the system (shown in dotted boxes and marked by "System Input"):

1) The SOA SModel
2) The library of pattern definitions (including the formal roles).

The actions in Figure 2 are numbered, showing the sequence of activities. Also, a dashed line separates the activities performed by the system designer from the automated processes. The system designer processes are systematically defined in this research and some tools are implemented to assist designer with them. There are four main stages:

A. Preliminaries: This stage uses the initial SModel to create and solve the base PModel using PUMA, and creates the SModel/PModel mapping table. As mentioned before, PUMA represents preliminary research [14]. Pattern application begins at step (4), where the designer selects a candidate pattern from the library presumably to improve some quality of the design (e.g. maintainability, flexibility, scalability, robustness, performance, or security). The selected pattern may have multiple variations, giving a set of candidate patterns rather than just one.

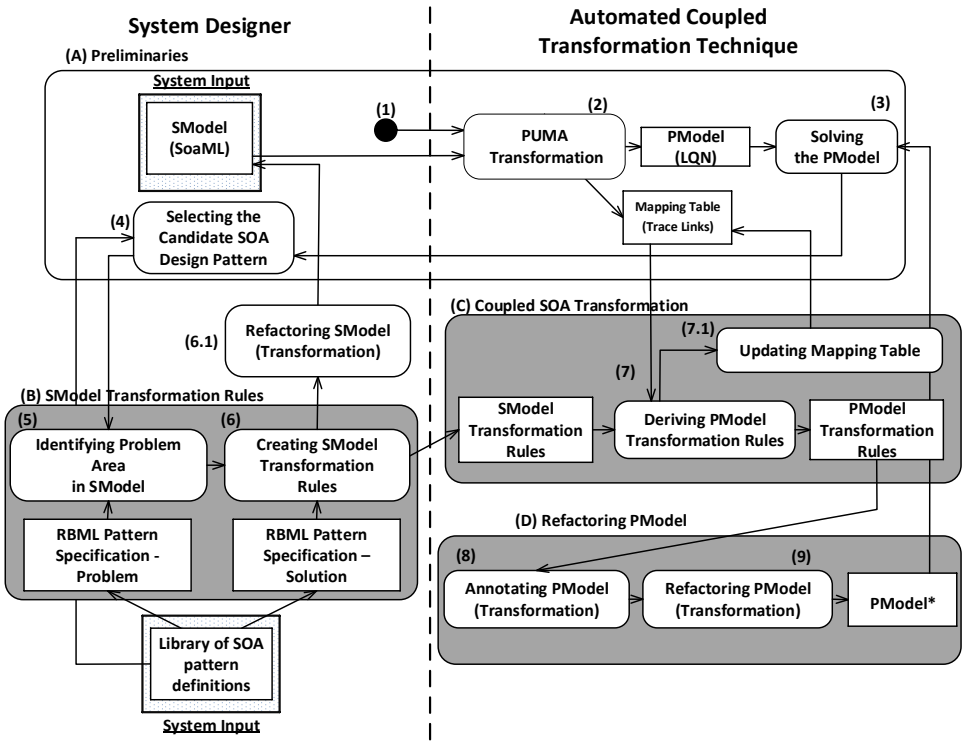

Figure 2: Overview of the proposed approach

B. SModel Transformation Rules: The selected pattern is specified using RBML in two parts: Problem and Solution Specifications. The designer indicates where the pattern is applied by binding pattern roles to entities in the SModel (step 5), specifies SModel refactoring transformation rules that will satisfy the solution specification (step 6) and will be used to refactor the SModel (step 6.1). System designer can either refactor the SModel manually or use existing SOA transformation techniques to perform the changes.

C. Coupled SOA Transformation: The transformation rules for refactoring the PModel are automatically derived from the SModel refactoring transformation rules from stage B by using the mapping table from stage A. During this process, the mapping table is also updated with the latest changes.

D. **Refactoring PModel:** In this stage, the PModel is first annotated with transformation directives corresponding to the refactoring rules, then a QVT based transformation engine traverses the annotated PModel and refactors it into PModel*. Next, PModel* can be solved again using the LQN solver and the results can be used to select another patterns to be applied. Therefore, stages B, C and D may be repeated until the system engineer gets the desired results.

1.4 Contributions of the Thesis

In general, the contributions of this thesis are summarized as follows:

1. Extension of the mapping concepts between SModel and PModel to bridge the semantic gap in the coupled transformation. More specifically, the types of traceability links mapping the PModel elements to the corresponding SModel elements from which they were generated are extended to identify sets of SModel elements to be mapped to one PModel element. These new types are not defined in the metamodel of the SModel, but simplify the SModel-to-Pmodel mapping.
2. Exploitation of the RBML definition of patterns to express constraints on correct role bindings and to create derived bindings that are needed for the coupled transformation.
3. Proposing a systematic approach with tool support that guides the designer to a correct application of the pattern, and records the SModel transformation rules.
4. Automatic derivation of the transformation rules for PModel refactoring from the SModel transformation rules using the extended mapping between the two models (hence the term “coupled transformation”).

5. Designing and implementing a QVT-based transformation engine for the automatic refactoring of PModel, capable of applying any changes corresponding to any SOA design pattern. It processes transformation directives that are generated from the PModel refactoring rules (derived in contribution 4).
6. Developing tools to support different phases of the process: a) record the SModel refactoring rules; b) automatic derivation of PModel refactoring rules; c) automatic annotation of PModel elements with transformation directives; d) QVT transformation for PModel refactoring. The proposed approach and the developed tools were evaluated by applying them to a set of SOA design patterns from the literature.
7. Application of ten design patterns to a case study SOA using the proposed techniques and verification and validation of its effectiveness and efficiency.

The following papers are the outcomes of this research work so far:

- Catia Trubiani, Antinisca Di Marco, Vittorio Cortellessa, **Nariman Mani**, Dorina Petriu, "Exploring Synergies between Bottleneck Analysis and Performance Antipatterns", Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE'14), Dublin, Ireland, 2014. [15]
- **Nariman Mani**, Dorina Petriu, Murray Woodside, "Propagation of Incremental Changes to Performance Model due to SOA Design Pattern Application", Proceedings of 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013), Prague, Czech Republic, 2013. [16]
- **Nariman Mani**, Dorina Petriu, Murray Woodside, "Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture",

Proceedings of the 37th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Model-Based Development, Components and Services (MOCS) track, Oulu, Finland, 2011 [5]

- **Nariman Mani**, Dorina Petriu, Murray Woodside, "Towards Studying the Performance Effects of Design Patterns for Service Oriented Architecture", Proceeding of ICPE'11 Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, 2011. [17]

1.5 Thesis Content

This thesis is structured as follow: The state of the Art and Background are discussed in Chapter 2 . Chapter 3 describes the process of creating refactoring rules for a SOA design (SModel) using RBML and role binding. Chapter 4 discusses the structure of the extended mapping table used as one of the key inputs to the coupled transformation technique. Chapter 5 explains the coupled transformation technique and the process of derivation of performance model refactoring transformation rules from the SOA design refactoring transformation rules using the mapping table. Chapter 6 discusses the process of annotating the performance model with derived changes and refactoring of the performance model using a QVT-based transformation engine. Chapter 7 shows the tool support for all the proposed techniques developed in this thesis. Chapter 8 discusses the case study of the thesis and also the verification and validation processes for the proposed techniques in this thesis. Finally, Chapter 9 discusses the conclusions (accomplishments, limitations and future work).

Chapter 2 State of the Art and Background

The chapter overviews the background and the state of the art related to the thesis research.

2.1 Service Oriented Architecture (SOA)

There are many definitions for Service-Oriented Architecture (SOA). Some authors defined SOA as an architecture that provides the means to integrate various systems and expose reusable business functions [18]. Newcomer et al. defined SOA as a style of system design that provides guidance for all aspects of creating and using business services during their lifecycle [19]. The OASIS (Advancing Open Standards for the Information Society) reference model [20] defined SOA as a concept for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.

According to [21], SOA architecture consists of three elements:

1) Business Architecture: defines the business goals, the business objectives and the business workflows;
2) Infrastructure Architecture: describes the components that provide the business capabilities, the structure and the behaviors of the underlying services provided by those components;
3) Data and Information: models information handled by the business processes.

Following [18] [19] [20] , this thesis defines SOA as collection of loosely coupled and autonomous components called services. A service is a coarse-grained piece of logic which provides a distinct business function and implements all the functionality promised by the contracts it exposes [1]. SOA aims to enhance the agility and cost-

effectiveness of an enterprise application by positioning services as the primary means for handling the system processes. Each service exposes processes and behaviors through contracts. SOA provides many architectural benefits to the design of a distributed system including reusability, adaptability, and maintainability.

2.1.1 Component Based Design

Component Based Design has evolved from the object oriented design paradigm [22] [23]. In the early days of object oriented analysis and design, fine-grained objects were considered to provide "reuse", but those objects are too small and there are no standards in place to make their reuse practical. Coarse-grained components have become more popular and they are the target for reuse in application development and system integration. These coarse-grained components provide certain well defined functionality from a cohesive set of finer-grained objects [23]. In [24], the authors included the notation of service in Component-Based Development for Enterprise Systems literature, describing a component as an executable unit of black box code that provides a set of physical encapsulated related services. The provided services can only be accessed through a consistent and published interface which also includes an interaction standard.

2.1.2 Service Oriented Architecture (SOA) Drivers

In this section, the types of the software systems that SOA design is appropriate for are discussed. In [25], the author defines three drivers for SOA:

- **Distributed Systems:** SOA facilitates interactions between service providers and service consumers, enabling the realization of business functionalities. According to OASIS [20] , SOA is a paradigm for organizing and utilizing distributed capabilities.

- **Different Owners:** SOA includes practices and processes that are based on the networks of distributed systems and are not controlled by single owners. Different teams, different departments, or even different companies manage different systems. Thus, different platforms, schedules, priorities, budgets, and so on must be taken into account. This concept is the key to understanding SOA and large distributed systems in general.
- **Heterogeneity:** In the past, a lot of approaches have been proposed to solve the problem of integrating distributed systems by eliminating heterogeneity such as harmonizing all the involved systems. Large systems use different platforms, different programming languages (and programming paradigms), and even different middleware. They are a usually a set of mainframes, SAP hosts, databases, J2EE applications, small rule engines, and so on and in other words, they are heterogeneous. The SOA approach accepts heterogeneity. It deals with large distributed systems by acknowledging and supporting this attribute.

2.2 Service Oriented Modeling Language (SoaML)

There are several modeling languages used in the SOA literature for modeling SOA architecture; i.e., the business and the infrastructure architectures. In this thesis, the Service Oriented Modeling Language (SoaML) is used for defining of SOA design or SModel. The SoaML specification is an OMG (Object Management Group) standard [9] for the design of services within a SOA system [9, 10, 26]. It is defined as a UML profile that provides a standard way to architect and model SOA solutions. In this section, a brief description is provided. OMG took on SoaML in 2009 after a three years process involving multiple participants from both industry and academia. SoaML is based on the

former experience, methodologies and products of SOA experts and is designed to support SOA best practices while standardizing related terms and notations [27]. SoaML has the ability to define the service structure and dependencies, to specify service capabilities and classification, and to define service consumers and providers [28]. 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 [6]. In this research, a BPM diagram is specified using the UML activity diagram notation [29], whose semantic is close to that of business processes. 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 is specified in a business process diagram and refined in regard to the 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.

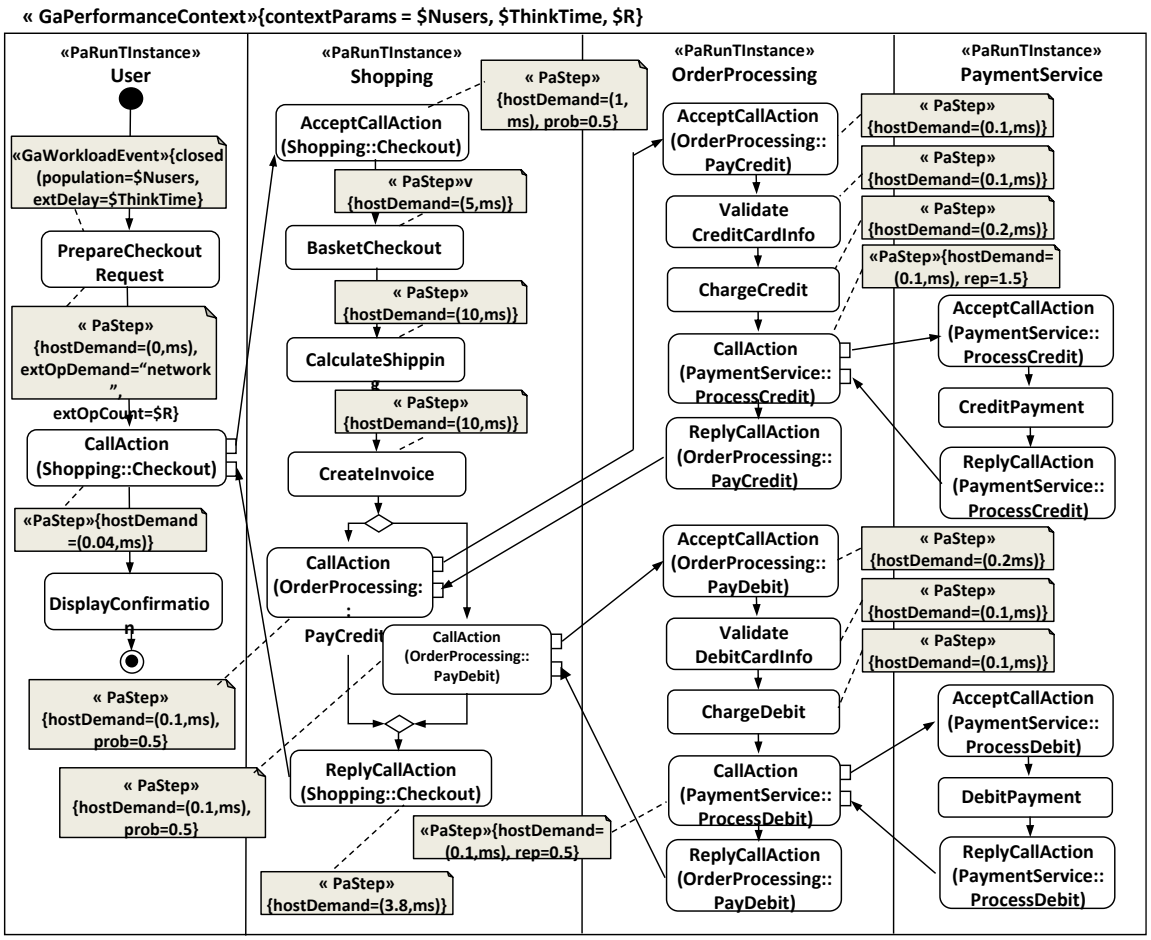

Figure 3: Checkout Business Process Model

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 [10]. Each participant plays either the role of Provider or Consumer with respect to a service. For instance, in Figure 4 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.

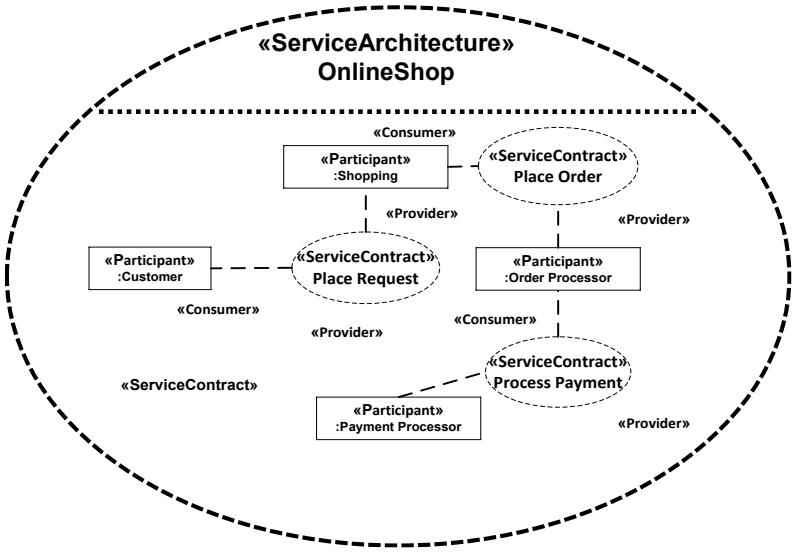

Figure 4: Service Architecture model (SEAM) for the Online Shop case study

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 fourth step, using the Service Composition or Components diagram, is about assembling and connecting the service participant's 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, this 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 [5].

Beside the UML activity diagram [30] which is used in this thesis, the Business Process Model and Notation (BPMN) [31] and the Business Process Execution language (BPEL) [32] are also used in the literature for the Business Architecture modeling of a SOA system. Authors of [33] [34] proposed some approaches using the UML profile to produce Platform Independent Models (PIM) which are transformed into Platform Specific Models (PSM) using a defined SOA profile. The PSM is used to generate middleware independent code, which is transformed into an executable code based on the target middleware. The authors of [35, 36] used Model Driven Architecture (MDA) to automatically generate executable Web Services Business Process Execution Language (WS-BPEL) from various UML model views.

There are also general purpose model-driven methodologies such as the Rational Unified Process (RUP) [37] and KobrA [38] which emphasize on the systematic use of models as primary software artifacts throughout the software engineering life-cycle. However, some of the main characteristics of service engineering (i.e. core concepts such as service, contracts etc.) are not covered by them.

The survey [39] overviews other works on SOA design methodologies. Service Oriented Analysis and Design (SOAD) [40] developed by IBM is one of the first systematic approaches to service engineering which was also further developed in Service Oriented Modeling and Architecture (SOMA) [41] . The SoaML approach is close to the process of SOMA. There is also the work in [42], Semantic Interfaces for Mobile Services (SIMS), which defines a top-down approach for specifying mobile services.

2.3 MARTE Performance Annotations

In this research, performance information is added to the UML+SoaML specification by adding annotations defined in the OMG standard profile MARTE [11, 43]. 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 composed of steps and a workload *«GaWorloadEvent»* attached to the first step. Concurrent runtime component instances *«PaRunTInstance»* correspond to activity diagram partitions (also known as swimlanes). *«PaStep»* represents the execution of an activity or an operation invoked by a message, and has attributes *hostDemand* for the required execution time, *prob* for the probability of the step, if it is optional, and *rep* for the repetitions if it is repeated. The workload *«GaWorloadEvent»* in Figure 3 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, a default deployment is assumed, such as one host node with one concurrent process for each instance. 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 add a "performance" dimension to UML models, in which is used for the

transformation of SModels into PModels and the propagation of changes due to the application of design patterns.

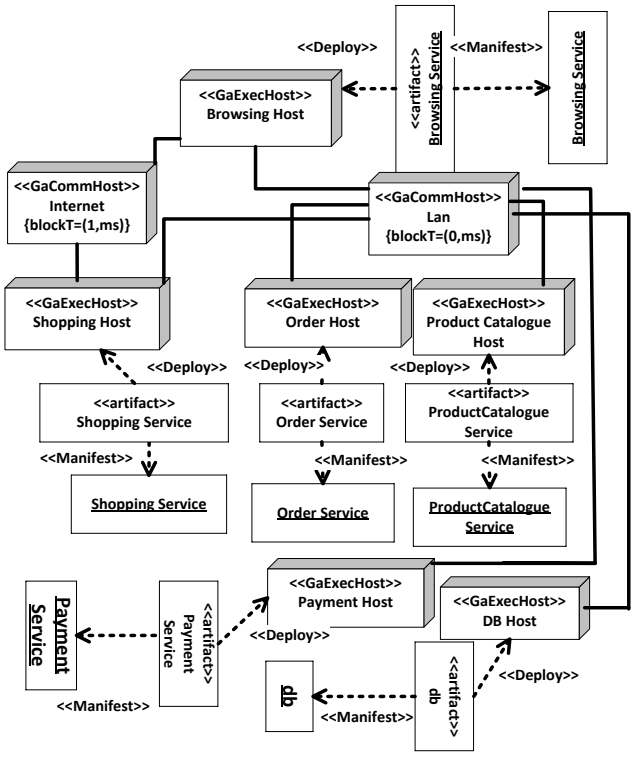

Figure 5: SOA Deployment Diagram

2.4 Performance Model in LQN (PModel)

There is a significant body of research on the derivation of a PModel from the design model (SModel) of a system [44-46]. The OMG standard profiles for UML performance annotations, SPT [47] and MARTE [11], have enabled research to transform UML design specifications into many kinds of performance models, based for example on Queuing Networks (QN) [46, 48], Layered Queueing Networks (LQN) [3, 49, 50], Stochastic Petri nets [51, 52], etc. The Palladio tool suite also evaluates performance models of software designs expressed in the Palladio Component Model, a UML-like framework, using simulations and LQN [53].

The proposed technique in this research is established based on PUMA [54]. PUMA is a set of transformations which take as input different SModels and transform them into different performance models [3, 49]. 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 [50], 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[14]. Alhaj et al [14, 55-57] have used PUMA to analyze service architectures described by PUMA, with automated completions to incorporate platform dependent elements. They extended PUMA and developed a model transformation chain called Performance from Unified Modeling Analysis for SOA (PUMA4SOA) which automatically generates performance models from the UML software design models of SOA systems with performance annotations.

The work in this thesis uses a mapping table containing traceability links which are established by PUMA4SOA. These traceability links and the concept behind them are discussed by the authors in [58] and the mapping table containing these links is discussed in Chapter 4 below. The mapping table is used to propagate the design pattern changes from the SModel to the PModel.

This work constructs performance models (PModels) in an extended queuing format called LQNs [3]. An LQN represents congestion in waiting for service provided by host processors and also by software servers. The LQN metamodel is shown in Figure 6 [59] .

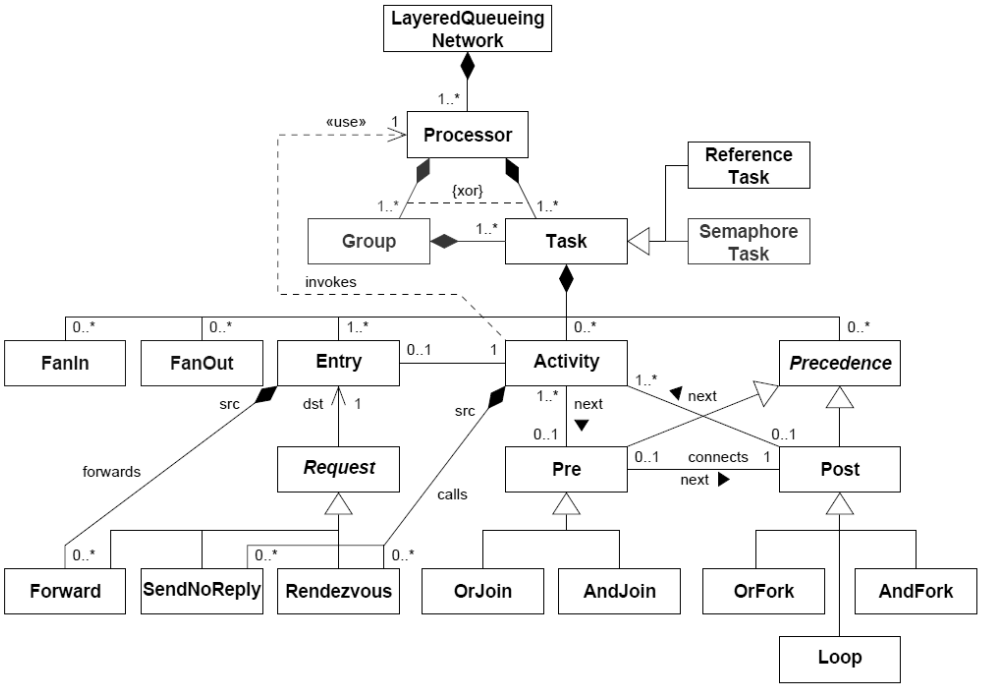

Figure 6 : LQN Metamodel [47, 59]

Figure 7 shows the LQN model corresponding to the SoaML business process in Figure 3 , SEAM in Figure 4, and the deployment diagram in Figure 5. 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 the OrderProcessing task in Figure 7) and the entry has a parameter for its host demand, equal to the *hostDemand* of the corresponding 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 with no reply). The arrow is annotated by the number of calls per invocation of the sender (e.g. (3)). For deployment, the host processor is indicated by an oval symbol attached to each task.

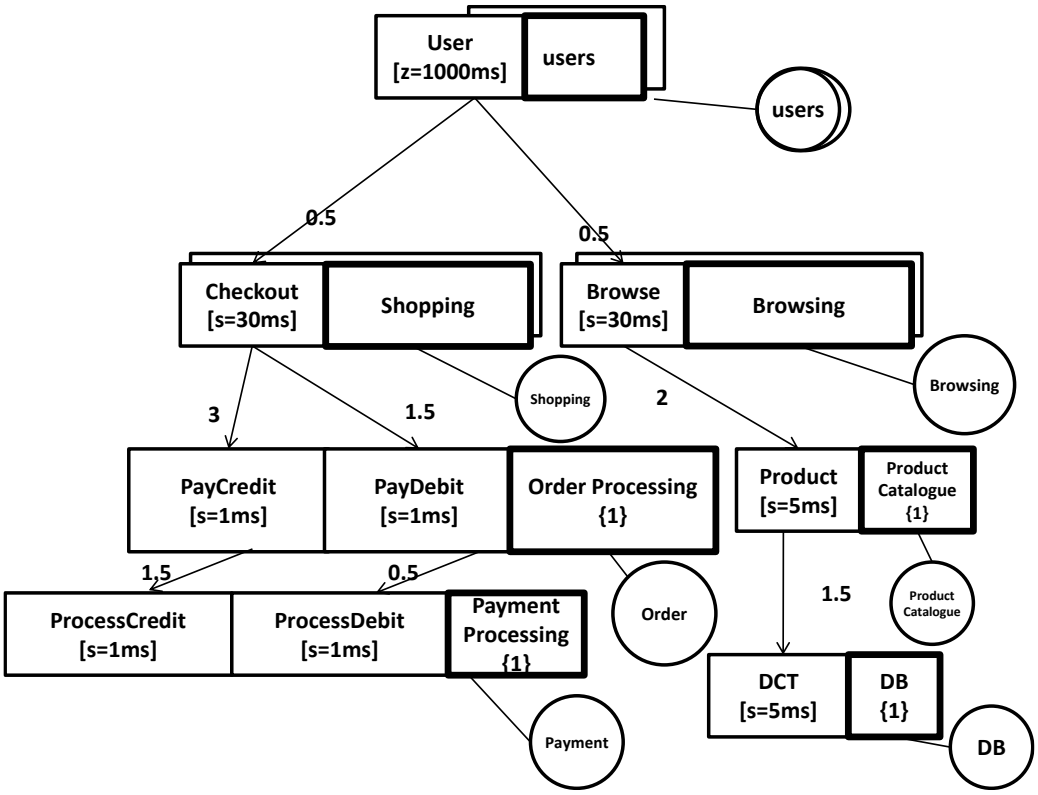

Figure 7: LQN Performance Model (PModel) corresponding to Figure 3 (BPM), Figure 4 (SEAM), and Figure 5 (Deployment)

2.5 Design (Anti)Patterns impact on Software Performance

The impact of design patterns on software performance has been studied mostly through the use of performance anti-patterns, introduced by Smith and Williams [60]. Anti-patterns are common design mistakes that cause undesirable results.

Cortellessa et al. present an approach based on anti-patterns for identifying performance problems and removing them [61]. In [61] the anti-pattern detection is done through OCL queries. A set of rules are defined from the informal representation of the problem on UML and MARTE. The rules are first described in a semi-formal natural language and then formalized using OCL queries. This requires an OCL query for each anti-pattern which needs to be detected inside the model. Furthermore, the authors of [61] stated that

the anti-pattern removal is not automated and each anti-pattern requires its own specific removal method. Arcelli et al. [62] introduced the RBML specification of anti-pattern problems and solutions, but pattern application is not automated.

Parsons and Murphy [63] introduced 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. The approach is dependent on the implemented system and its runtime behavior.

Menascé et al [64] present a framework SASSY (Self-Architecting Software Systems) which allows designers to specify system requirements in a visual activity-based language and automatically generates 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 architectural patterns related to quality of service and thus to performance. This work defines its own visual activity-based language and does not focus on UML based languages such as SoaML.

Xu [65] applied rules to performance model results to diagnose performance problems and to propose solutions for fixing them. This results in automated rule-driven changes at the PModel level, which could be interpreted to suggest corresponding design changes. However, the rules are different from patterns, and the changes were not propagated automatically to the PModel.

Kassab et al. [2] proposed an approach based on goal modeling for a quantitative evaluation of the support provided by architectural patterns and tactics for a given set of

quality attributes, such as performance and security. Our approach is different, being focused on performance and using specialized performance models for a more thorough analysis of performance characteristics under changes due to the application of patterns.

This thesis uses RBM for the specification of the SOA design patterns and the OMG standard language QVT-O for refactoring the PModel according to the pattern. Another difference is the exploitation of the "coupled transformation" concept to automatically derive the PModel refactoring transformation from the SModel refactoring transformation. To the best of our knowledge, none of the works in literature addresses the problem as proposed in the thesis.

2.6 Traceability

In the model-driven software development life cycle, different kinds of models are created, updated, transformed and/or deleted. The dynamic use of models becomes challenging when it comes to the ability of maintaining the consistency between different models. Establishing relationships between the elements of different models by means of cross-model trace-links enhances the maintainability of the software under development. In a more general sense, traceability is defined as a software mechanism for creating relationships between different software artifacts during the software life cycle, which is used to understand the relationships and dependencies between the software artifacts involved in the development process. There are a few traceability approaches which are focused on software modeling. For instance, in [66] the traceability approaches are categorized in three main classes:

1. Requirements: traceability is the ability to trace a requirement to different models throughout the entire life cycle;

2. Modeling: traceability is the ability to follow the relationship between elements in different models;
3. Transformation: traceability is defined as trace-links between the elements of the source and the target model in a model transformation.

Traceability techniques are categorized in [67] based on storage and manageability into:

1. Embedded traceability: trace-links are defined as new model elements within the same models they are linked to;
2. External traceability: the trace-links are stored in an external new model to maintain traces separately from the models they are linked to.

Traceability approaches are categorized in [68] based on the types of the trace-links into:

1. Explicit traceability links: trace-links are expressed directly in the models using the abstract syntax of the language (e.g., UML dependencies);
2. Implicit traceability links: trace-links are hidden, using model management operations (e.g., traces maintained by the model transformation engine).

In this thesis, the traceability links constitute an external model, separate from both the source and the target models. They are stored in a mapping table as an outcome of the initial PUMA transformation of the input SOA design model into a LQN model. Therefore, according to the above classification, our traceability approach falls under the "transformation" class, is "external" and the trace links are "explicit". Further details about the content of the mapping table in the thesis are provided in Chapter 4 .

2.7 SOA Design Patterns

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 [69-72]. A single design problem can be resolved by many solutions, which may be expressed by different pattern variants. In this work, the assumption is that one problem has one solution; meaning this thesis treats each pattern variant as a new pattern.

In this thesis, the RBML formal specification of these patterns (discussed in Section 2.8) helps the system designers to systematically identify the place in a SModel where a pattern should be applied.

This section describes a pattern called "Service Façade" from the category of service implementation patterns [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

by providing an interface for 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, Figure 4, and Figure 5 and trace its performance impact on the PModel of the case study.

There are several ways that the façade pattern could be applied to the SOA specification. For example, the façade can be implemented into a new service which acts as intermediate interface for a core service in the SOA. Also, a façade can be implemented as a new task inside a service which protects the core logic of the service from future changes. Furthermore, there is a choice of implementing the façade on a dedicated execution host or the same as the core service. All of these decisions must be made by the system designer prior to the application of design pattern. If the design pattern description does not provide the details of the choices, then the task of the system designer becomes harder on making those decisions.

In general, if there is a service core which is prone to future changes, a service façade can help to protect it. In our case study, 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.

2.8 Pattern Specification using Role Based Modeling Language

Informal descriptions of the design patterns (similar to the one discussed in Section 2.7) are useful for communicating proven solutions to the development team, but they lack the formality needed to support precise specification and automated tools. Formal pattern specification languages using a mathematical notation (e.g., see [73, 74]) provide the desired precision but demand sophisticated mathematical skills. For a better match to software engineering skill sets, this thesis uses a Role-Based Modeling (RBM) approach similar to [6], which was used for a related purpose (i.e., specifying performance anti-patterns) in [75].

A pattern specification using RBM consists of a Structural Pattern Specification (SPS) that describes its structure, and a set of Behavioral Pattern Specifications (BPS) describing its behavior. In [6], France et al. extended the SPS concept for class diagram views of pattern solutions. They also define subtypes of UML metamodel classes describing class diagram elements (e.g., UML metamodel classes Class, Association) and specifies semantic pattern properties using constraint templates. Also the BPS concept for UML sequence diagram was extended in [6] as follows: a BPS consists of an interaction role that defines a specialization of the UML metamodel class Interaction and is a structure of lifelines and messages.

The RBML is used for specifying SOA design patterns (specified by SoaML). SOA design patterns are at a higher level of abstraction than some mid-level OO design patterns (e.g. Visitor and Decorator). SOA design patterns mainly address concurrent

entities (the processes and threads), their inter-communication and sharing of resources, the service invocation and the steps they perform. In this research uses three views to describe SOA design patterns: collaboration diagrams are used for SPS (as in the service architecture model) and activity diagrams for BPS (as in business process models) as SoaML is using UML behavior modelling (i.e. activity, interaction or state machine) for service choreographies [10]. Also the deployment diagram is used for deployment view.

In [7], the Role Model was defined as a structure of meta-roles (henceforth called roles). A role defines properties that determine a family of UML model elements (e.g., class and generalization constructs). The type of model elements characterized by a role is determined by its base, where a role base is a UML metamodel class (e.g., Class, Generalization) [76]. Although SoaML uses BPM for specifying the service behavior, this thesis uses UML activity diagram as it has the closest notation for specifying the processes in the SOA.

Bachman and Daya [76] first introduced the role concept in the object oriented community. Since then, there has been considerable work done on using roles for object oriented data modeling [76-84]. There are also other techniques for formal presentation of design patterns. Lauder and Kent [85] proposed an approach for presenting patterns visually using graphical constraint diagrams for describing the patterns in three layers of models: role-model, type-model, and class-model. However, the graphical form of constrains is not currently integrated with the UML and it is not clear how tools can support the notation. El Boussaidi et al. [86] tackle the problem of understanding the design problems solved by patterns; analyzing the representability of these design problems and how they can be expressed in terms of problem models; as well as

evaluating different problem model representations for existing patterns. Guennec et al. [87] proposed a UML metamodeling technique in which pattern properties are expressed in terms of metacollaborations that consist of roles that are played by instances of UML metamodel classes. However, the approach does not address the specification of semantic pattern properties (e.g., behavioral properties) and the characterization of UML behavioral models.

2.9 Model Transformations

In this research, the SOA design patterns are applied to the SOA design and the performance model via model transformation and refactoring techniques. In Section 2.9.1, some existing work on SOA design transformation and refining its corresponding performance model are discussed. Then in Section 2.9.2, the standard model transformation language used to implement the proposed refactoring technique and its related work are presented.

2.9.1 Refactoring the SModel and the corresponding PModel

Martin Fowler [88] defined refactoring as a change to the structure of a software design to make it easier to understand and cheaper to modify without worsening the user experience regarding the expected behavior. A lot of existing work on refactoring is using declarative rules to detect refactoring opportunities and to apply appropriate refactorings. Ghannem et al. [89] address the problem of detecting the best refactoring opportunities by searching the space of possible refactoring solutions supported by heuristic methods. In other approaches the need for refactoring arises when a pattern or symptom can be recognized in the design or code, which can indicate a potential problem; solving the problem by applying the pattern can lead to a potential improvement. The application of

SOA design patterns to a candidate SOA design model is such an example. If any locus in the candidate SOA design matches the problem description of a SOA design pattern, the respective design model can be refactored by applying the pattern recommend changes.

One of the challenges in performance modeling is the accuracy of the model. A solution for improving the prediction accuracy of performance models was introduced by Woodside et al. [90] as "completions". The authors proposed that additional information (which they called "performance completion") is necessary to evaluate a software specification for its performance potential. The authors claim that completions for evaluating other attributes, such as reliability or security, are also possible although their main focus is performance. In [90] is described how completions are added to a specification regardless of the language used. The completions are defined as changes which can be inserted into the speciation either by annotations, refinement, transformation and parameters, although the detailed process for each approach was not described. In a different work, Petriu et al. [91, 92] proposed a transformation technique to study the performance impact of security aspects of software systems. This work focuses on aspect models, which are individually transformed from UML to CSM and composed with the primary model; the result of the composition is transformed from CSM into a LQN performance model for analysis. Further security changes to the aspect models requires the reconstruction of the CSM and performance model.

The work in this thesis studies design patterns as a means of specifying software changes. The RBML specification of SOA design pattern helps the system designer to identify the problem in the design and to use the pattern solution to refactor the design model.

2.9.2 Query/View/Transformation (QVT)

As discussed in Section 1.3, QVT-O is used in this research for refactoring the SOA performance model in order to fulfill the design pattern requirements. QVT is a OMG standard model transformation language [12] with few available implementations and very limited documentation. The only stable implementation of QVT-O we could find is provided by the Model-to-Model (MMT) project under Eclipse [93]. However, even this QVT-O implementation only partly conforms to the adopted standard, since some concepts are not implemented and some are extended.

The Object Constraint Language (OCL) [94] is a constraint language that can be used in conjunction with any modeling language whose metamodel is specified in OMG's Meta Object Facility (MOF) [95], including UML. QVT [12] is a standard language for model transformations. It is developed by the OMG and can be used to transform any MOF-compliant model into another MOF-compliant model. The QVT specification integrates the OCL and contains three languages which are QVT Core, QVT Relations and QVT Operational (QVT - O). QVT core is defined as "a small model/language which only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models" [12, 13]. QVT core is a declarative language and it only has an abstract syntax. QVT relations language is also declarative, but it offers more complex object pattern matching and has a concrete syntax. QVT-O extends both relations and core, but also it is imperative and offers a procedural concrete syntax.

QVT is used as a model transformation technique for many model refactoring works. For example, in [96] an automatic refactoring tool for UML models is proposed. The tool was developed as an Eclipse IDE plugin which works with the Eclipse Papyrus UML

editor and utilizes the QVT-O transformation language to implement UML model and diagram refactoring.

In [97-99], an approach to assist the evolution of design patterns by model transformation technology using QVT is presented. The paper provides a formal foundation for the approach by defining the predicates that can be used to describe the properties of each design pattern, software system design, and design pattern evolutions. In [100] an approach for a simple criterion of semantic preservation is presented, as well as a technique for proving the semantic preservation of refactoring rules that are defined for UML class diagrams and OCL constraints. The work focuses on OCL annotated models, so that any changes made for refactoring a model are automatically reflected in OCL constraints.

QVT and its model querying capability are also used in design pattern detection research. In [101], a pattern is modeled with a Visual Pattern Modeling Language (VPML), a design pattern specification modeling technique proposed by the authors, and mapped to a corresponding QVT-Relations transformation. Such a transformation accepts an input model where pattern occurrences are to be detected and reports those occurrences in a result model. The focus of the work in [101] is on design pattern detection, and not on model refactoring by pattern application.

In this thesis, QVT is used to refactor LQN performance models, according to the changes produced by the application of a pattern in the corresponding SModel. Two QVT-related challenges are addressed in the thesis: a) the QVT transformation must be generic, i.e., able to process *any* set of refactoring rules corresponding to *any* pattern; b)

the refactoring rules are automatically generated and must be presented to the QVT transformation such that it can be read and processed.

2.10 Coupled Transformation

The research in this thesis proposes a coupled transformation technique to refactor the performance model. The coupled transformation technique means that the knowledge from the SOA design refactoring transformation is used to perform the second refactoring transformation on the performance model. This is possible because the performance model itself is related to the SOA model by a transformation from the SOA to the performance model. Coupling the transformations ensures that the performance analysis remains synchronized with the software changes, and relates the resource and performance changes back to the design pattern. In this thesis, a SOA design transformation is systematically defined by the designer (with tool support) for the application of design patterns to SOA design models. The transformation rules can be recorded for all types of SOA design diagrams, such as BPM, SEAM and Deployment diagram. A command-based coding system is defined for systematically recording the SOA design changes. Based on the recorded transformation codes from the SOA design transformation and the mapping table between SModel and PModel, the performance model transformation rules are generated. Therefore the transformation of the performance model is coupled with the SOA design transformation. To the best of author's knowledge, there has been no previous coupled transformation technique to keep in synchronization the evolution of two models related by a given model transformation.

The term coupled transformation was initially introduced in [102] for adding details to the performance prediction model through a transformation derived from a

transformation chain used to create the system's implementation. The author of [102] called this a "coupled transformation" as the transformation producing the performance model was coupled to a model-to-code transformation. In this thesis the PModel refactoring is coupled with the SModel refactoring based on the mapping between the two models (more specifically, on the mapping table obtained when PModel was generated from SModel by the PUMA model transformation).

2.11 Software Co-evolution

Evolution and co-evolution are critical in the life cycle of all software systems, so an increasing number of evolution mechanisms and tools are being developed. There has been work in the literature aiming to categorize and compare the change support offered by these various tools and techniques. Earlier work has proposed taxonomies of software change that focused on the purpose of the change (i.e., the *why*). Another view of the domain is taken in [103], by focusing more on the technical aspects, i.e., the *how*, *when*, *what* and *where*, of software change. However, this taxonomy considers only the traditional software development process and does not extend to model-driven development.

In the model-driven domain there is a lot of interest in evolution and co-evolution of related modeling artifacts (such as model, metamodels and transformations). Existing research considered different co-evolution cases such as: a) the co-evolution of model instances with metamodel changes [103, 104], and b) the co-evolution of a transformation with metamodel changes [105]. In both cases, there are situations where the changes can be propagated automatically, while others require designer intervention.

Our approach in this thesis is different, as we investigate a special case of co-evolution of a software model (SModel) and of a corresponding performance model (PModel), where the latter has been generated from the former by a given model transformation. The cause of the change is the application of a design pattern to the SModel, which is realized by refactoring the SModel. The SModel changes will be propagated to the PModel via a coupled refactoring of the PModel. In this work we will show how the PModel can co-evolve with the Smodel via coupled refactoring transformations.

Chapter 3 : Refactoring a SOA Design Model

The first step toward the application of a design pattern to a SOA model is to identify the design problem to which the pattern can be applied. Usually, the problem is located by following the instructions provided by the design pattern description. Design patterns are usually described in three distinct sections, Problem, Solution and Application Rules.

Pattern specification languages can make this task easier by providing a generic but visual presentation of the problem area. Furthermore, they can also provide a visual specification of the pattern solution, showing what that part of the model will look like after the application of the design pattern. RBML is used in this research for problem and solution specification of the SOA design patterns. Pattern specifications are presented by role based models (i.e. RBML) and the model binding is used for identifying the model artifacts involved in the problem. The process of specifying a design pattern using RBML was initially introduced in [6-8]. In this thesis, the process is adopted for specification of the SOA models presented by SoaML. A systematic approach will be described to help the system designer to record SModel changes in form of SModel refactoring transformation rules using the RBML presentation of pattern problem and solution and also the role binding between the RBML model and the SModel. Figure 8 shows an overview of the proposed techniques described in this chapter.

Although the process of binding the roles and creating transformation rules from RBML problems and solutions cannot be fully automated due to its uncertainties (i.e. pattern specification's level of abstraction, system designer choice of elements to play the roles), it can be made systematic, and constrained to conform to the pattern definition.

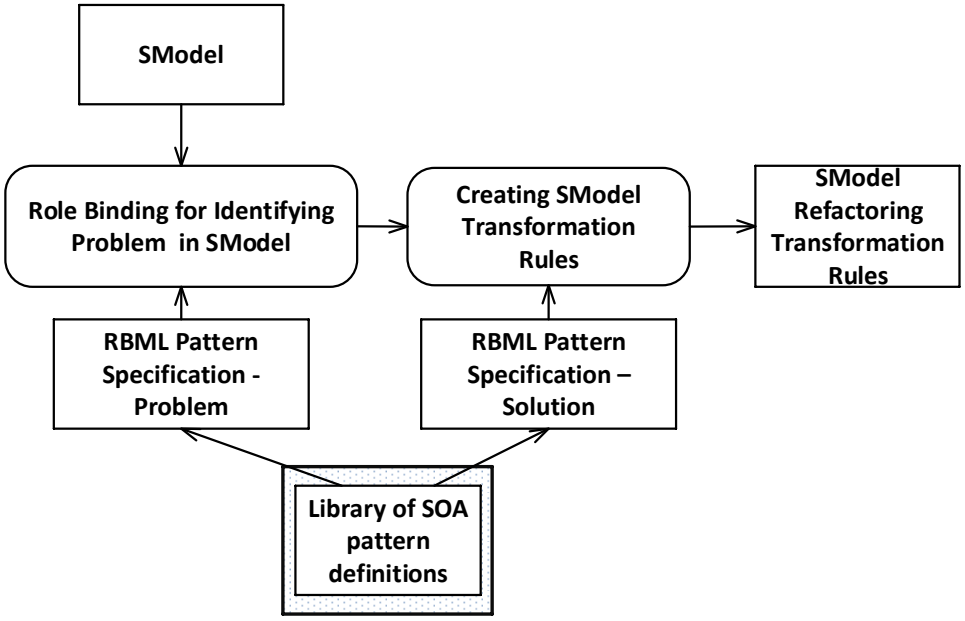

Figure 8: Role Binding and Recording SModel Refactoring Transformation

The description of role-based models for pattern description is provided in Section 3.1. The problem identification and role binding is defined in Section 3.2. Finally, the process of recording SModel refactoring transformation rules from pattern specification is described in Section in 3.3.

3.1 Role-Based Models for Specifying SOA Design Patterns

SOA design patterns describe generic solutions for architectural, design and implementation problems, which are independent of the applications to which the pattern will be applied. In RBML, the pattern specifications are expressed in terms of generic roles, which act as formal parameters that must be bound (before applying them to a specific application) to actual parameters from the application context. A role based model is made up of classes called Role Classes and relationships called Role Relationships.

3.1.1 Structure of a Role Class

Figure 9 shows the structure of a single role class. A 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 defines the number of classifiers playing the role.

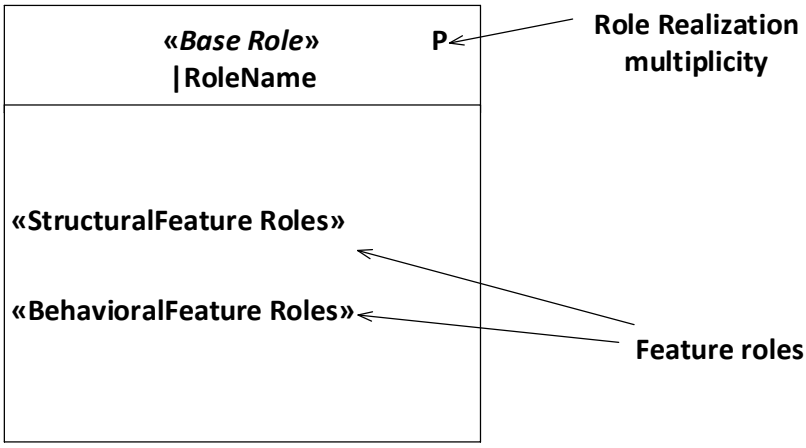

Figure 9: Structure of a Role Class

The second section contains feature roles that determine a set 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) StructuralFeature roles that specify the properties that are played by attributes.
2) BehavioralFeature roles that specify properties that are played by operations.

3.1.2 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 10.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 "shop" relationship with the "Shopping" service.

3.1.3 Structural Pattern Specification (SPS) and Behavioral Pattern Specification (BPS)

RBML is used for specifying both the structural and behavioral aspects of SOA design patterns. Therefore, each problem and solution specification contains a Structural Pattern Specification (SPS) and also a Behavioral Pattern Specification (BPS).

The SPS is used to represent the static aspects of a SOA design pattern. In pattern specification using SPS, each SOA participant (e.g., providers and consumers) can be considered as a role. An example of service façade SPS is shown in Figure 10. By convention, we consider that the names of generic roles start with the character "|" (similar to [6]). Figure 10.A SPS presents a role for core service logic. In Figure 10.B, the "Shopping" service is playing that role (assuming that this was the decision of the designer). The dashed arrows indicate the bindings between the role class and the role

relationship, and their players. The solid arrows represent derived bindings, which are discussed in Section 3.2 below.

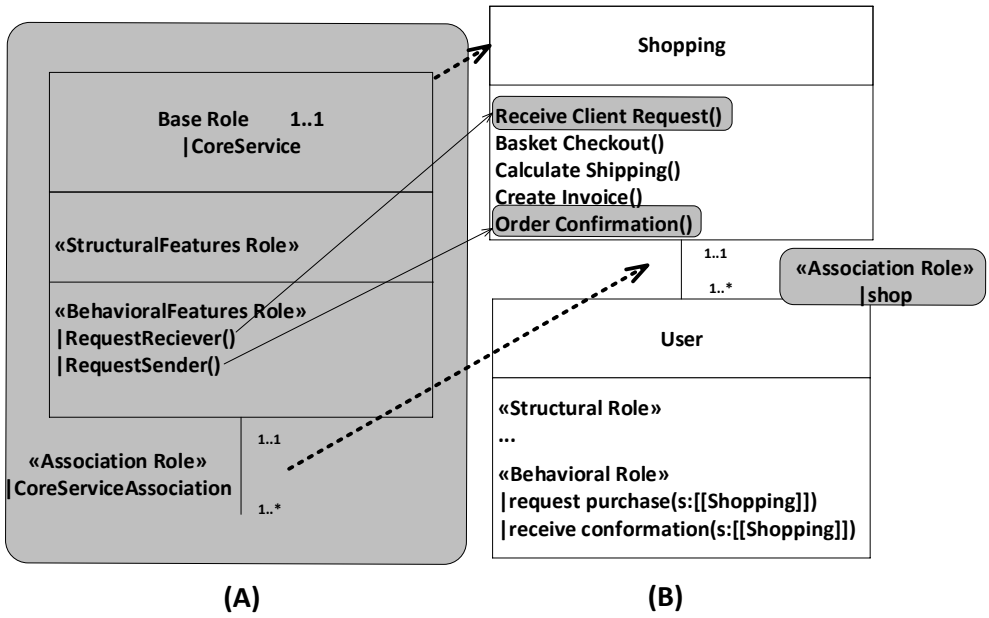

Figure 10: (A) SPS of a coupled service in Service Façade (B) Conformance to coupled core service in Shopping example

In SOA systems, the architecture is represented not only with class diagrams, but also with collaboration diagrams, according to SoaML. Therefore, this thesis uses SPS specifications in the form of collaborations as well (e.g. Figure 11).

Also, the BPS is used to present the interactions between participants in SOA behavior diagrams (e.g., a BPM diagram). A BPS is defined using the roles specified as formal parameters in the SPS for specifying the participants (the swimlanes) and may add other roles to specify some of their actions. Figure 11 shows the three views for the role-based specification for the Service Façade pattern: structural specification (SPS) at the top, behavioral specification (BPS) in the middle and deployment specification at the bottom.

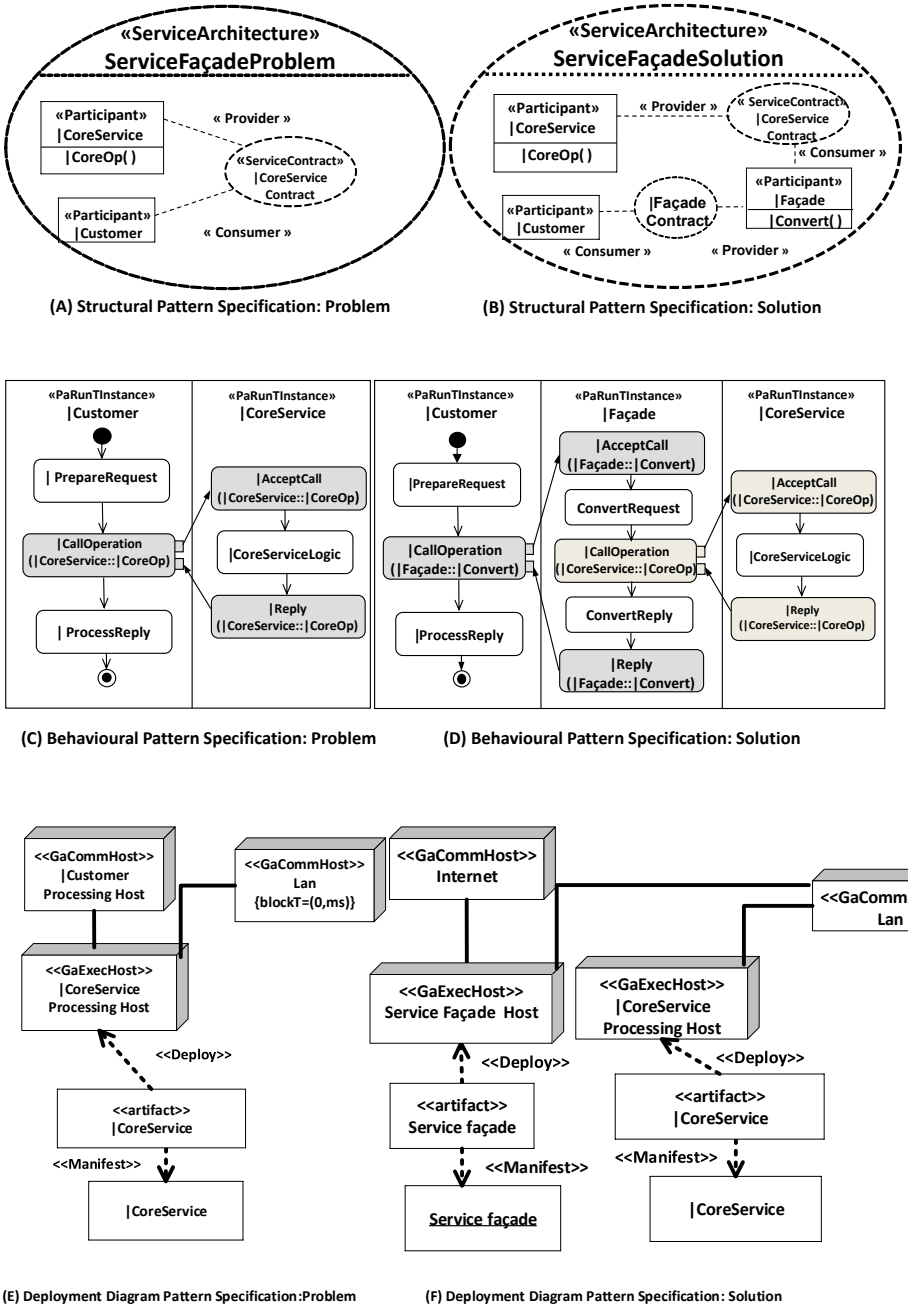

Figure 11: Service Façade pattern specification for SEAM and BPM: (A) Structural problem specification; (B) Structural solution specification; (C) Behavioral problem specification; (D) Behavioral solution specification; (E) Deployment Diagram Problem Specification; (F) Deployment Diagram Solution Specification

In Figure 11, the left side of the figure represents the problem the pattern is meant to solve (before the pattern application), and the right side represents the solution (after

pattern application). In the SPS, generic roles are played by the service participants, their operations and service contracts, while in the BPS generic roles are played by participants (associated with swimlanes), operation behavior (described by an activity sub-graph starting with *AcceptCallAction* and ending with *ReplyAction* in the swimlane of the callee) and operation invocations (*CallOperationActions*).

3.2 Problem Identification and Role Binding

A system designer decides what pattern is to be applied, takes the RBM definition from a library of pattern definitions, and binds the elements in its Problem Pattern Specification (for both Structural Pattern Specifications (SPS) and Behavioral Pattern Specifications (BPS)) to the elements of the SModel.

A SModel element can be bound to a RBM element if:

1. The SModel element type matches the RBM element type.
2. If there are constraints defined for the two matching elements, then the constraints must be compatible. We consider that the constraints are compatible if the pattern application will not constrain the SModel. This is true if the constraints specified for an RBM elements (call them CR) are the same as or looser than the constraints for the SModel elements (call them CS). Either set of constraints may be empty. In UML, a constraint is an extension mechanism for refining the semantics of a UML model element by expressing a condition or a restriction to which the model element must conform. Constraints for the elements can be specified in many ways such as Natural languages, Programming languages, Mathematical notations, or Object Constraint Language (OCL). Because of this generality, this research depends on the judgment of the designer to determine the compatibility

of constraints. In general three different scenarios are expected, based on the intersection of RBM and SModel constraints. Formally, regarding CR and CS as the sets of systems that satisfy each set of constraints, then every system that satisfies CS also satisfies CR, if $CS \subseteq CR$ (CS is included in CR), and we will say that CS satisfies CR.

a. If CS satisfies CR then the constraints are compatible, and the binding does not impose additional constraints on the SModel.
b. If CS does not satisfy CR, then some additional refinement would be implied by binding the pattern. Then the pattern cannot be bound, unless the extra refinement or restriction defined by the RBM constraint is also applied to the SModel first. The designer can make this decision.

3. The SModel element has all the attributes and operations defined by the RBM element. The SModel can have more attributes and operations than the RBM element and it will be up to the system designer to establish the bindings between them.
4. For an SModel behavioral view, it must be possible to bind its behavior to the elements of the RBM-BPS. The execution flow and the ActivityPartitions of actions must match.

Not every element in SPS and BPS is a role, so two types of bindings are defined:

1. Role Binding: Defined for the role elements in RBM. In role binding, the binding is established between a RBM role and a SModel element or a set of them (which play the role). The binding on actions can be one-to-many.

2. Derived Binding: Defined for RBM elements which are not defined as roles and will be referred to as “Other Elements” hereafter (e.g. Calls, Replies, Attributes). The bindings for these elements are derived from the role binding of role elements they are attached to: either to a sole element (e.g. Attributes) or to more elements (e.g. Connectors):
    a. For Other Elements attached to a sole role element, the binding is derived from the role binding of the sole element. For example, if a role binding is established between RBM role and a SModel element, then a binding can be also derived for their multiplicity attributes.
    b. For Other Elements attached to many role elements, the role bindings of all the participants are required for the derived binding to be established. For example, if roles at origin and destination of a connector in RBM are bound to SModel elements using role binding defined above, then a binding can also be derived for the connector in RBM and SModel. (It is assumed there is only one connector).

3.3 Creating SModel Transformation Rules using Pattern Solution

Once the subset of SModel elements matching the pattern problem is bound with the RBM problem specification through role binding, the pattern can be applied to the SModel. This means that refactoring actions can be created by the designer according to the RBML solution specification. The refactoring transformation is described and recorded in terms of actions to add, delete, and modify model elements. Specific rules are defined for specific element types (e.g. addAssoc/deleteAssoc for adding/deleting associations). The actions transform the SEAM by applying these rules to services and their interactions, and transform the BPM by applying them to ActivityPartitions

(swimlanes), Activities, Actions and ActivityEdges. Later on, SModel transformations are used in a fully automated process to derive the PModel refactoring transformation actions.

Three kinds of SModel transformation actions are created, depending on the relationship of the elements in the problem and solution RBMs:

- Elements in "Group 1" are present in both the problem and the solution RBMs. They are used to identify the binding of the pattern problem, but are not modified in the pattern solution, although their interactions with other elements may change. Therefore a *modify* transformation action might be created for them, or they may remain unchanged. For example, if there are service invocations among the elements in this group (i.e., *CallOperationAction, AcceptCallAction,* and *ReplyAction*) and the solution specification indicates that their point of Call/Reply changes, then *modifyActionCall/ modifyReplyCall* transformation actions are used to redirect them.
- Elements in "Group 2" are present in the problem specification, but not in the solution specification, meaning that the SModel elements bound to them are to be removed from the SModel. A *delete* transformation action is created for each such element.
- Elements in "Group 3" are present in the solution specification but not in the problem specification, meaning that matching elements are required to be added into the SModel for them. An *add* transformation action is created for each element.

SModel elements that are not in any of the above groups remained untouched. Table 1 describes some of the most-used rules for transforming the SModel Structural Model, SEAM and Behavioral Model, BPM. The process proposed in this Chapter is supported by a tool for systematic recording of the SModel application rules which is described in Section 7.1.

Table 1: Example of SModel Refactoring Transformation Rules

Rules for refactoring the SModel Deployment Diagram (DP)	
addDeploymentArtifact	Takes a name as argument and creates an artifact in DP
deployArtifactToHost	Takes two arguments, the names of an existing artifact and a host node in DP and creates a deployment relationship between them.
addDeploymentHost	Takes a name as argument and creates an host in DP
Rules for refactoring the SModel Structural Model (SEAM)	
addParticipant	Takes one argument representing the name of the Participant to be added to the SModel SEAM.
deleteAssoc	Takes two arguments representing the names of participants whose association will be deleted.
addAssoc	Takes the names of two participants as arguments. A SEAM association is added between them.
Rules for refactoring the SModel Behavioral Model (BPM)	
addActivityPartition	Takes one argument representing the name of a new activity partition (swimlane) to be added to the SModel BPM
addActivity	Takes two arguments which are the name of a subgraph and the name of a swimlane, and creates a subgraph inside the swimlane. The details of creating the subgraph elements are described in a following block enclosed in '{}' that contains additional commands (e.g., addAction, addAcceptCallAction, addControlFlow, addDecision, addMerge, addFork, etc.)
addAction	Takes one argument, the name of the action which it creates.
addAcceptCallAction	Takes two arguments: the name of AcceptCallAction to be created.and a string, either Sync for a Synchronous call or

	Async for Asynchronous call. It creates the action denoted as *"ActionName(class-name::operation-name)"*.
addReplyAction	Takes the name of ReplyAction *"ActionName(class-name::operation-name)"* and creates it..
addCallOperationAction	Take two arguments: the name of CallOperationAction to be created and a string, either Sync for a Synchronous call or Async for Asynchronous call.. It creates the action denoted as *"ActionName(class-name::operation-name)"*.
addControlFlow	It takes two arguments, the names of two subgraph elements and creates a control flow link between them. The first is the origin and second the destination.
modifyActionCall	It takes two arguments: the first is the name of an existing CallOperationAction along with the name of the operation it is calling (e.g. CallOperationAction(Shopping::Checkout)) and the second is the new name. This command modifies the existing CallOperationAction to make a call to a new operation.
addActionCallReplyEdge	It takes three arguments: 1) name of a CallOperationAction; 2) name of AcceptCallOperation; 3) ReplyAction. It creates a call from (1) to (2) and also the reply from (3) to (1). If (2) and (3) are not from the same swimlane, it means that the call has been forwarded. In this case, an addActionCallEdge is created for each forwarding call in a following block enclosed in '{}'. (Note that the names for CallAction is shown by (ClassName::OperationName)).
addActionCallEdge	Takes two arguments: a CallOperationAction marked as Async and an AcceptCallOperation in another swimlane and it creates an asynchronous call between them.
deleteActionCallReplyEdge	Takes three arguments: 1) name of a CallOperationAction; 2) name of AcceptCallOperation; 3) ReplyAction. It removes the call from (1) to (2) and also the reply from (3) to (1).
Rules to Set MARTE Attributes of the SModel (They set the value if the attribute exists and also add the attribute if it does not exist)	
SetPaStepHostDemand	Takes the name of a UML action and the value for its «PaStep» *hostDemand*

SetPaStepHostProb	Takes the name of UML action and the value for its «PaStep» *prob*
SetPaRunInstancePoolSize	Takes the name of UML AD swimlane and the value for its «PaRunTInstance» *poolSize*
SetGaExecHostresMult	Takes the name of UML DD compute node and the value of its «GaExecHost» *resMult*

3.3.1 Recording the SModel Transformation Rules for the Façade Design Pattern

The SEAM for the case study system, Browsing and Shopping, is shown in Figure 12.B, with the corresponding BPN in Figure 3 and deployment diagram in Figure 5.

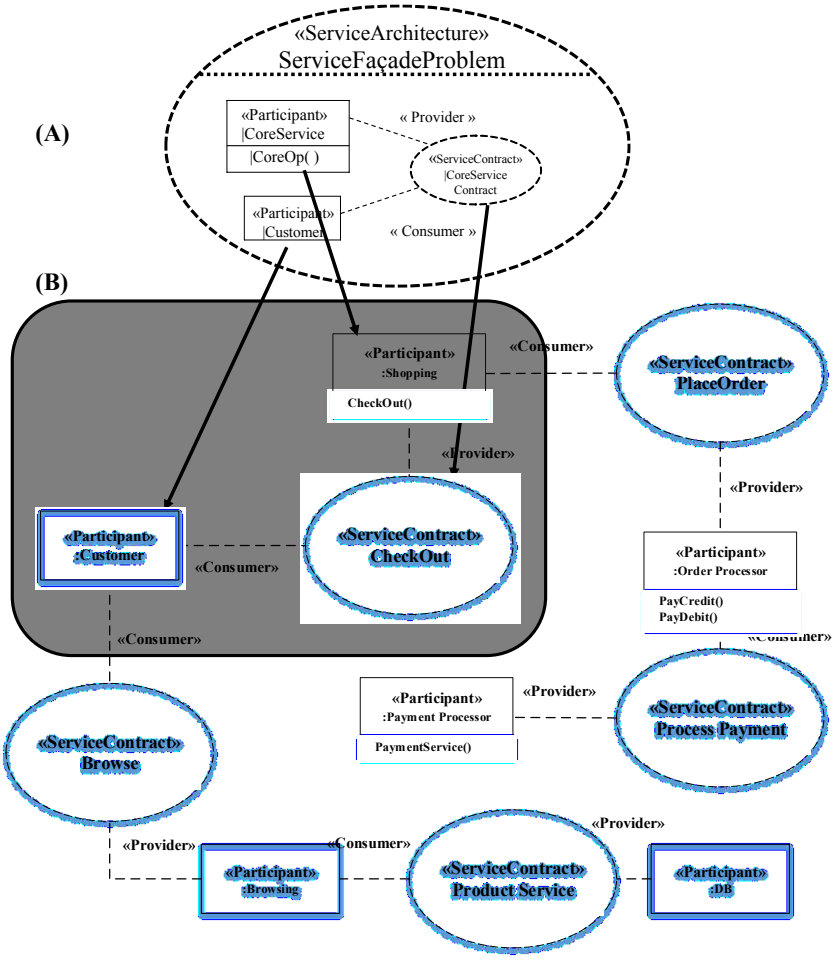

Figure 12: (A) Facade Pattern Problem Specification for SEAM; (B) SModel Service Architecture to which the pattern is applied, showing the problem identification

Figure 12.A shows the problem SPS of the Service Façade pattern. The subset of elements shaded in grey conforms to the SPS when Checkout (an operation of the Shopping Participant) is the core operation.

The technique for role binding discussed in Section 3.2 and the problem SPS and BPS shown in Figure 11.A and Figure 11.C help the system designer to identify the subset of SModel elements that are involved in the façade pattern application. Then the technique from Section 3.3 and the solution SPS and BPS shown in Figure 11.B and Figure 11.D are used by the system designer to record the SModel transformation rules.

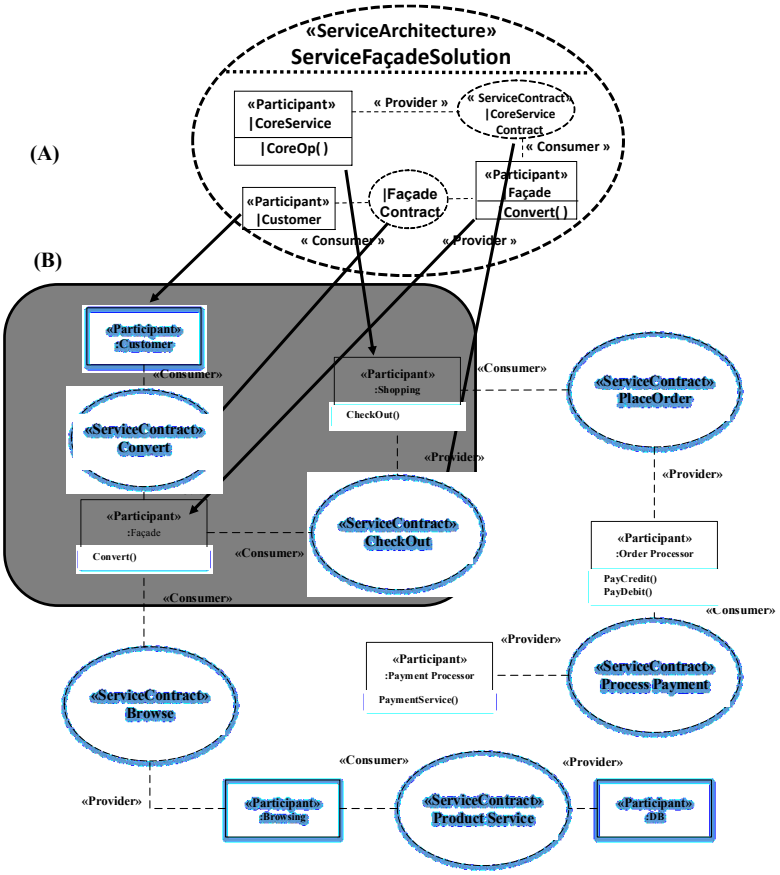

Figure 13: (A) Facade Pattern Solution Specification for SEAM; (B) SModel Service Architecture (SEAM) after the application of the pattern

Figure 13 shows the SModel SEAM after refactoring (Figure 13.B) and the part shaded in grey represents the binding with solution SPS (Figure 13.A). The role bindings are indicated by arrows.

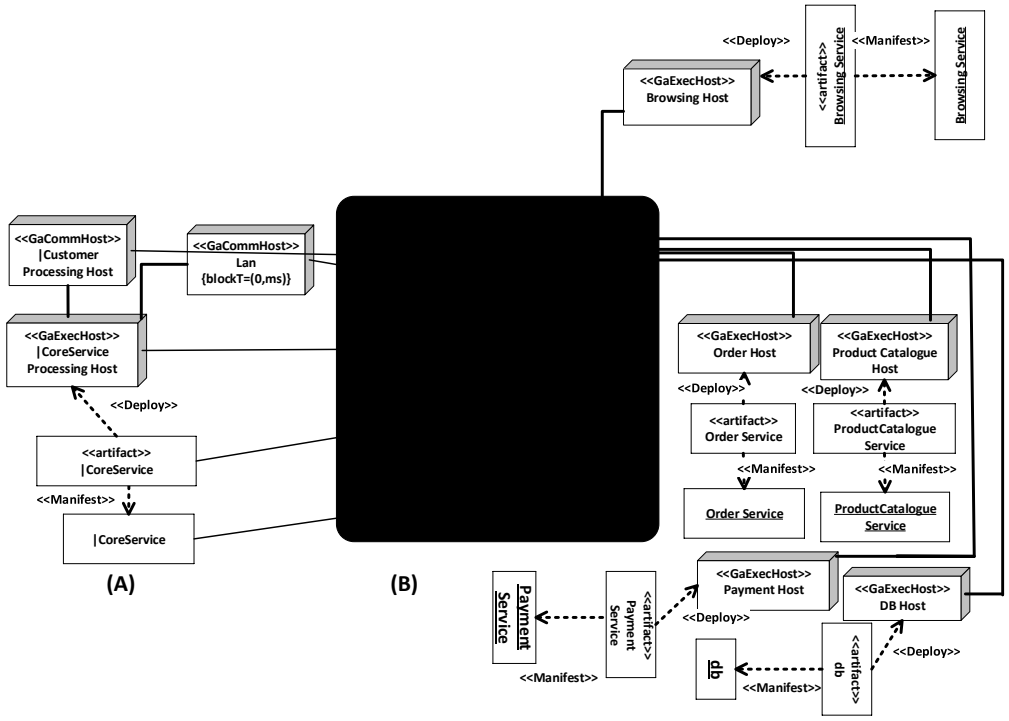

Figure 14: (A) Facade Pattern Problem Specification for Deployment Diagram; (B) SModel Deployment Diagram to which the pattern is applied, showing the problem identification

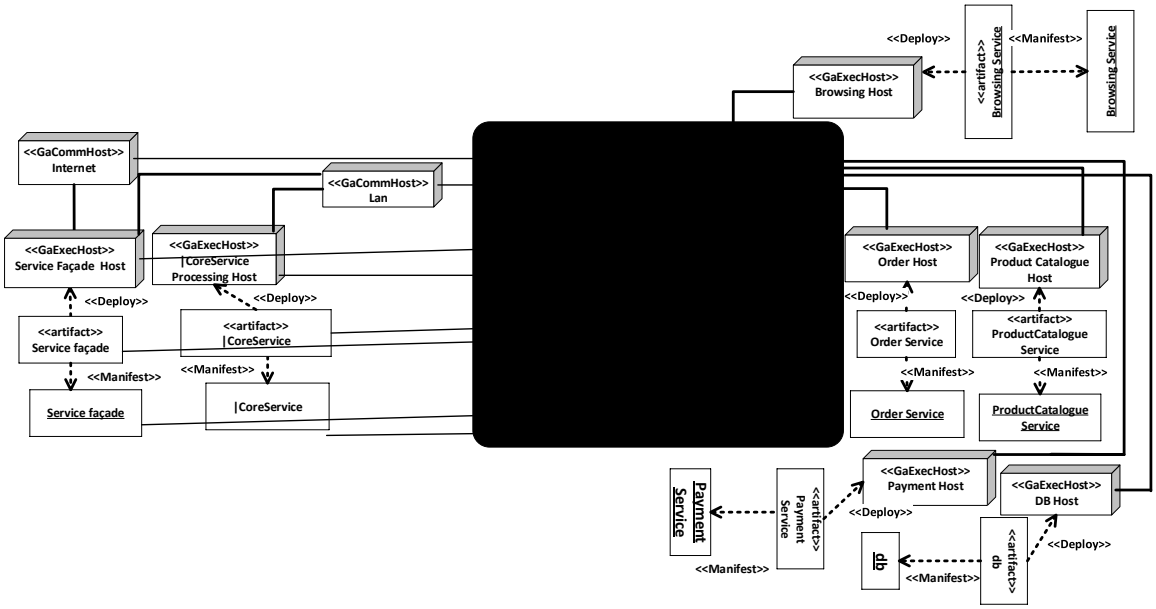

Figure 15: (A) Facade Pattern Solution Specification for Deployment Diagram; (B) SModel Deployment Diagram to which the pattern is applied

Similarly, Figure 14 shows the façade pattern problem specification of the SModel deployment diagram for the Browsing and Shopping SOA and Figure 15 shows the façade pattern solution specification and its conformance to the refactored SModel deployment diagram. The recorded SModel transformation rules for three SModel diagrams (SEAM, BPM and Deployment) are shown in Table 2. The SModel transformation rules in Table 2 are grouped based on the diagram they are applied to.

Table 2: SModel Refactoring Rules for the Façade Pattern

Rules to refactor Deployment Diagram (DP)
D1. addDeploymentArtifact(Façade) D2. deployArtifactToHost(Façade, Order) D3. addDeploymentManifest(façade) D4. manifestArtifact(Façade , façade)
Rules to refactor the SModel Structural Model (SEAM)
S1. addParticipant (Facade) S2. deleteAssoc (CoreServiceContract, CoreService) S3. addAssoc (CoreServiceContract, Façade) S4. addAssoc (Façade, CoreService)
Rules to refactor the SModel Behavioral Model (BPM)

B1. addActivityPartition (Façade)
B2. deleteActionCallReplyEdge (CallOperation(Shopping::Checkout), AcceptCall (Shopping::Checkout), ReplyAction(Shopping::CoreOp))
B4. addActivity(Façade, Convert1)
{
B4.1. addAcceptCallAction (AcceptCallAction (Façade::Convert), Sync);
B4.2. SetPaStepHostDemand (AcceptCallAction (Façade::Convert), '0.5ms')
B4.3. addAction (ConvertRequest);
B4.4. SetPaStepHostDemand (ConvertRequest, '2ms')
B4.5. addCallOperationAction (FaçadeCallOperationAction (Shopping::Checkout), Sync);
B4.6. SetPaStepHostProb(CallOperationAction (FaçadeCallOperationAction (Shopping::Checkout), '0.5')
B4.7. SetPaStepHostDemand (CallOperationAction (FaçadeCallOperationAction (Shopping::Checkout), '2ms')
B4.8. addAction (ConvertReply) ;
B4.9. SetPaStepHostDemand (ConvertReply, '2ms')
B4.10. addReplyAction (ReplyAction (Façade::Convert));
B4.11. SetPaStepHostDemand (ReplyAction, '0.5ms')
B4.12. addControlFlow(AcceptCallAction (Façade::Convert), ConvertRequest);
B4.13. addControlFlow(ConvertRequest, CallOperationAction (Shopping::Checkout));
B4.14. addControlFlow(CallOperationAction (Shopping::Checkout), ConvertReply);
B4.15. addControlFlow(ConvertReply, ReplyAction (Façade::Convert));
}
B6. modifyActionCall(CallOperationAction(Shopping::Checkout), CallOperationAction

```
(Façade::Convert))
B7.  SetPaStepHostProb(CallOperationAction (Façade::Convert) , '1')
B8.  addActionCallReplyEdge(CallOperationAction (Façade::Convert), AcceptCallAction
(Façade::Convert), ReplyAction (Façade::Convert))
B10. addActionCallReplyEdge (FacadeCallOperationAction (Shopping::Checkout),
AcceptCallAction (Shopping::Checkout), ReplyAction(Shopping::Checkout))
B11. SetPaStepHostProb(CallOperationAction (Façade::Convert) , '0.5')
```

Chapter 4 Mapping Table

In Chapter 3 , the process of recording the SModel refactoring transformation rules using the role binding was described. In this thesis, the SModel refactoring rules are translated into PModel refactoring rules through a coupled transformation technique (discussed in Chapter 5). “Coupled transformation” means that the SModel refactoring rules are used for the derivation of the PModel refactoring rules, based on the cross-model mapping between SModel and PModel; this mapping is an outcome of the transformation which generated the target PModel from the source SModel in the first place. This process requires a mapping table among SModel and PModel elements as an input, which is discussed in this chapter.

In this thesis, a technique based on traceability links between SModel and PModel elements is employed to propagate the SOA design pattern changes into the performance model of SOA (PModel) by translating them into the PModel domain. Traceability is a software design approach for establishing relationships between various software artifacts (across all kinds of models). It allows the software engineers to analyze the relationships and dependencies between artifacts; it also helps to analyze the impact of changes in different artifacts. PUMA4SOA [57, 106, 107] modeling framework, an extension of PUMA [3] that generates a LQN performance model from the UML design model of a SOA system is capable of creating these trace links. The procedure to create such links during the PUMA4SOA process is discussed in [58].

In this research, the traceability links correspond to the mappings between SModel elements and PModel elements, and are generated by the same transformation that derives the PModel from the SModel. The traceability links are stored in an external

model (represented here by the mapping table), separate from both the SModel and the PModel. Each type of traceability links produced by the SModel-to-PModel transformation describes the mapping relationship between an SModel element type (i.e., a metaclass of the UML metamodel) and a PModel element type (i.e., a metaclass of the LQN metamodel). However, this thesis extends the traceability links types to identify a set of SModel elements that are mapped to a PModel element. These extra types are not defined in the UML metamodel, but simplify the SModel-to-Pmodel mapping. The new added types are: a set of Actions (an Activity Subgraph) invoked by a Call, pairs of Call and Reply Actions, and Forwarding Calls (a set of synchronous and asynchronous calls). Figure 16 shows the overview of the PUMA process and the production of the mapping table. Section 4.1 discusses the structure of the mapping table and Section 4.2 shows the mapping table for the Browsing and Shopping case study.

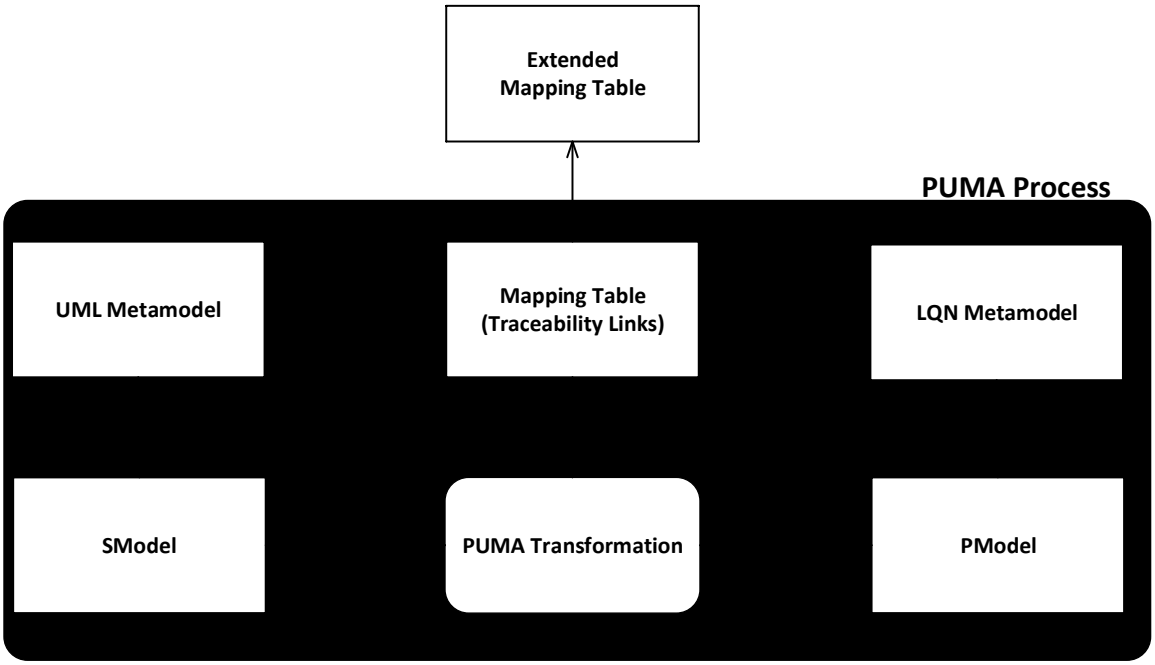

Figure 16: Mapping Table created during PUMA process

4.1 Structure of the Mapping Table

The mapping table has three sub-tables:

- StructuralElements (StrElement)
- Calls
- Attributes

Each link in the mapping table relates one or a set of SModel structural elements (SMEs), SModel element attributes (SMEAtts), or SModel element calls (SMECalls) to one or a set of corresponding PModel structural elements (PME), PModel element attributes (PMEAtt), or PModel element calls (PMECall). The extension to the mapping table is introduced to bridge a semantic gap between the SModel and the PModel, having to do with their level of detail. There are three extensions to the way the SModel is represented in the Mapping Table, using sets of SModel elements:

- An entry in the PModel corresponds to a set of activities in the SModel that are connected by hyperedges and form an Activity Subgraph. An SModel activity subgraph in the BPM models always starts with an *AcceptCallAction* following a call from a CallOperationAction in another swimlane, and either ends by providing a reply with a *ReplyAction* or forwards the call using an asynchronous *CallOperationAction*. The action corresponding to the elements of an activity subgraph is defined within '{}' starting with the action names, separated with commas. Then the activity control flows are each defined as a couple (indicating source and destination) within '()' and separated with commas. At end, all the hyper edges (Decisions, Joins, Forks, Merges) are each also defined within '()', separated with commas. If it is a Decision or Fork, the first element is the name of

the action that brings the follow to the Decision or Fork node and the rest are the name actions which have incoming flows from the Decision or Fork node. If it is Join or Merge, the last element is the name action which receives the incoming flow from the Join or Merge node and the rest are the name of actions which the Join or Merge node receive incoming flow from them. This set is mapped to a LQN Entry and all the elements inside the BPM activity subgraph are each mapped (one to one mapping) to the elements of the LQN activity subgraph inside the LQN entry. The activity subgraph inside a PModel entry is also defined in the same way that the SModel activity subgraph defined.

- A synchronous call (call-reply) in the PModel corresponds to a pair of messages (that is, of ActivityEdges that cross the boundary between two ActivityPartitions), called here a Synchronous Call. A pair of calls from the *CallOperationAction* in one BPM swimlane to an *AcceptCallOperation* in another swimlane and the reply call from *ReplyAction* to a *CallOperationAction*. *AcceptCallOperation* and *ReplyAction* are both from the same class (i.e. swimlane) in this set. Each call is defined as couple within '()' and both calls are specified within '{}' separated with commas. This set is mapped to LQN Synchronous Call.
- A forwarding call corresponds to a sequence of messages, called here a Forwarding Call. This is a set of both synchronous and asynchronous calls. A set of synchronous calls from the *CallOperationAction* in one BPM swimlane to an *AcceptCallOperation* in another swimlane and the reply call from *ReplyAction* to a *CallOperationAction*. *AcceptCallOperation* and *ReplyAction* are from different swimlanes in this set. The fact that the reply is coming from a swimlane which is

different than the swimlane of the receipt of initial call shows that the request has been forwarded to other swimlanes for processing. The request can be forwarded once or multiple times. In this set, each call is defined as couple within '( )' and all calls are specified within '{ }' separated with commas. The first two couples are the initial synchronous call and the reply call. The rest of the couples represent the forwarding calls. Because the set that represents a forwarding call in the SModel contains both synchronous and asynchronous calls, it is mapped to a PModel set contains a LQN synchronous call and one or more LQN forwarding calls.

With these extensions, a direct mapping can be made between elements of the extended SModel and the PModel. Each trace link in the mapping table is a triple: *Link = (link name, SME , PME).*

Table 3 shows all SModel and PModel structural and call elements types, and all the attribute types that are used in the thesis. The sets are labeled as '(Set)'.

Table 3: Types of SModel and PModel elements in the trace links of the Mapping Table

Sub-table (A) StrElements Trace Links	
SMEs	**PME**
Participant (in SEAM)	LQN Task
Host Node (in Deployment)	LQN Host
ActivityPartition/Swimlane (in BPM) stereotyped «PaRunTInstance»	LQN Task
Activity Subgraph (Set)	LQN Entry
BPM Action	LQN Activity
Sub-table (B) Calls Trace Links	
SMECall	**PMECall**
Synchronous Call (Set)	LQN Synchronous Call
Forwarding Call (Set)	LQN Forwarding Call (Set)
Asynchronous Call	LQN Asynchronous Call

Sub-table (C) Attributes Trace Links	
SMEAtts	**PMEAtt**
MARTE *extDelay*	Think Time of a workload
MARTE *resMult*	Processor Multiplicity
MARTE PaRunTInstance	Task Multiplicity
MARTE *hostDemand* attributes of an action	Host Demand attribute of the corresponding LQN Action
MARTE *prob* and *rep* attributes for a CallOperationAction (determining how many calls are made on average)	Number of Calls attribute for the corresponding LQN Call

4.2 Mapping table for Browsing and Shopping SOA and its Performance Model

Table 4 shows a few examples of traceability links from the case study Browsing and Shopping SOA, whose BPM, SEAM, and Deployment diagrams are given in Figure 3 to Figure 5. In Table 4, link BTL1 shows an example of an SModel activity subgraph mapped to an LQN entry and its activities. BCTL1 is an example of a set representing an SModel BPM synchronous call from the User to the Shopping swimlane and its mapping to an LQN call. Also, BATL1 and BATL2 shows the mapping for the MARTE attributes and the LQN model attributes for *hostDemand* and call *Prob.*

Table 4: Mapping Table between Shopping and Browsing SModel and PModel

Sub-table (A) StrElements Links		
Link	**Set of SMEs**	**PME**
DTL3	Deployment Node Order Host	LQN Host Order
DTL2	Deployment Artifact Browsing	LQN Task Browsing
STL3	SEAM Participant User	LQN Task User
STL2	SEAM Participant Browsing	LQN Task Browsing
BTL1	{AcceptCall (Shopping::Checkout), BasketCheckout, CalculateShipping, CreateInvoice, CallOperation(OrderProcessing::PayCredit), CallOperation(OrderProcessing::PayDebit), ReplyCallAction(Shopping::Checkout), (AcceptCall (Shopping::Checkout), BasketCheckout),(BasketCheckout, CalculateShipping), (CalculateShipping, CreateInvoice), ***Decision***(CreateInvoice, CallOperation(OrderProcessing::PayCredit) , CallOperation(OrderProcessing::PayDebit), ***Join*** (CallOperation(OrderProcessing::PayCredit) , CallOperation(OrderProcessing::PayDebit),	LQN Entry Checkout {AcceptCall (Shopping::Checkout), BasketCheckout, CalculateShipping, CreateInvoice, CallOperation(OrderProcessing::PayCredit), CallOperation(OrderProcessing::PayDebit), ReplyCallAction(Shopping::Checkout), (AcceptCall (Shopping::Checkout), BasketCheckout),(BasketCheckout, CalculateShipping), (CalculateShipping, CreateInvoice), ***Branch***(CreateInvoice, CallOperation(OrderProcessing::PayCredit) , CallOperation(OrderProcessing::PayDebit), ***Join*** (CallOperation(OrderProcessing::PayCredit) , CallOperation(OrderProcessing::PayDebit),

	ReplyCallAction(Shopping::Checkout)))}	ReplyCallAction(Shopping::Checkout)))}
Sub-table (B) Calls Links		
Link	**Set of SMEAtt**	**PMECall**
BCTL1	{(CallOperationAction(Shopping::Checkout) ,(AcceptCallAction (Shopping::Checkout)), ReplyAction(Shopping::Checkout), CallOperationAction(Shopping::Checkout))}	LQN synchronous Call (CallOperationAction(Shopping::Checkout),Checkout)
Sub-table (C) Attributes Links		
Link	**Set of SMEAtt**	**PMEAtt**
BATL1	MARTE Attribute *hostDemand* for the AcceptCallAction(Shopping::Checkout)	Host Demand attribute of LQN AcceptCallAction in Entry Checkout
BATL2	MARTE Attribute *prob* for the (CallOperationAction(Shopping::Checkout) AcceptCall (Shopping::Checkout))	Number of Calls attribute for the Call Edge from CallOperationAction(Shopping::Checkout) to Entry Checkout

Chapter 5 : Coupled Transformations

For propagating to the PModel the SModel changes due to the application of a pattern, the SModel refactoring transformation rules recorded by the system designer need to be translated into PModel transformation rules. In this Chapter, a translation approach is proposed based on the coupled transformation approach and the mapping table (discussed in Section 4.1). The coupling of the two refactoring transformations is possible because the performance PModel itself is related to the SModel by a SModel-to-PModel transformation. Figure 17 shows an overview of the transformation coupling proposed in this thesis.

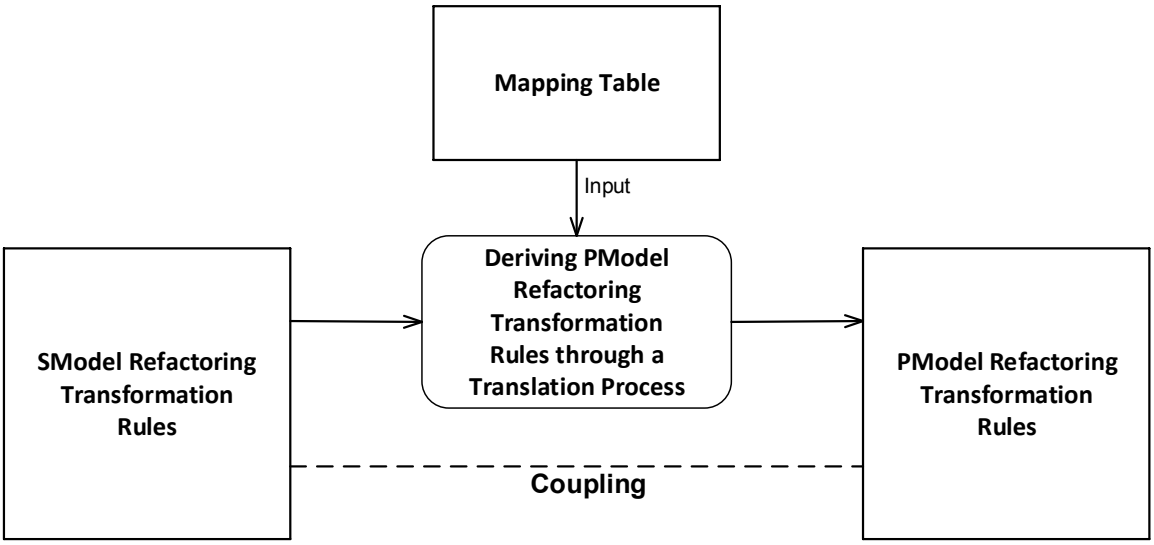

Figure 17: Coupled Refactoring Transformations

The general rules for the derivation of PModel refactoring transformation are discussed in Section 5.1, the derivation of the refactoring rules for Processors and Tasks in Section 5.2, for Entries and Calls in Section 5.3, and for Activity in Section 5.4. Finally, the derived PModel refactoring rules for the Browsing and Shopping case study are presented in Section 5.5.

5.1 General Rules for Coupled Transformation

Each SModel transformation rule is divided in two parts: the command (i.e. rule) name and the arguments. A rule is processed as follows:

1. The command name is translated into one or more PModel transformation rule command(s). The action part of the name (add/delete/modify) is retained, and the operand-type part (e.g. Participant) is mapped according to the type correspondences from the Mapping Table listed below and detailed in Table 3:

SModel Type	PModel Type
Activity Partition/Participant	LQN Task
Deployment Host	LQN Processor
BPM Activity (Subgraph)	LQN Entry
BPM Call/Reply pair (Sync)	LQN Call (Sync)
BPM Call (no replay)	LQN Call (Async)
BPM Action	LQN Activity
BPM Control Flow	LQN Sequence
BPM Decision, Join, Fork, Merge	LQN Branch, Join, Fork, Merge

2. The arguments of a derived PModel command are generated as follows:
    - If the SModel command creates a new element, then its name is used as the argument for the creation of the corresponding PModel element. If an existing SModel element is used as argument in a command, then its corresponding PModel element found from the mapping table is used as the corresponding argument in the PModel command.

- If the SModel command deletes or modifies an existing element, then the corresponding PModel element from the mapping table is used as the PModel command argument.

3. The derived PModel command(s) and arguments are combined as a PModel transformation rule.

A simplified pseudocode example for deriving PModel from SModel transformation rules for "addParticipant" and "modifyActionCall" is shown Figure 18. For example, the SModel "addParticipant" operation is mapped to "addTask" in the PModel, and the "addParticipant" argument becomes the new task name.

```
Foreach (smodelRule in smodelTransformationRules)
{           String SModelRuleCommand = ExtractCommand(SModelRule);
            List<String> smodelArtifacts = new List<String>();
            smodelArtifacts = ExtractRuleArguments(SModelRule);
            switch (SModelRuleCommand)
            { case "addParticipant":
                        pmodelRuleCommand = "addTask";
                        pmodelArtifacts.add(smodelArtifacts[0]); // task name
                        pmodelArtifacts.add(smodelArtifacts[1]); // processor name
                            // Combine command and Arguments and add them to PModel Transformation Rules
                        pmodelTransformationRules.add(pmodelRuleCommand,pmodelArtifacts);
                    break;
              case "modifyActivityCall":
                          List<string> results_modifyActivityCall = new List<string>();
                          if (MappingTableSearchByKey(smodelArtifacts[0]).Count != 0)
                         results_modifyActivityCall = MappingTableSearchByKey(smodelArtifacts[0]);
                      else
                         result_modifyActivityCall.Add("Error");
                      foreach (string result in result_modifyActivityCall)
                      {  pmodelRuleCommand = "modifyActivity";
                       pmodelArtifacts.add(result); // entry name
                       pmodelArtifacts.add(smodelArtifacts[0]); // old activity name
                       pmodelArtifacts.add(smodelArtifacts[1]); // new activity name
                              // Combine command and Arguments and add them to PModel Transformation Rules
                       pmodelTransformationRules.add(pmodelRuleCommand,pmodelArtifacts); }
                      break ; // Similarly the rest of SModel Transformation Rules are processed in Switch/Case
Statement  }
}
```

Figure 18: Simplified pseudocode for deriving PModel transformation rules from SModel transformation rules

Modifications to calls require special consideration in the translation. The SModel "modifyActionCall" operation changes a service invocation from a *CallOperationAction* to an *AcceptCallAction*. As this might apply to more than one call to the same *AcceptCallAction,* the mapping table is searched (by the MappingTableSearchByKey command) to identify all the PModel activities making the call. Then the operation is mapped to one or more "modifyActivity" operations in the PModel domain, to change all the calls.

To make sure that derived PModel commands are working with elements that exist at the time of refactoring, the SModel transformation rules are processed in the following order:

- All SModel transformations rules that add/modify structural elements/attributes
- All SModel transformations rules that delete elements calls
- All SModel transformations rules that delete structural elements/ attributes
- All SModel transformations rules that add/modify elements calls

In this way, for example, the target entry for a new call will be created before the call itself.

5.2 Deriving PModel Processors and Tasks

UML deployment diagram presents the physical deployment of artifacts and nodes in the SModel. The deployment diagram nodes (annotated by MARTE «GaExecHost») show the hardware components of the system and the artifacts («SchedulableResource») show the software components. The behaviour of artifacts is modeled in the BPM diagram swimlanes, which are mapped to PModel tasks. Therefore the deployment diagram artifacts («SchedulableResource») which are deployed to physical nodes («GaExecHost») indicate how the PModel tasks are allocated to PModel processors.

The SModel rule *addDeploymentArtifact* which creates an artifact in the deployment diagram, is translated to PModel rule *addTask* which creates a task in the PModel. The SModel rule *deployArtifactToHost* which deploys a «SchedulableResource» to a «GaExecHost», is translated to the PModel rule addToProcessor which deploys a task to a processor. Similarly, other changes (e.g. modify, delete) applied to a deployment diagram artifact or host are translated to changes to PModel tasks and processors.

There is a potential for duplicating the creation of a Task, which may be created because of a new SEAM Participant, a new BPM swimlane, or a new Deployment artifact. There is a constraint between these SModel elements (i.e., for every SEAM Participant there is a BPM swimlane and a Deployment Artifact in the SModel). All three are mapped to the same element (i.e., LQN Task), so only one of them should be translated into a new Task in the PModel.

5.3 Deriving PModel Entries, Phases, Nested Calls, and Forwarding Calls

As described in Chapter 4 a PModel Entry contains a set of PModel Activities and precedence relationships that together form a PModel subgraph, which is mapped to an SModel subgraph composed of a set of Actions and edges. The structure of the SModel and PModel subgraphs is as follows. The SModel subgraph always starts with an *AcceptCallAction* following a call from a *CallOperationAction* in another swimlane. At some point in the graph we suppose there is a *ReplyAction* replying to the call, or an asynchronous *CallOperationAction* which forwards the call. The corresponding PModel subgraph starts with a *firstActivity* to begin the execution. For the *ReplyAction* or the asynchronous *CallOperationAction*, the corresponding Activity is designated as a *replyFwdActivity* which (in conformance with the SModel behaviour) either sends a reply

to the caller, or forwards the request to another entry, respectively. Therefore the PModel *firstActivity* is mapped to the SModel *AcceptCallAction* and the PModel *replyFwdActivity* is mapped to the SModel *ReplyAction* if it replies and to the asynchronous *CallOperationAction* if it forwards the call. In the PModel, the entry operation groups the activities up to the *replyFwdActivity* into its so-called “Phase 1” and the activities after the *ReplyFwdActivity* (if any) into its "Phase 2”.

The following interpretations are applied to the SModel behavior specification:

- *CallOperationActions* are capable of making either synchronous or asynchronous calls. The call type is set as parameter.
- Respectively, *AcceptCallOperations* are also capable of accepting either Synchronous or Asynchronous calls.
- Replies to the original caller (i.e. *CallOperationAction*) are provided by *ReplyAction*. The caller information is passed to the *ReplyAction* from the *AcceptCallOperation*.
- If the *AcceptCallOperation* received the caller information through an asynchronous call from a *CallOperationAction*, then this shows that the subgraph received a forwarding call. This means that the *CallOperationAction* will not wait for a reply and it only provides the information for the call initiator. The subgraph can forward the request again using an asynchronous *CallOperationAction* or provide a reply to the call initiator using *ReplyAction.*

5.3.1 Translation of Operations for Activities and Entries

In Chapter 3 , addActivity is introduced as an SModel transformation rule that takes the name of SModel swimlane as an argument and adds an empty activity subgraph to it. The

elements of the subgraph must also be added with the help of additional commands given in a block defined between "{}" which follows the addActivity command. Depending on the element types, these commands are: *addAction*, *addAcceptCallAction, AddCall-OperationAction, AddReplyAction, addActivityEdge*, *addDecision, addMerge, addFork, addJoin* .

Each addActivity (and its associated subgraph) in the SModel domain creates an addEntry in the PModel domain. The activity subgraph is also mapped to an LQN activity subgraph contained by the LQN entry. For deriving the PModel transformation rules the first step is to analyze the activity subgraph block defined within "{}". Each addActivity and its associated activity subgraph block is processed as follows:

- The part of subgraph starting at *AcceptCallAction* and ending at *ReplyAction* (triggered by an operation invocation *CallOperationAction* from another swimlane) is called "first phase sub-diagram" and is mapped to a first phase PModel entry. The part of subgraph that comes after the *ReplyAction* is called "second phase sub-diagram" and is mapped to the second phase of the same entry. Figure 19 shows an example of a synchronous operation invocation in SModel (Figure 19.A) and its mapped scenario in the PModel (Figure 19.B). Action A and B along with *AcceptCallAction* and *ReplyAction* are transformed into first phase PModel activities inside Entry A. Actions C and D are transformed into second phase PModel activities inside the Entry A.

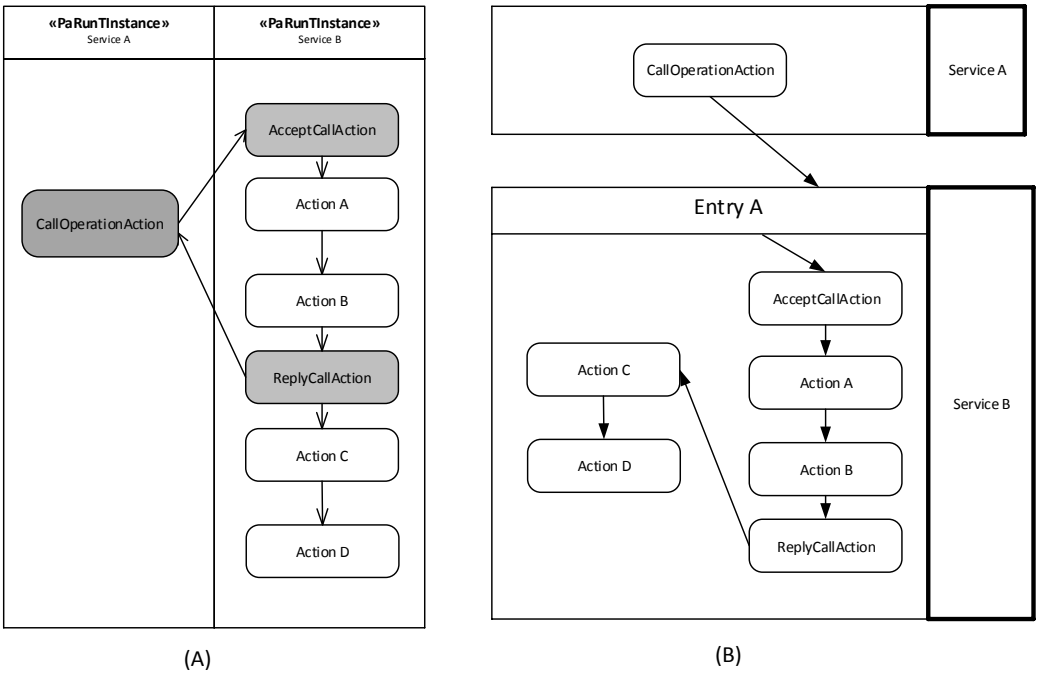

Figure 19: (A) A Synchronous call operation invocation in SModel (B) Transformation in PModel

In the SModel commands, if the *addAcceptCallAction* is taking a synchronous call from a *CallOperationAction* in another swimlane and if there is *addReply-Action*, then all the *addAction*, *addAcceptCallAction, addCallOperationAction,* and *addReplyAction* actions are mapped to *addActivity* actions in PModel domain which create activities inside the first phase of the entry.

- If there is a *CallOperationAction* among the actions in the sub-diagram within the *AcceptCallAction* and *ReplyAction* that makes a synchronous operation call to a sub-diagram in another swimlane, then a PModel synchronous call is created from the corresponding activity of the entry mapped to the first sub-diagram to the entry mapped to the second sub-diagram. Figure 20 shows a nested synchronous call scenario in SModel (Figure 20.A) and its mapping to PModel

(Figure 20.B). A PModel Call is created from the CallOperationAction Activity inside the first phase of Entry A to Entry B in the other task.

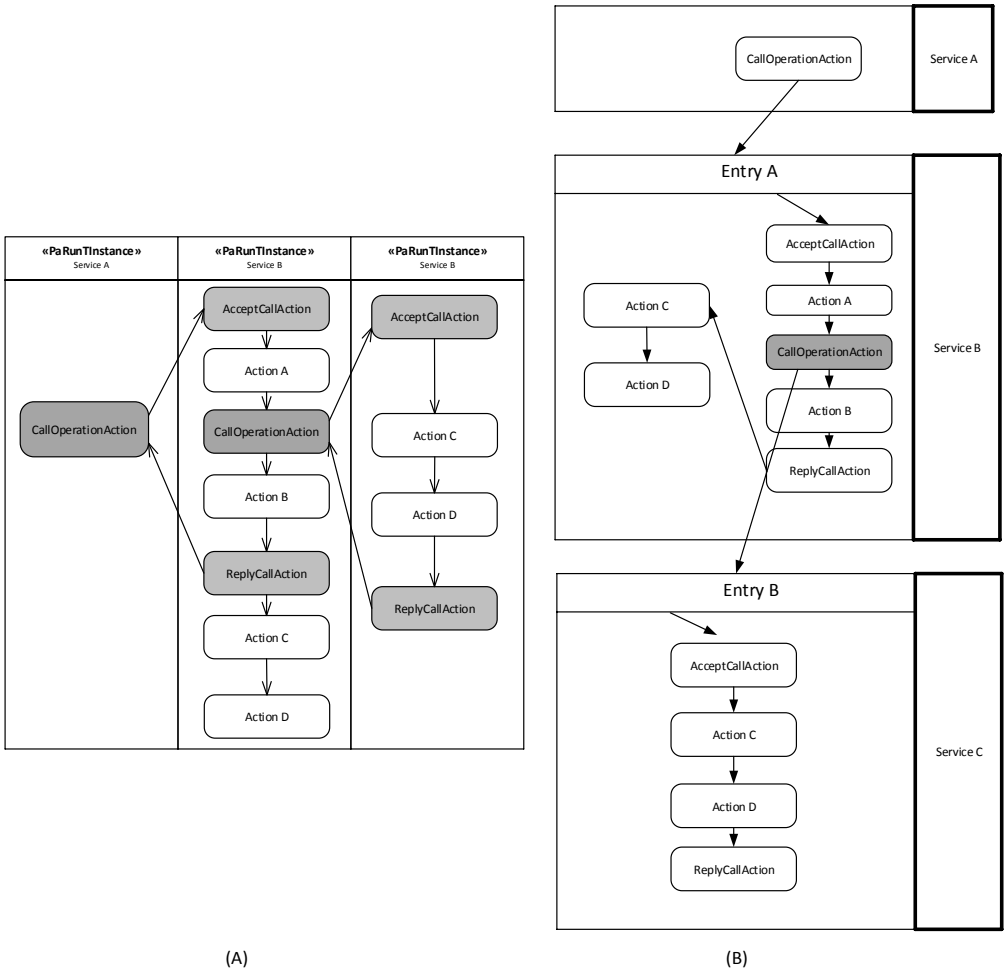

Figure 20: (A) Nested Synchronous call in SModel (B) Transformation in PModel

In the SModel commands, there will be *addCallOperationAction* within the commands for a sub activity diagram which makes a synchronous call to another sub activity diagram using *addActionCallReplyEdge*. This is mapped to an *addCall* in the PModel domain, which makes the nested call in the PModel.

- If the sub-diagram starting with an *AcceptCallAction* does not contain any *ReplyAction* but contains a *CallOperationAction* instead which makes an asynchronous call to another sub-diagram, then this is a case of forwarding call. The sub-sub-diagram from *AcceptCallAction* to *CallOperationAction* is mapped to the phase 1 of the respective entry, and the rest of the sub-diagram is mapped to the second phase of the same entry. Also a PModel forwarding call is created from the PModel activity corresponding to *CallOperationAction* to the entry mapped to the target sub-diagram. Figure 21.A shows the SModel scenario for the call forwarding and Figure 21.B shows the mapped scenario in PModel. Action A along with *AcceptCallAction* and *CallOperationAction* are transformed into activities inside first phase of Entry A. A PModel forwarding call is created from *CallOperationAction* to Entry B in Service C.

There are other call forwarding aspects that we should consider. For instance, let us assume that in the SModel commands there is an *addAcceptCallOperation* which creates an *AcceptCallOperation* receiving a synchronous call, but without any *addReplyAction* to create a reply to the caller. Also there is an *addCallOperationAction* within the commands for a sub activity diagram which makes an asynchronous call to another sub activity diagram. Then there is an *addActionReplyCallEdge* whose two arguments, *AcceptCallOperation* and *ReplyAction,* are from different swimlanes, indicating that *CallOperationAction* makes a call to a swimlane and gets the reply from a different one. Because of that, one or more *addActionCallEdge* rules are defined in {} for *addActionReplyCallEdge,* each representing a forwarding Call. This is translated to an *addFwdCall* in PModel domain, creating a forwarding call from the

CallOperationAction that makes the asynchronous call to the entry providing the reply to the original caller. This is mapped to an *addCall* in the PModel domain which makes the nested call in the PModel.

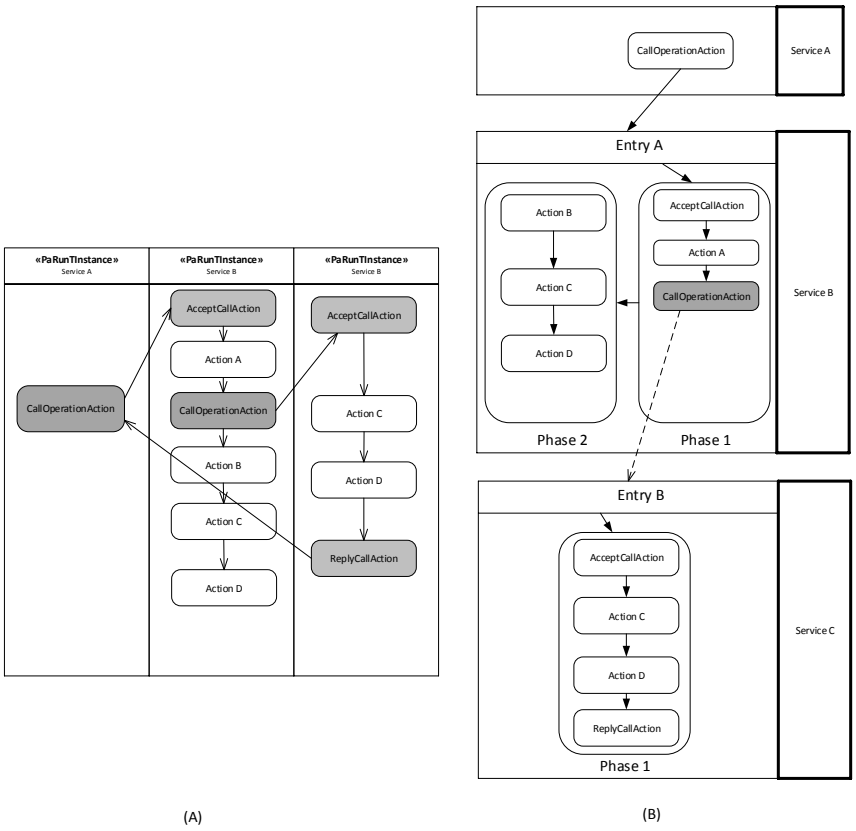

Figure 21: (A) Call Forwarding Scenario in SModel (B) PModel Transformation

5.4 Deriving PModel Activity

In Section 5.3, the SModel activity subgraph and the mappings to the PModel are discussed. In this section, the detailed mapping of the activity subgraph is explained. The activities in the PModel are a one-to-one image of the Actions in the corresponding SModel Activity subgraph. Therefore wherever a SModel element is added or deleted, the corresponding PModel element must be added or deleted in the PModel transformation.

In SModel, an activity diagram is described by actions and precedence relationships which are control flows, decisions, merges, forks and joins. In PModel, the detailed execution of an entry is shown by activities which are connected by sequences, branches, merges, forks and joins. Therefore, during the analysis of a SModel sub activity diagram, SModel rule addAction is mapped to PModel rule addActivity. Similarly, addDecision to addBranch ,addMerge to addMerge, addForks to addForks, addJoin to addJoin. Once all the elements are added to the activity diagram, they will be connected by addControlFlow in SModel and by addSequence in PModel.

5.5 Deriving PModel Transformation Rules for the Façade Design Pattern

The PModel transformation rules for adding a façade design pattern derived from the SModel refactoring transformation rules from Table 4 is shown in Table 5.

The SModel "*addDeploymentArtifact*" is mapped to "*addTask*" in the PModel domain. Because the command is creating a new PModel task, the SModel argument is used as the new task name. *addAcceptCallAction (AcceptCallAction (Façade::Convert), Sync)* with a *addReplyAction (ReplyAction (Façade::Convert))* indicates that the subgraph receives synchronous calls. Also *addCallOperationAction (FaçadeCallOperationAction (Shopping::Checkout), Sync)* indicates that the subgraph makes a nested synchronous call. *addActionCallReplyEdge* creates this nested synchronous call. In the PModel domain, *addEntry*, *addActivity*, and *addSequence* take care of creating the PModel activity subgraph for the PModel entry.

The SModel "*modifyActionCall*" changes a service invocation call made by a *CallOperationAction* to an *AcceptCallAction*. The command is interpreted as "*modifyActivity*" in the PModel domain. As there might be more than one invocation call

for an *AcceptCallAction,* the mapping table is used to identify all the PModel entries containing this action, using them along with the old and new call names as arguments for the derived PModel command. *addActionCallReplyEdge* are mapped to *addCall* in PModel to create the synchronous calls in the transformed PModel. Similarly, other SModel transformation rules are also mapped to the PModel transformation rules.

Table 5 : Derived PModel Transformation Rules from SModel Transformation Rules for Façade Design pattern application to Browsing and Shopping SOA

PModel Rules
LQN
L1. addTask (Facade) L2. addToProcessor (Façade, Order) L3 deleteCall(CallOperation(Shopping::Checkout),Shopping) L4 addEntry(Convert, Façade) L5 addActivity(AcceptCallAction(Shopping::Checkout),Convert) L6 addHostDemand(AcceptCallAction(Shopping::Checkout),'0.5ms') L6 addActivity(ConvertRequest,Convert) L7 addHostDemand(ConvertRequest,'2ms') L8 addActivity(FaçadeCallOperation(Shopping::Checkout),Convert) L9 addHostDemand(FaçadeCallOperation(Shopping::Checkout),'2ms') L10 addActivity(ConvertReply,Convert) L11 addHostDemand(ConvertReply,'2ms') L12 addActivity(ReplyAction(Shopping::Checkout),Convert) L13 addHostDemand(ReplyAction(Shopping::Checkout),'0.5ms') L14 addSequence(AcceptCallAction(Shopping::Checkout),ConvertRequest,Convert) L15 addSequence(ConvertRequst,CallOperationAction,Convert) L16 addSequence(FaçadeCallOperation(Shopping::Checkout),ConvertReply,Convert) L17 addSequence(ConvertReply,ReplyAction(Shopping::Checkout),Convert) L18 modifyActivity(CallAction(Shopping::Checkout),CallOperationAction(Façade :: Convert)) L19 addCall(CallOperationAction(Façade :: Convert),Convert,'1') L20 addCall(FacadeCallOperationAction(Shopping::Checkout),Shopping, '0.5')

Chapter 6 Refactoring the PModel

This chapter describes the automated process of refactoring the LQN Performance Model (PModel) with the derived PModel transformation refactoring rules using Query/View/Transformation Operational (QVT-O).

Standard model transformation languages such as QVT [12], are able to provide model refactoring for any MOF (Model Object Facility) compliant model. QVT Operational uses an imperative approach and provides a procedural concrete syntax for defining the transformation rules. But QVT Operational requires hard-coded transformation rules, so pattern-specific transformations would have to be coded for every pattern. Since our goal was to build a generic transformation approach that would work for many different patterns and all types of PModel changes, the set of refactoring rules needs to be communicated at run-time to a universal QVT-O based transformation engine for PModel refactoring. A few options for implementing such an approach were considered:

- Processing PModel refactoring transformation rules provided as a list to a generic QVT engine, which takes an LQN PModel as input. However, the QVT language has limited capabilities for processing lists and reading external files. Therefore this option could not be implemented.
- Formatting the PModel transformation directives into metadata annotations composed of key-value pairs (similar to the approach adopted by the User Requirement Notation (URN) in [108]). The transformation directives in the form of metadata would be used to guide the QVT engine. However, the concept of metadata is not supported by the exiting LQN metamodel (which has no extension mechanisms). This solution

would require extending the LQN metamodel with metadata. The disadvantage is that such a change would also require modifications in the tool support.

Therefore we adopted a different solution in the thesis: extending the LQN metamodel with transformation directives (TD) attached to the affected model elements only for the refactoring phase. A regular LQN model is transformed into this new LQN+TD with annotation capabilities using QVT, where the transformation directives are added to the model according to the PModel refactoring rules derived in Chapter 5. At the end of the refactoring process a regular PModel is generated, which can be solved with the existing tool support.

This approach allows for customized annotation for every LQN PModel element type, which makes the processing easier for the generic QVT engine, as different kinds of changes are attached to different PModel element. Also, in the case when a large set of the changes are required for an element (such as adding a subgraph), multiple annotations can be composed. Therefore, in this thesis the PModel refactoring is performed in three main steps:

1. The PModel is transformed to a PModel with annotation capabilities conform to the extended LQN+TD metamodel.
2. The PModel refactoring transformation rules obtained by the derivation process discussed in Chapter 5 are used to annotate the PModel with transformation directives.
3. A QVT-based transformation engine (designed and developed in the thesis) will traverse the annotated PModel and refactor it. Figure 22 shows the overview of the proposed process described in this chapter of thesis.

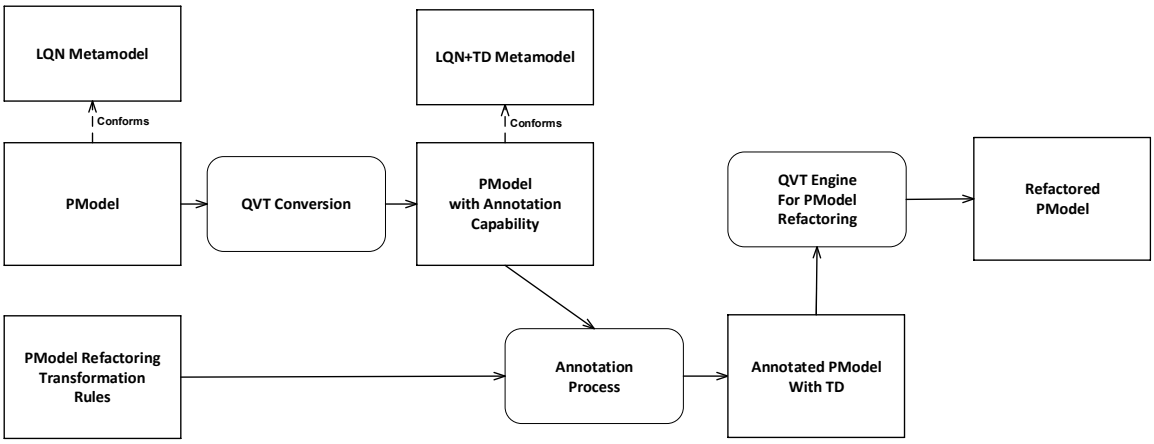

Figure 22: Refactoring PModel using TD Annotations

In Section 6.1, the Transformation Directives (TD) and the LQN+TD metamodel are discussed and in Section 6.2 the algorithm behind the QVT transformation engine is described.

6.1 Annotating the PModel using Transformation Directives

In order to add the transformation rules as annotation to the performance model they need to be formatted into special PModel elements which are attached to the existing PModel elements. This thesis introduces the new PModel elements as Transformation Directives (TDs). Each PModel element owns zero or more TDs, each describing a change about that model element. The PModel is annotated with TDs using the PModel refactoring transformation rules discussed in Chapter 5 .

The regular LQN metamodel [3] does not have any extension mechanisms (such as stereotypes or metadata), therefore this thesis defines an extended metamodel. Figure 23 shows that this new metamodel combines the regular LQN metamodel (shown on the right side of the figure) with a set of TD metaclasses (shown in the grey area on the left).

Standard model transformation techniques such as QVT can easily transform a LQN conform to the existing LQN metamodel into an extended model conform to the proposed LQN+TD metamodel. Once the PModel has been annotated and then refactored by the transformation engine, it can be transformed back into a regular LQN model.

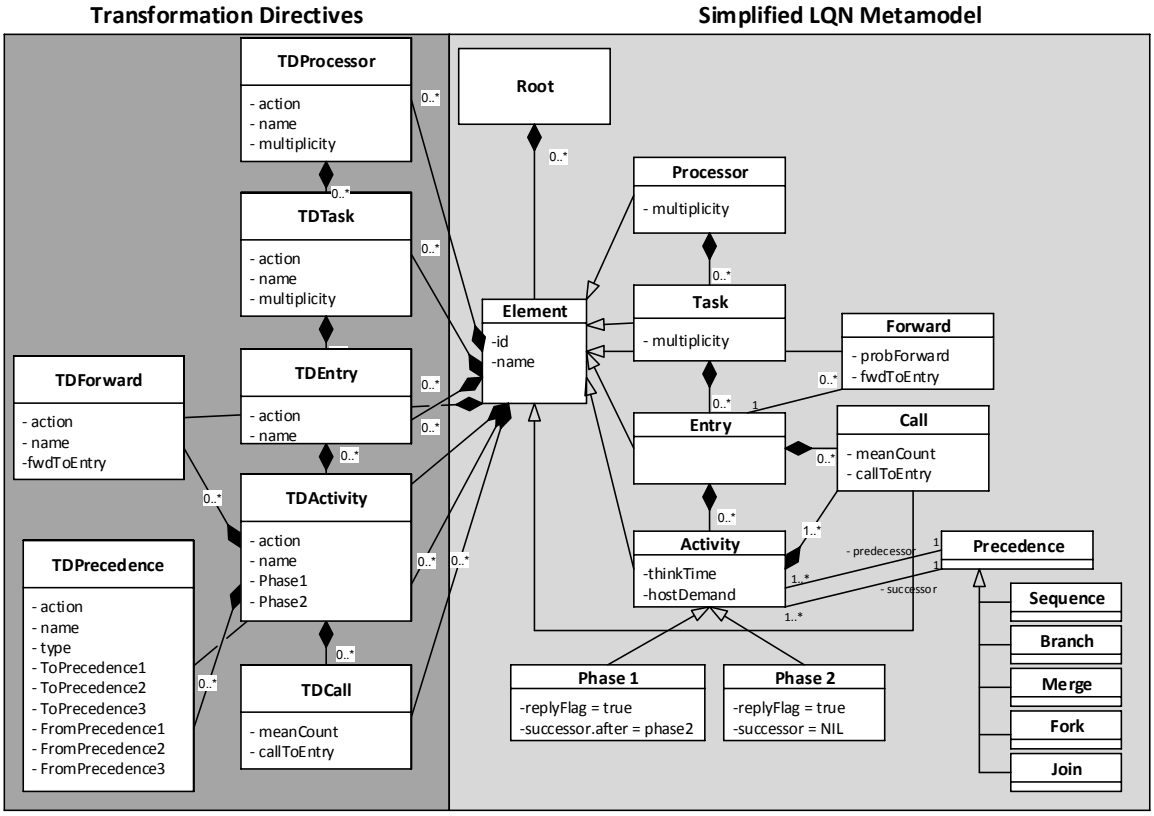

Figure 23 : Extended LQN + Transformation Directives Metamodel

The metamodel in Figure 23 shows a composition relationships between the "Element" metaclass and each defined TD metaclass. This means that every PModel element instance can own zero or more instances of any TD type. Every PModel type has a corresponding TD type (e.g. TDProcessor corresponds to Processor). Every TD metaclass defines an attribute named "action". If a PModel element has some more attributes that might be changed by a PModel transformation rule, then the corresponding TD also has an attribute with the same name. For example, the PModel processor has an attribute

called “multiplicity”. Therefore the TDProcessor also has an attribute named “multiplicity”. Each PModel transformation rule annotates the PModel as folows:

- If the PModel transformation rule asks to remove a PModel element, then a TD with the same type as the PModel element under change is created and attached to it. The “action” attribute of the TD is set to “Remove”. For example, if a task is being removed in the PModel, a TDTask is created for the task with the “Action” attribute set to “Remove”.
- If the PModel transformation rule modifies the attributes of an existing PModel element, then a TD with the same type as the PModel element under change is created and attached to it. The “action” attribute of that TD is set to “Modify” and the attribute being modified contains the new value. For example, if the “Multiplicity” attribute of a processor is modified, then a TDProcessor is created for that processor with “action” attribute set to “Modify” and the “multiplicity” attribute holding the new multiplicity value.
- If the PModel transformation rule adds a new element to an existing PModel element, then a TD with the same type as the new element is created. The TD here acts as a placeholder for the new element. In this case, the “action” attribute of the TD is set to “Add” and the other attributes hold the respective new values. For example if a new task named “Façade” is added to an existing PModel processor called “Order Processor”, then a TDTask with the attribute “action” set to “Add” and attribute “name” set to “Façade” is created for “Order Processor”.

In Figure 23, there are also composition relationships defined among various types of TDs (e.g. TDProcessor, TDTask, TDEntry, etc.) in the same structural order that they are

defined for regular LQN elements (e.g. Processor, Task, Entry, etc.). This helps in defining PModel subsystems to be added using the TDs. For example, to add a task with several entries to a PModel processor, a TDTask which owns a TDEntry for each PModel entry is created. The TDTask and all its TDEnteries are acting as a place holder for the added elements.

An example of an annotated PModel with TDs is shown in Figure 24. TD elements are shown using a shape similar to UML note shape. A new processor (TDProcessor) is defined for the model named “Façade” and a new task (named Service Façade) is created for it using TDTask. Then, three new entries Convert1, Convert 2, and Convert 3 are defined for the three “convert” channels using TDEntry. Two LQN calls are created (using TDCall) for each entry, to the existing entries “Browse” and “Checkout”. Also three user processors (TDProcessor), tasks (TDTask), and entries (TDEntry) and three calls to each convert channel (i.e. Convert1, Convert 2, and Convert 3) are added. Two existing calls from user to Browse and Checkout are annotated to be deleted.

Although the technique in this thesis allows for the annotation of the activities inside each entry, this is not illustrated in Figure 24 to avoid crowding the figure. Instead we have taken advantage of the fact that the PModel specification allows for PModel presentation and analysis (i.e solving using LQN solver) based on a calculated hostDemand for each entry.

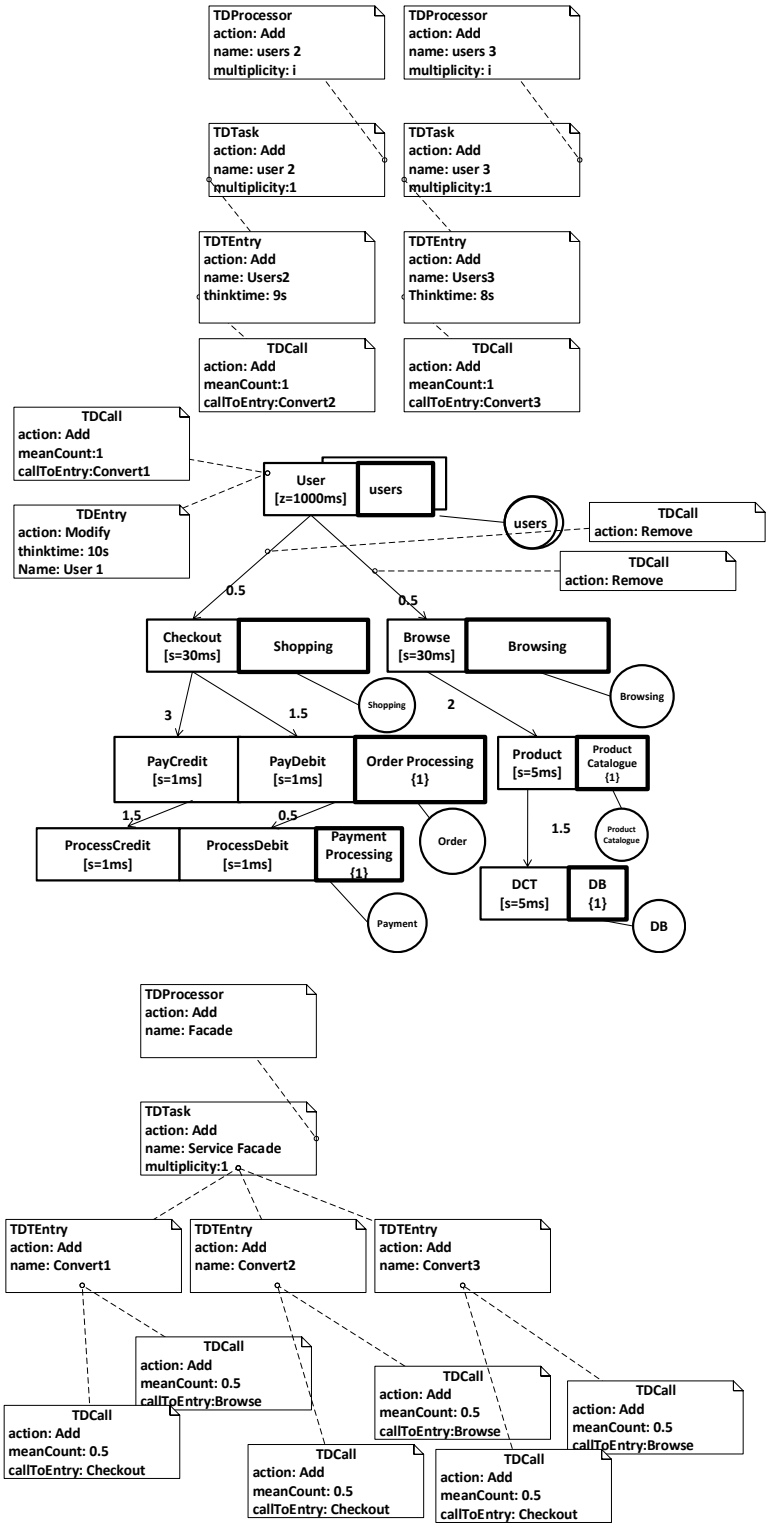

Figure 24: Browsing and Shopping PModel annotated with TDs

6.2 Refactoring PModel using QVT

Once all the propagated changes are annotated, the PModel is passed to the refactoring process. In this thesis, the transformation engine is designed and developed using QVT-O transformation language. The transformation engine reads the annotated PModel as the source model and traverses all its nodes, reads the TDs, and generates an output model with the changes applied. TDs in the PModel transformation are processed in the following order: 1) TDProcessors; 2) TDTask; 3) TDEntry; 4) TDActivity; 5) TDCall; and 6) TDForward.

Starting from the PModel element type "Processor", the QVT engine processes each PModel Processor element as follows:

- If the processor from the source model is not annotated by TDProcessor to be removed, an identical processor is created with a possible subgraph (i.e. Tasks, Entries, Activities, etc. attached to the processor) in the output model. If the processor is annotated as to be removed, then proceed to the next processor.
- If the processor is annotated by TDProcessor indicating the processor to be modified, then modify the processor using the information provided in the TD attributes.
- If the processor is annotated by TDTasks indicating new tasks to be added to the processor, then create a task for each TDTask using the information it provides. If the TDTask has a subgraph attached to it (e.g. TDTask can have TDEntries with TDActivities attached to it), it will be transferred to the newly created tasks.
    - If the newly created task, has TDEntries indicating new entries to be added to the task, then create an entry for each TDEntry. If a TDEntry has a subgraph attached to it, it will be transferred to the newly created entry.

- If the newly created entry has TDActivities indicating new activities to be added to the entry, create an activity for each TDActivity. If any TDActivity has a subgraph, it will be transferred to the newly created activities.
    - If the newly created activity has TDPrecedences indicating new precedencies to be added to the newly created, then create the precedence based on the its type for the activity.
    - If the newly created activity has TDCalls indicating new calls to be made by the activity, they are left for later processing.

The above process is repeated for the rest of PModel artifacts in the following order: Tasks, Entries, Activities and finally Calls. The annotated model shown in Figure 24 is given to the QVT transformation engine discussed in this section and the result is shown in Figure 25.

Although the technique in this thesis allows for the annotation of activities inside each entry, these are not illustrated in Figure 25 to avoid crowding the presentation of the PModel. Instead the Figure shows a single total hostDemand for each entry, calculated as the weighted sum of the activity demands (weighted by the number of times the activity is executed, per invocation of the entry). This is a permitted option in the PModel, using a single aggregated activity to represent all the activities for the entry, provided there are no forks and joins in the activity subgraph. It is also possible to aggregate the phase 1 and phase 2 activities of the entry separately and define a hostDemand for each phase.

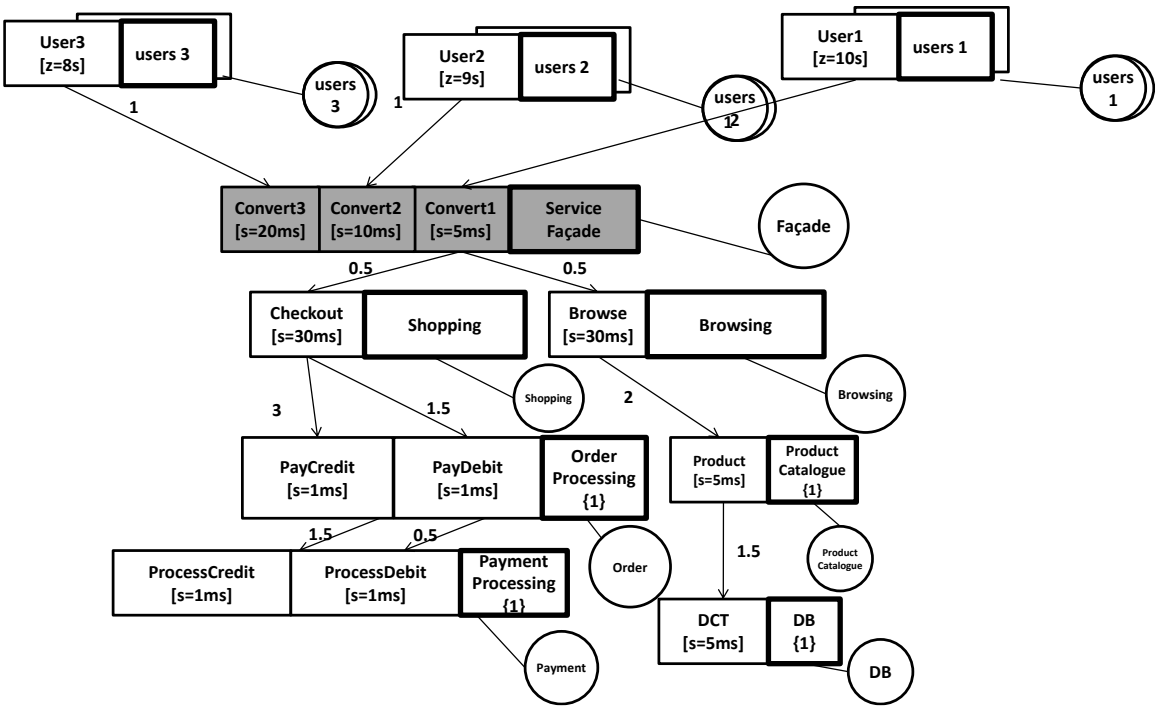

Figure 25: Refactored PModel (after applying the multi-channel variant of the façade pattern)

Chapter 7 Tool Support

In this Chapter, the tools which have been implemented to support the coupled transformation technique developed in this thesis are described. There are four tools:

- SModel transformation rules recording tool (Technique discussed in Chapter 3 the tool is described Section 7.1)
- PModel transformation rule derivation tool (Technique discussed in Chapter 5 and the tool is described in Section 7.2)
- PModel annotation tool (Technique discussed in Section 6.1 and the tool is described in Section 7.3)
- QVT based transformation engine (Technique discussed in Section 6.2 and tool is described in Section 7.4)

7.1 SModel Transformation Rules Coding Tool

Although the process of recording SModel transformation rules is not automated in this thesis, the complete procedure is systematically defined. Therefore, a tool was implemented based on the systematic approach to help the system designer to record the SOA design changes. The tool provides the recorded SModel transformation rules in a format that can be processed for the derivation of PModel transformation in another tool. Figure 26 shows how the system designer can use the tool to record the SModel changes.

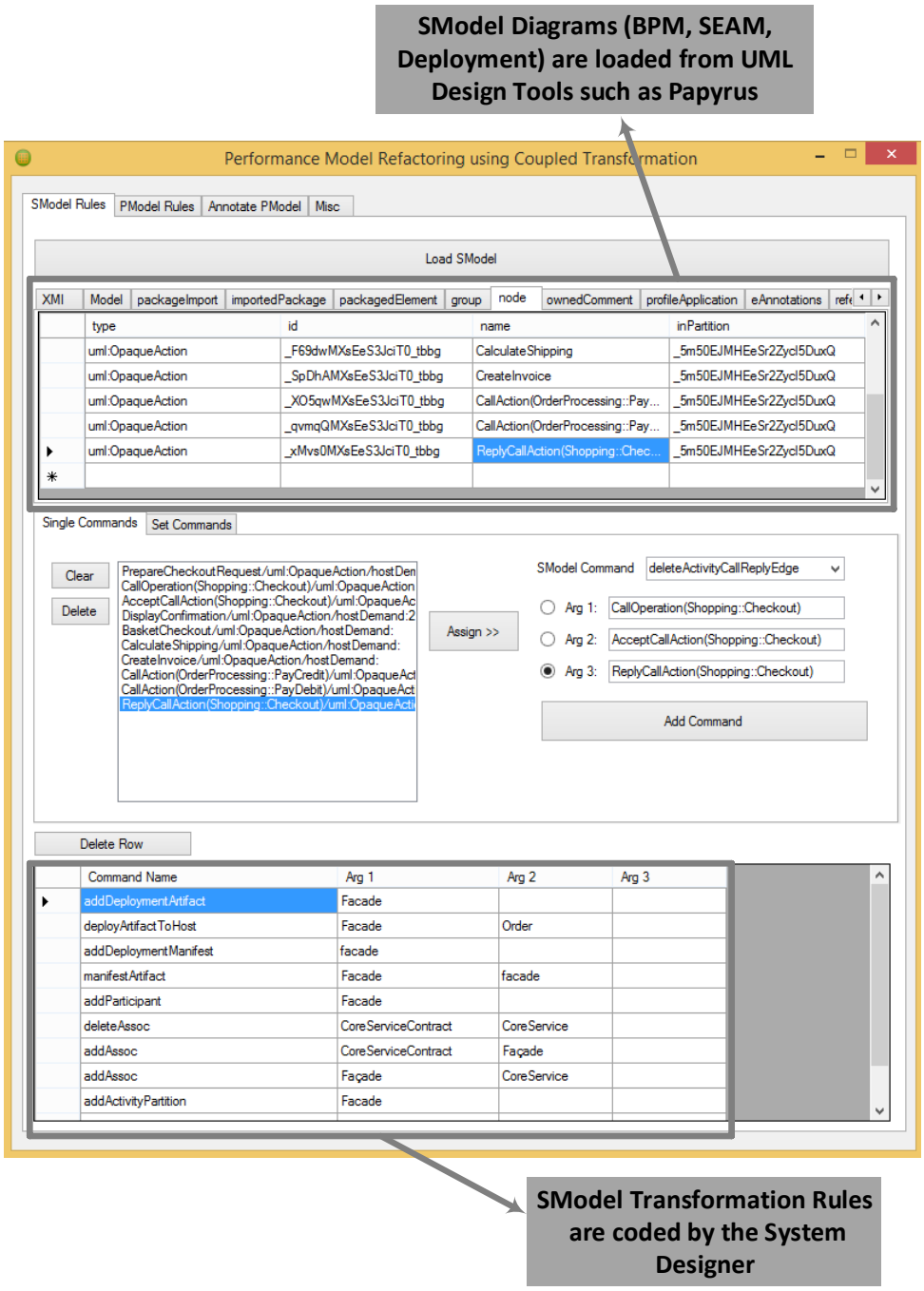

Figure 26: Tool support for recording SModel Transformation Rule

The tool also help the system designer in selecting the SModel elements for the transformation rules by loading the existing SOA design from UML modeling tools such as Papyrus [109] or MagicDraw [110]. Both tools support the SoaML profile and the annotation of SModel with MARTE profile. The SOA design models (i.e. SModel) are

imported into the tool through the files which those UML editors provide. The tool reads the files and organizes the SOA design model elements in tables (i.e. a table for each element type including attributes and MARTE annotations).

The system designer can choose SOA transformation rules from the list of provided rules and assign the SOA design elements as their arguments. The tool provides two categories of SModel transformation rules:

1. Single command rules which impact a single model element
2. Set command rules which impact a set of model elements.

The tool also ensures the quality of the recorded rules to some extent, by:

- Enforcing some constraints such as restricting the designer to choose rules from a pre-defined list, and elements from those loaded from the SModel (i.e. to avoid typos)
- Also the tool makes sure the rules are properly ordered (e.g., associations are deleted before deleting a particular element).

7.2 PModel Transformation Rule Derivation Tool

A screenshot of this tool is shown in Figure 27. This tool loads the mapping table as described in Chapter 4 in form of an xml file, processes the recorded SModel transformation rules and automatically derives the PModel transformation rules. The output of this tool is a set of derived PModel transformation rules, which is used by the annotation tool to annotate the PModel of the SOA with changes.

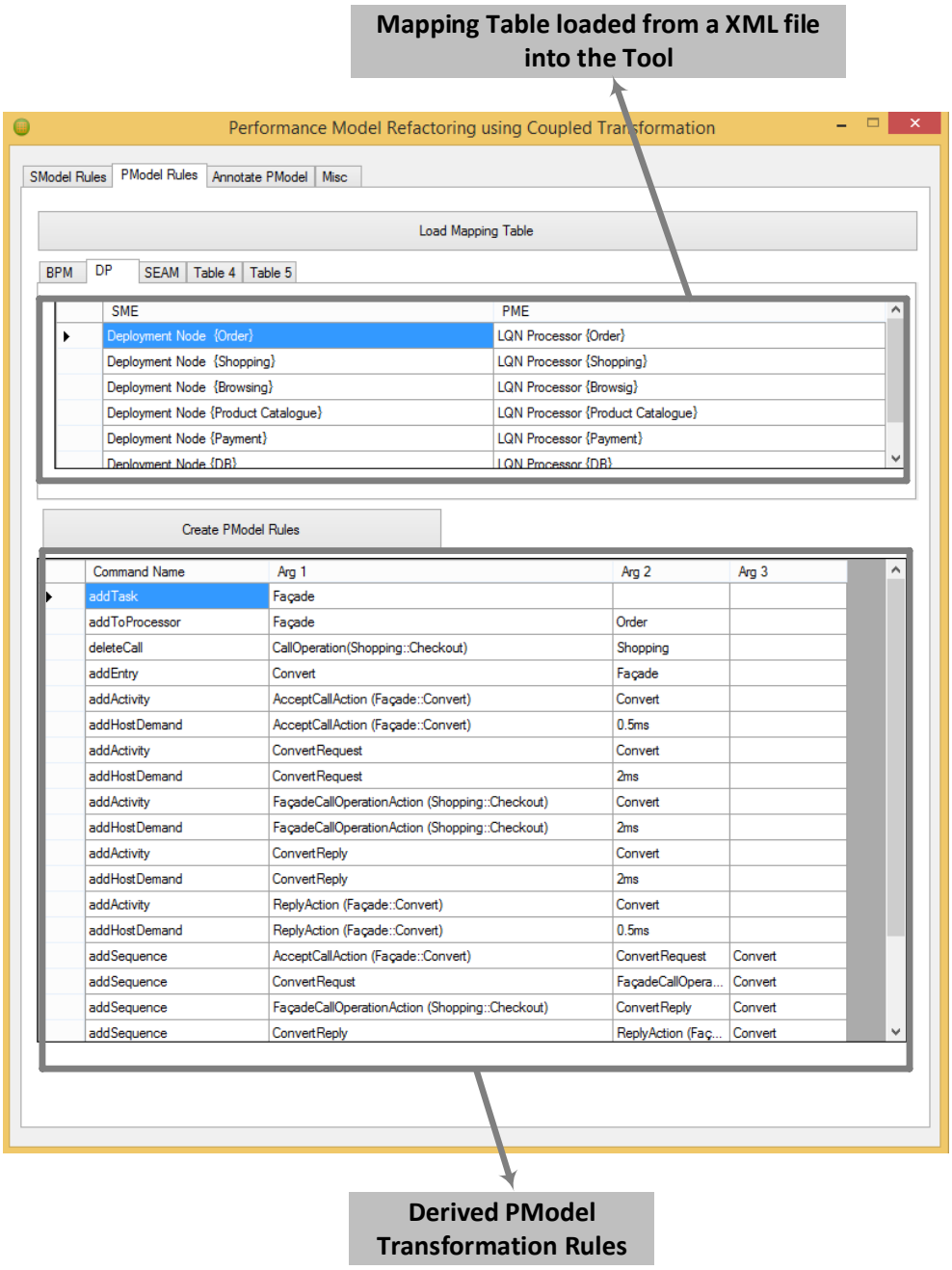

Figure 27: Tool support for automatically deriving PModel transformation rules from SModel transformation rules

7.3 PModel Annotation Tool

Figure 28 shows a screenshot from the tool for annotating the PModel with the derived transformation rules with a few example of annotation tags highlighted and identified.

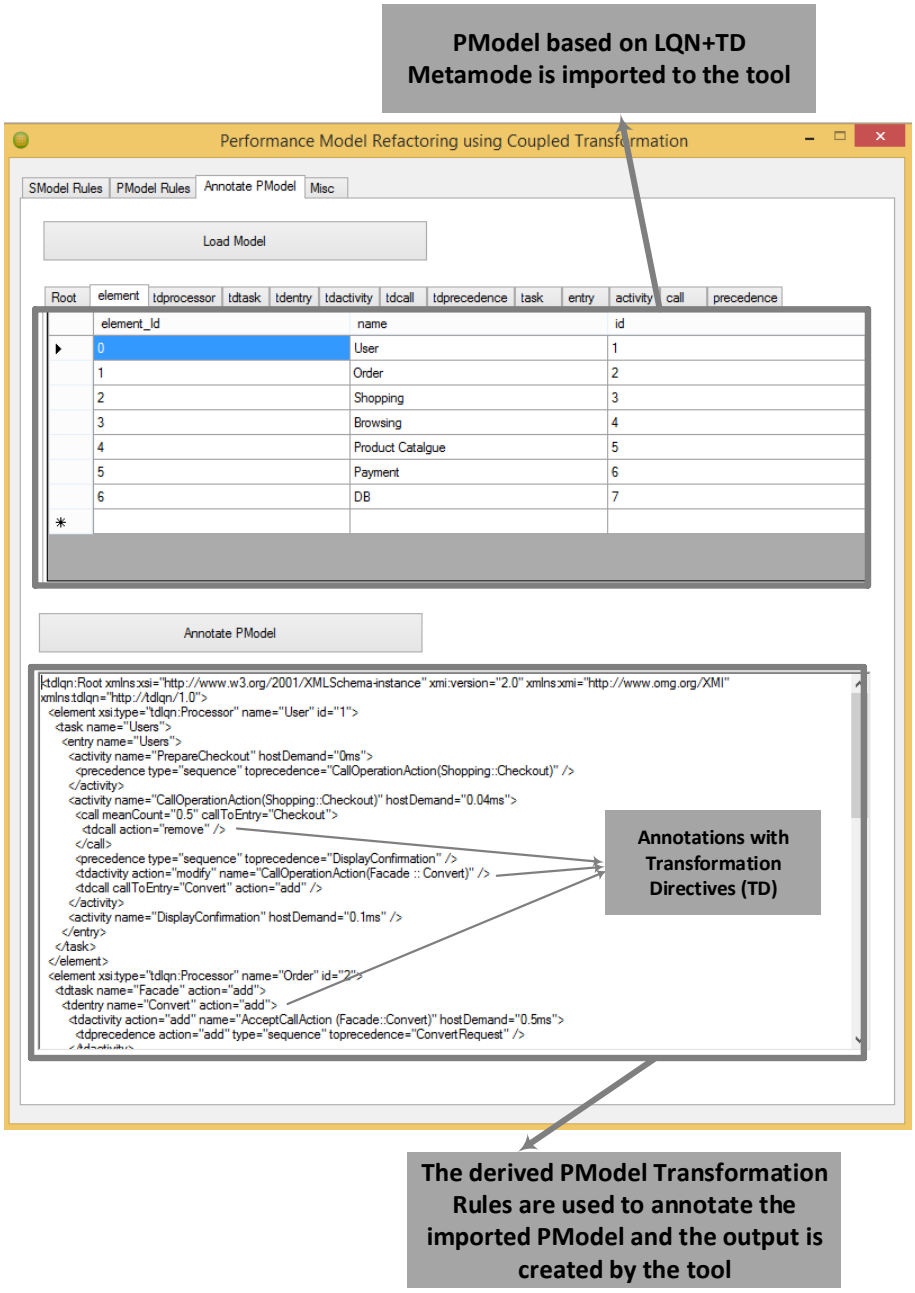

Figure 28: Tool support for annotating the PModel with derived PModel transformation rules

The LQN+TD metamodel (discussed in Section 6.1) is implemented using the "Eclipse Modeling Tools" [111], therefore the models can be created using the Eclipse modeling tool or by a QVT model transformation [12] [112] (i.e. PModel based on LQN metamodel can be transformed to PModel based on LQN+TD metamodel using QVT-O) and processed by the proposed techniques in this thesis.

This tool loads a LQN+TD PModel and automatically annotates it with the derived PModel transformation rules using the TDs.

7.4 QVT-based Transformation Engine

The annotated PModel created by the annotation tool introduced in Section 7.3 is processed by a QVT based transformation engine to refactor the PModel. The Eclipse Modeling Tool with QVT plug in [111, 112] was used to implement this transformation engine based on the technique discussed in Section 6.2. There are two options to develop a QVT transformation using Eclipse:

1. Installing a plug-in metamodel in Eclipse and create the transformation.
2. Using the QVT-O environment in an inner Eclipse instance.

Option number 1 is suitable for a stable and pre-defined metamodel for a transformation project. Option number 2 is suitable if the metamodel is under modification during the process of developing QVT transformation. In this thesis, option number 2 was used while defining the structure of LQN+TD metamodel and then option number 1 was taken once the metamodel was defined and ready to use. The metamodel was created using EMF (Eclipse Modeling Framework). EMF is based on two metamodels; the Ecore and the Genmodel model. The Ecore metamodel contains the information about the defined

classes. The Genmodel contains additional information for the codegeneration, e.g. the path and file information.

The output of this tool is a PModel based on the LQN+TD metamodel. For each PModel element type, the tool first check for removal and modfications annotated with a corresponding TD annotations (TDProcessor for processors, TDTasks for tasks and so on). Then, if a TD subgraph (i.e. a subgraph defined with TD annotations) is defined for the PModel element, it will be processed into depth. Then the tool process the next model element from the same type. Types of the PModel elements are processed in the following order:

- First, the root element (i.e. also called the model) is traversed for any TDProcessor which added a new processor to the system. If a TD subgraph is defined for the TDProcessor (i.e. defined with TDTask, TDEntry, TDActivity,TDCalls, TDPrecedence, TDForward), it will be processed as well.
- The tool processes all the model processors for TDProcessor annotation (for removal and modification) and then the attached TD subgraph defined by annotations (i.e. defined with TDTask, TDEntry, TDActivity,TDCalls, TDPrecedence, TDForward) for the added tasks, entries, activities, etc.
- The tool processes all the PModel Tasks for the TDTask annotations (for removal and modification) and then TD subgraph defined by the annotation (i.e. defined with TDEntry, TDActivity, TDCalls, TDPrecedence, TDForward) for added entries, activities, etc.
- The tool processes all the PModel Entries for the TDEntry annotations (for removal and modification) and then the TD subgraph defined by the annotation

(i.e. defined with TDActivity, TDCalls, TDPrecedence, TDForward) for added activities, etc.

- The tool process all the PModel Activities for TDActivity annotations (for removal and modification) and then the TD subgraph defined by the annotation (i.e. defined with TDCalls, TDPrecedence, TDForward) for added calls, precedence, and forwards.
- At end the tool processes all the PModel Calls, Precedencies, and Forwards for TDCall, TDForward, and TDPrecedence annotations respectively (for removal and modification).

Figure 29 shows a pseudocode which processes TD annotations for PModel processors which add tasks (i.e. TDTask attached to PModel Processor, part of the second process mentioned above). In Figure 29 code, TDTask refers to the set of TDTask annotations belonged to a PModel processor and this concept also applied to TDEntry, TDActivity, TDCall, TDPrecedence, and TDForward

The process starts when the PModel processor is checked for TDProcessors attached to it and there is no TDProcessor with action set to “Remove” or “modify” (lines 2-4). If true, it traverses the set of attached TDTask annotations to the processor and adds all the tasks and the subgraph defined to them to the PModel processor (lines 6-15). Then in a nested loop, it traverses all the TDEntry annotations to process all the added entries and similarly for all the activities inside a newly added entry (lines 16-26). For every TDActivity, the program traverses all the TDCall, TDPrecedence, and TDForward annotations (lines 32-44). For processing of the TD annotations at each level, a new instance from the PModel element is created and all the attributes are set based on the

info extracted from corresponding TD annotation. Then, the remaining part of subgraph is copied into the newly created element.

```
mapping Processor :: P2P() : Processor
    when    {
    self.TDProcessor-> first().action <> "Remove" and
    self.TDProcessor-> first().action <> "Modify"
    }
    {
        var countT: Integer;
        countT:=0;
        foreach (TDTask attached to Processor)
        {
        if(self.TDTask->at(count).action = "AddTask")then
         {
           var newTask : LQN:: Task := object LQN::Task{name:= self.TDTask->at(countT).name; multiplicity:= self.TDTask->at(countT).multiplicity}; -- creating new task
           newTask.TDEntry += self.TDTask.TDEntry; -- moving the remaining part of the subgraph the the new entry
           var countE: Integer;
           countE:=0;
           foreach (TDEntry attached to newTask)
           {
               if(newTask.TDEntry->at(countE).action = "AddEntry")then
               {
                var newEntry : LQN:: Entry := object LQN::Entry{name:= newTask.TDEntry->at(countE).name}; -- creating new entry
                newEntry.TDActivity += newTask.TDEntry.TDActivity;
                var countA: Integer;
                countA:=0;
                    foreach (TDActivity attached to newEntry)
                    {
                        if(newEntry.TDActivity->at(countA).value = "AddActivity")then
                        {
                          var newActivity : LQN:: Activity := object LQN::Activity{name:= newEntry.TDActivity->at(countA).name};
                          newActivity.TDCall += newTask.TDEntry.TDActivity.TDCall;
                          newActivity.TDPrecedence += newTask.TDEntry.TDActivity.TDPrecedence;
                          newActivity.TDForward += newTask.TDEntry.TDActivity.TDForward;
                          foreach (TDCall attached to newActivity)
                          {
                           -- Process TDCall
                          }
                          foreach (TDPrecedence attached to newActivity)
                          {
                           -- Process TDPrecedence for Sequence, Branch , Merge, Fork , Join
                          }
                          foreach (TDForward attached to newActivity)
                          {
                           -- Process TDForward
                          }
                          newEntry.activity += newActivity;
                        }endif;
                     countA:=countA+1;
                    }
                newTask.entry += newEntry;
                }endif;
             countE:=countE+1;
           }
        result.task+=newTask;
        }endif;
        countT:=countT+1;
        }
    }
```

Figure 29 : QVT-Based Transformation Engine: Pseudocode for processing of TD Annotations for PModel processors which add Tasks

Chapter 8 : Case Study, Verification and Validation

This Chapter presents an extended case study and also the processes used for the verification and validation of the coupled transformation technique. The design of a SOA based Shopping and Browsing system is used for the case study, and for verification and validation. It is assumed that the initial performance analysis of the SOA is done based on the PUMA transformation. The starting point is the Shopping and Browsing preliminary design presented in Section 2.2, with the performance model generated by the PUMA transformation given in Figure 7 of Section 2.4.

In this thesis, the coupled transformation technique implementation is validated by a designed test suite with test cases described in Section 8.1. The designed test suite contains test scenarios and test cases to verify the functionalities of the two main automated units of the process: 1) Derivation of performance model transformation rules from SOA design transformation rules; 2) QVT based performance model transformation. The test suite uses ten SOA design patterns which are categorized based on the changes they can impose on the LQN performance model. Design patterns from each category are chosen to set up the test conditions for test cases in the test suite.

Sections 8.2 and 8.3 verify the efficiency and effectiveness of the overall coupled transformation technique and the specific approach taken here, by using it in the application of the design patterns to the case study, including performance analysis of the resulting designs.

8.1 Testing for System Validation

The approach in this thesis and the tools described in Chapter 7 are validated using a test suite with set of unit tests and ten Test Scenarios (TS). Unit tests are discussed in Section 8.1.1 and test scenarios in Section 8.1.2. For the present purposes a test scenario is a set of test cases that ensure that the approach and the process flows are tested from end to end. They are a series of tests that follow each other, each dependent on the output of the previous one. Test cases are low level actions, the set of valid and invalid executable procedures of a test scenario.

8.1.1 Unit Testing

To ensure that most of the major functionalities of the approach are tested, the transformation operations were categorized based on their impact on the LQN performance model. Unit test cases were created for all the operations in each category and they were run independently. The operation categories are:

1- Operations which add/delete/modify LQN processors

2- Operations which add/delete LQN tasks

3- Operations which add/delete/modify LQN activities

4- Operations which add/delete LQN precedence relationships

5- Operations which add/delete LQN Sync Calls

6- Operations which add/delete LQN Async Calls

7- Operations which add/delete LQN Forwarding Calls

8.1.2 Test Scenarios

To ensure that the methods created for refactoring of the performance model can work together to apply a design pattern, ten test scenarios (TSs) were created, each integrating

several operations for applying a design pattern. Table 6 shows the test cases included in each test scenario, by showing which transformation operations it covers.

For setting up the test conditions for the test cases in each TS, the system under test (i.e. the Shopping and Browsing SOA) and ten SOA design patterns were used. The TSs are as follows:

- TS1: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Façade Design Pattern" to the shopping and browsing SOA. In this test scenario, the façade design pattern is used to implement a multi-channel end point which facilities the connectivity of various user types (e.g. Mobile, Desktop, Kiosk) for a core service. "Service Façade" is defined as follows [1]:
    - ***Problem****: The tight coupling of the core service logic to its contracts can obstruct its evolution and negatively impact service consumers.*
    - ***Solution:*** *Façade logic is inserted into the service architecture to establish a layer of abstraction that can adopt future changes to the core service contract.*

  To avoid major changes to the core service and also facilitate the future changes to the multi-channel end point (i.e. adding more user types), the system designer may decide to use this design pattern to implement the multi-channel capability.
- TS2: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Service Decomposition" pattern to the shopping and browsing SOA. This design patterns requires that some elements to be deleted

from model, but after some behavior has been duplicated into a newly added component. Service Decomposition is defined as follows [1]:

- ***Problem****: To serve a large and complex business task, a corresponding amount of solution logic (service) needs to be created, resulting in a self-contained application with traditional governance and reusability constraints.*
- ***Solution:*** *The large business task (i.e. service) should be broken down into a set of smaller, related tasks, leading to a corresponding set of smaller, related services which satisfy those tasks.*

- TS3: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Security Design Pattern" to the shopping and browsing SOA. Of the many possible ways to improve, the Secure Sockets Layer was chosen here. For SSL the pattern description is defined as follows [1]:
  - ***Problem:*** *Communication between client (browser) and the service provider over the internet needs to be secure*
  - ***Solution:*** *A protocol called SSL allowing for encryption/decryption of client requests and server responses to protect message data at the transport layer needs to be used on both client and server sides.*
- TS4: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Redundant Implementation" to the shopping and browsing SOA. The redundant Implementation design pattern is defined as follows [1]:

- ***Problem:*** *A service that is being actively in use introduces a potential single point of failure that may risk the reliability of all components which they dependent on if an unexpected error condition occur.*
- ***Solution:*** *High demand services can be deployed using redundant implementations or with failover support.*

- TS5: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Partial State Deferral" to the shopping and browsing SOA. Partial State Deferral design pattern is defined as below [1]:
  - ***Problem:*** *Service may be required to store and manage large amounts of data, causing increased memory consumption and slow response time.*
  - ***Solution:*** *A subset of services' state data can be temporarily deferred even when services are required to remain stateful.*
- TS6: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Asynchronous Queuing" pattern to the shopping and browsing SOA. The asynchronous Queuing design pattern is defined as below [1]:
  - ***Problem:*** *When a service capability requires that consumers interact with it synchronously, it can inhibit performance and compromise reliability.*
  - ***Solution:*** *A service can communicate the requests with its clients via an intermediate queuing buffer that receives request messages and then forwards them on behalf of the service consumers. Then a respond is ready for the client, the service sends the reply directly to the client.*

- TS7: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Service Callback" pattern to the shopping and browsing SOA. Service Callback is defined as follows [1]:
    - ***Problem:*** *Communicating with service clients synchronously through the issuance of multiple messages or when service message processing requires a large amount of time can be resource intensive and costly.*
    - ***Solution:*** *A service can require that consumers communicate with it asynchronously and provide a callback address to which the service can send response messages.*
- TS8: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Concurrent Contracts" pattern to the shopping and browsing SOA. Concurrent Contracts is defined as follows [1]:
    - ***Problem:*** *A service may have to serve clients with various needs. A service's contract may not be applicable to all potential service clients.*
    - ***Solution:*** *Multiple contracts can be created for a service, each specialized at a specific type of consumer.*
- TS9: This test scenario verifies the behavior of some of the methods (shown in Table 6) by applying the "Event-Driven Messaging" pattern to the shopping and browsing SOA. Event-Driven Messaging is defined as follows [1]:
    - ***Problem:*** *Following traditional Message Exchange Patterns (MEPs), a service client would need to repeatedly poll the service in order to find out whether a specific event had occurred which is inefficient because it leads to numerous unnecessary service invocations.*

- **Solution:** *The client subscribes to the service asynchronously with a callback address for one or more specific events. The service, in turn, automatically issues notifications of relevant events to this and any of its subscribers.*

Table 6: Test Scenarios for the verification of the Coupled Transformation Technique

PModel Artifacts	*Processor*			*Task*		*Entry*		*Activity*			*Precedence*		*Sync Call*			*Async Call*			*Forwarding Call*	
TS#	**Add**	**Delete**	**Modify**	**Add**	**Delete**	**Add**	**Delete**	**Add**	**Delete**	**Modify**	**Add**	**Delete**	**Add**	**Delete**	**Modify**	**Add**	**Delete**	**Modify**	**Add**	**Delete**
TS1			X	X		X		X		X	X		X	X						
TS2			X	X		X	X	X			X		X	X						
TS3								X			X	X								
TS4	X			X		X		X			X		X							
TS5								X			X				X					
TS6										X			X	X	X				X	
TS7									X			X	X	X					X	
TS8						X		X			X									
TS9						X										X				
TS10								X								X				

- TS10: This test scenario verifies the behavior of some of the methods (Shown in Table 6) by applying the "User Interface Mediator" pattern to the shopping and browsing SOA. User Interface Mediator is defined as follows [1]:
  - **Problem:** *Service clients expect responses to requests in a timely manner and the underlying services may not be designed or able to provide these*

responses in a synchronous and proper time. Poor or inconsistent user experience can lead to a decrease in the usage and overall success of the solution.

- ***Solution:*** *An intermediary service (Called UI Mediator) is placed between a service or service composition and the frontend solution user-interfaces. The service is responsible for providing the user with continuous feedback and for gracefully facilitating various runtime conditions so that the underlying processing of the services does not affect the quality of the user experience.*

8.2 Overall Effectiveness of the Coupled Transformation Approach

In this section, the coupled transformation approach is used for the application of nine additional design patterns beside Service Façade (Service Decomposition, Security Design Patterns, Service Callback, Redundant Implementation, Partial State Deferral, Asynchronous Queuing, Concurrent Contracts, Event-Driven Messaging, User Interface Mediator) to the Shopping and Browsing SOA to validate its effectiveness for the purpose of performance analysis by the system designer. For each design pattern, the recorded SModel transformation rules, the derived PModel transformation rules and the refactored PModel are provided.

8.2.1 Application of Service Decomposition Pattern

First we suppose the designer decision is to apply Service Decomposition (a popular SOA design pattern) and decompose the" Shopping and Browsing" service shown in Figure 30 into two smaller services. The description of the problem and solution for this pattern is provided in Section 8.1.

To apply it, the designer must decide (1) whether the decomposed services should run on one host or two, and (2) the thread pool size of each decomposed task. The LQN model before application of the Service Decomposition Pattern is shown in Figure 30.

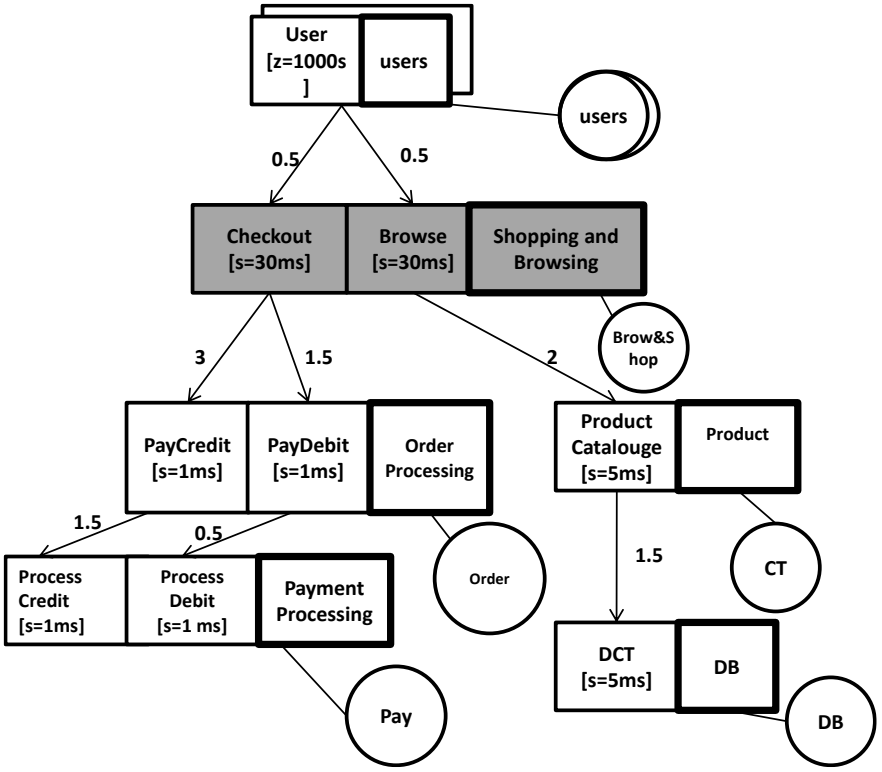

Figure 30: LQN model for Shopping and Browsing SOA before Service Decomposition

Suppose that the system designer decided to deploy each decomposed task into two individual hosts, and has chosen pool sizes for the tasks. The host that runs the task before decomposition is reused in the model after the decomposition to run one of the decomposed tasks, and a new processing host is created to run the other task. Table 7 shows the recorded SModel transformation rules for the SModel Deployment diagram and BPM. The rules for the deployment diagram include creating a new processing node, deploying a new artifact into it and renaming the other processing node to have a suitable name for its new role. Table 8 shows the derived PModel transformation rules from the SModel transformation rules shown in Table 7.

Table 7: SModel Transformation rules for Shopping and Browsing SOA and Service Decomposition Design Pattern

SModel Rules

Deployment Diagram

```
D1.    addDeploymentArtifact(Browsing)
D2.    addDeploymentHost(BrowsingProc)
D3.    deployArtifactToHost(Browsing, BrowsingProc)
D4.    renameDeploymentArtifact(ShoppingAndBrowsing,Shopping)
D5.    renameDeploymentHost(Brow&Sho,ShoppingProc )
```

BPM (UML AD) Diagram

```
B1 deleteActionCallReplyEdge (CallOperation(Product::ProductCatalogue), AcceptCall
(Product::ProductCatalogue), ReplyAction(Product::ProductCatalogue);
B2 deleteActivity(ShoppingAndBrowsing, Browse)
              {
B2.2                    deleteActionCallReplyEdge (CallOperation(Browsing::Browse),
AcceptCall (Browsing::Browse), ReplyAction(Browsing::Browse));
B2.3                    deleteControlFlow(AcceptCallAction (Browsing::Browse),
AddSelectionCriteria);
B2.4                    deleteControlFlow(AddSelectionCriteria,
RetrieveCatalogueCallOperationAction (Product::ProductCatalogue));
B2.5                    deleteControlFlow(RetrieveCatalogueCallOperationAction
(Product::GenDBQuery), FormatCatalogue);
B2.6                    deleteAcceptCallAction (AcceptCallAction (Browsing::Browse),
Sync);
B2.7                    deleteAction (AddSelectionCriteria);
B2.8                    deleteCallOperationAction (RetrieveCatalogueCallOperationAction
(Product::ProductCatalogue), Sync);
B2.9                    deleteAction (FormatCatalogue) ;
B2.10                   deleteReplyAction (ReplyAction (Browsing::Browse));

                         }
B3 addActivityPartition (Browsing);
B4 addActivity(Browsing, Browse);
              {
B4.1                    addAcceptCallAction (AcceptCallAction (Browsing::Browse), Sync);
B4.2                    SetPaStepHostDemand (AcceptCallAction (Browsing::Browse), '10ms');
B4.3                    addAction (AddSelectionCriteria);
B4.4                    SetPaStepHostDemand (AddSelectionCriteria, '10ms');
B4.5                    addCallOperationAction (RetrieveCatalogueCallOperationAction
(Product::ProductCatalogue), Sync);
B4.6                    addAction (FormatCatalogue) ;
B4.7                    SetPaStepHostDemand (FormatCatalogue, '10ms');
B4.8                    addReplyAction (ReplyAction (Browsing::Browse));
B4.9                    addControlFlow(AcceptCallAction (Browsing::Browse),
AddSelectionCriteria);
B4.10                   addControlFlow(AddSelectionCriteria,
RetrieveCatalogueCallOperationAction (Product::ProductCatalogue));
B4.11                   addControlFlow(RetrieveCatalogueCallOperationAction (Product::
ProductCatalogue), FormatCatalogue);
              }
B5 addActionCallReplyEdge (RetrieveCatalogueCallOperationAction
(Product::ProductCatalogue),  AcceptCallAction (Product::ProductCatalogue),
ReplyAction(Product::ProductCatalogue));
B6 SetPaStepHostProb(CallOperationAction (RetrieveCatalogueCallOperationAction
(Product::ProductCatalogue), '2');
B7 renameActivityPartition (ShoppingAndBrowsing, Shopping);
B8 addActionCallReplyEdge (CallOperationAction (Browsing::Browse), AcceptCallAction
(Browsing::Browsee), ReplyAction(Browsing::Browse));
B9 SetPaStepHostProb(CallOperationAction (Browsing::Browse), '0.5');
B10 addActionCallReplyEdge (CallOperation(Product::ProductCatalogue), AcceptCall
(Product::ProductCatalogue), ReplyAction(Product::ProductCatalogue);
B11 SetPaStepHostProb(CallOperation(Product::ProductCatalogue),'2');
```

Table 8: Derived PModel Transformation rules for Shopping and Browsing SOA and Service Decomposition Design Pattern

PModel Rules
LQN
L1 deleteCall(CallOperationAction(Browsing::Browse), Browse); L2 deleteCall(CallOperationAction(Product::ProductCatalogue), ProductCatalogue); L3 deleteEntry(Browse,ShoppingAndBrowsing); L4 addTask(Browsing); L5 addProcessor(BrowsingProc); L6 addToProcessor(Browsing, BrowsingProc); L7 addEntry(Browse, Browsing); L8 addActivity(AcceptCallAction,Phase1,Browse); L9 addActivity(AddSelectionCriteria,Phase1,Browse); L10 addHostDemand(AddSelectionCriteria, '10ms'); L11 addActivity(RetrieveCatalogueCallOperationAction,Phase1,Browse); L12 addHostDemand(RetrieveCatalogueCallOperationAction, '10ms'); L13 addActivity(FormatCatalogue,Phase1,Browse); L14 addHostDemand(FormatCatalogue, '10ms'); L15 addActivity(ReplyAction,Phase1,Browse); L16 addSequence(AcceptCallAction,AddSelectionCriteria,Phase1); L17 addSequence(AddSelectionCriteria,RetrieveCatalogueCallOperationAction,Phase1); L18 addSequence(RetrieveCatalogueCallOperationAction,FormatCatalogue,Phase1); L19 addSequence(FormatCatalogue,ReplyAction,Phase1); L20 addCall(RetrieveCatalogueCallOperationAction(Product::ProductCatalogue), Product, '2'); L21 addCall(CallOperationAction(Browsing::Browse), Browse , '0.5'); L22 addCall(CallOperationAction(Product::ProductCatalogue), ProductCatalogue , '2'); L23 renameTask(ShoppingAndBrowsing, Shopping) L24 renameProcessor(Brow&Sho,ShoppingProc)

The PModel rules in Table 8 first delete the Browse entry from the ShoppingAndBrowsing task, then create a new task with a new Browse entry and recreate all the activities of the deleted Browse entry. At end, they create new calls to the newly created task and entry and rename the processor and task name for the decomposed task. The LQN model after applying the service decomposition pattern is shown in Figure 31 with the new tasks shaded.

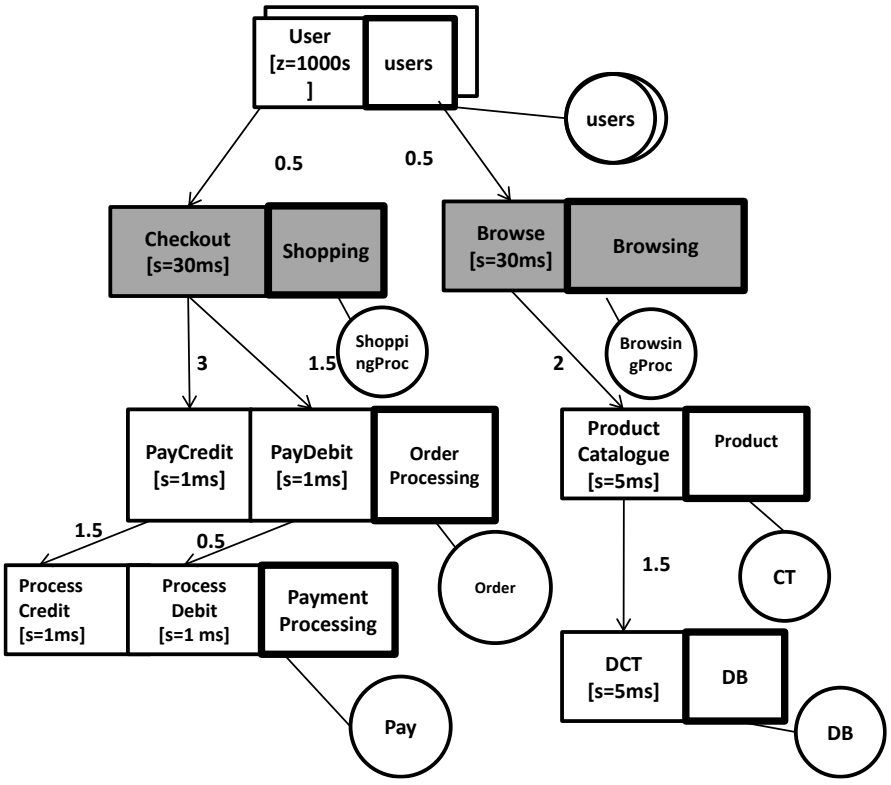

Figure 31: LQN model for Shopping and Browsing SOA after Service Decomposition

8.2.2 Security Design Pattern

Distributed systems such as SOA systems have many users and have to meet different and sometimes conflicting non-functional requirements, such as security and performance. For example, a highly secure system may pay a performance price compared to an unsecure system, due to the extra security checks it must do. System designers need to make choices between competing design solutions in order to satisfactorily balance system requirements. The technique in this thesis can help them to evaluate the performance effect of a security design pattern applied to SOA. In this section, a security pattern is applied to Shopping and Browsing SOA. The description of the problem and solution for this pattern is provided in Section 8.1.

Secure Sockets Layer (SSL) is a cryptographic protocol that provides communications security over the Internet. For the SSL security pattern to be applied, some extra functions are required to be implemented to encrypt the communication data from the sender and decrypt them for the receiver. Figure 32 shows a part of SModel BPM diagram with the added actions (shown with grey color in Figure 32) for the user to encrypt information before calling the checkout operation from the shopping class and decrypt the information after receiving the reply. Similarly, actions are added to the checkout process in the shopping swimlane to decrypt the data after accepting the call and to encrypt the reply to the user before sending it.

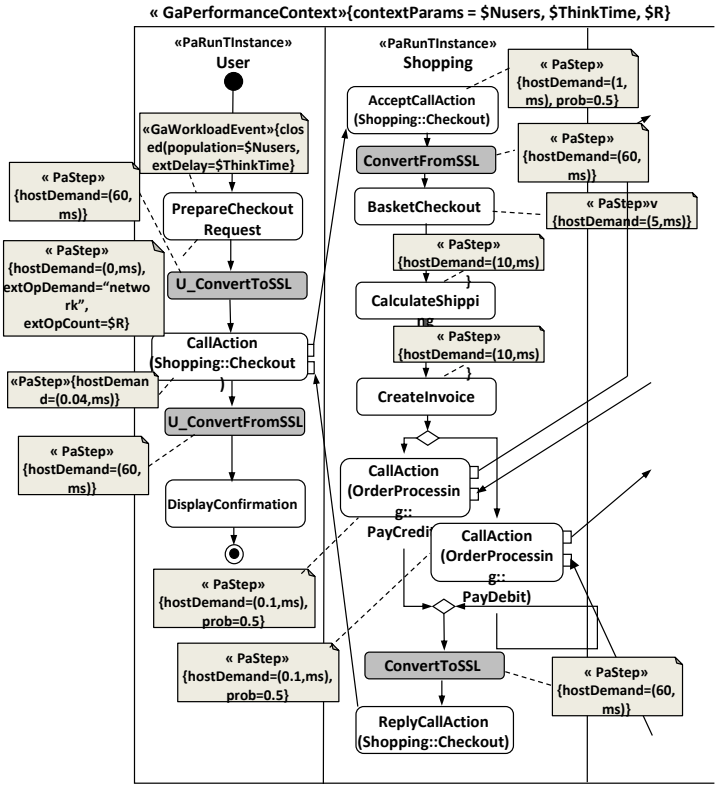

Figure 32: Partial Checkout Business Process Model for the Online Shop with SSL Security Design Pattern

Table 9 shows the recorded SModel transformation rules for the added actions to SModel design. In Table 9, both user and checkout subgraphs (i.e. UML activites) are modified to have the encryption and decryption actions. First the calls are deleted and then the new actions are added. Then, the some new calls are added to the newly created actions. Table 10 shows the derived PModel transformation rules from the SModel transformation rules shown in Table 9.

Table 9: SModel Transformation rules for the Shopping and Browsing SOA with SSL Security Design Pattern

SModel Rules
BPM (UML AD) Diagram

B1 modifyActivity(Users, User)
{
B1.1 deleteControlFlow(PrepareCheckoutRequest, AcceptCallAction (Shopping::Checkout));
B1.2 deleteControlFlow(AcceptCallAction (Shopping::Checkout),DisplayConfirmation);
B1.3 addAction (U_ConvertToSSL);
B1.4 SetPaStepHostDemand (U_ConvertToSSL, '60ms');
B1.5 addControlFlow(PrepareCheckoutRequest, U_ConvertToSSL);
B1.6 addControlFlow(U_ConvertToSSL, CallOperationAction(Shopping::Checkout));
B1.7 addAction (U_ConvertFromSSL);
B1.8 SetPaStepHostDemand (U_ConvertFromSSL, '60ms');
B1.9 addControlFlow(AcceptCallAction (Shopping::Checkout), U_ConvertFromSSL);
B1.10 addControlFlow(U_ConvertFromSSL, DisplayConfirmation);
}
B2 modifyActivity(Shopping, Checkout)
{
B2.1 deleteControlFlow(AcceptCallAction (Shopping::Checkout), BasketCheckout);
B2.2 deleteDecision(CallAction(OrderProcesing::PayCredit),CallAction(OrderProcesing::PayDebit),ReplyAction(Shopping::Checkout));
B2.3 addAction (ConvertFromSSL);
B2.4 SetPaStepHostDemand (ConvertFromSSL, '60ms');
B2.5 addControlFlow(AcceptCallAction (Shopping::Checkout), ConvertFromSSL);
B2.6 addControlFlow(ConvertFromSSL, BasketCheckout);
B2.7 addAction (ConvertToSSL);
B2.8 SetPaStepHostDemand (ConvertToSSL, '60ms');
B2.9 addDecision(CallAction(OrderProcesing::PayCredit),CallAction(OrderProcesing::PayDebit),ConvertToSSL);
B2.10 addControlFlow(ConvertToSSL, ReplyAction(Shopping::Checkout));
}

Table 10: Derived PModel Transformation rules for Shopping and Browsing SOA and SSL Security Design Pattern

PModel Rules
LQN
L1 deleteSequence(PrepareCheckoutRequest,AcceptCallAction (Shopping::Checkout),Phase1); L2 deleteSequence(AcceptCallAction (Shopping::Checkout),DisplayConfirmation,Phase1); L3 addActivity (U_ConvertToSSL, Checkout); L4 addHostDemand(U_ConvertToSSL, '60ms'); L5 addSequence(PrepareCheckoutRequest,U_ConvertToSSL,Phase1); L6 addSequence(U_ConvertToSSL,CallOperationAction(Shopping::Checkout),Phase1); L7 addActivity (ConvertFromSSL, Checkout); L8 addHostDemand(ConvertFromSSL, '60ms'); L9 addSequence(CallOperationAction(Shopping::Checkout),ConvertFromSSL,Phase1); L10 addSequence(ConvertFromSSL,ReplyAction(Shopping::Checkout),Phase1); L11 deleteSequence(AcceptCallAction,BasketCheckout,Phase1); L12 deleteDecision(CallAction(OrderProcesing::PayCredit),CallAction(OrderProcesing::PayDebit) ,ReplyAction(Shopping::Checkout)); L13 addActivity (ConvertFromSSL, Checkout); L14 addHostDemand(ConvertFromSSL, '60ms'); L15 addSequence(AcceptCallAction,ConvertFromSSL,Phase1); L16 addSequence(ConvertFromSSL,BasketCheckout,Phase1); L17 addActivity (ConvertToSSL, Checkout); L18 addHostDemand(ConvertToSSL, '60ms'); L19 addDecision(CallAction(OrderProcesing::PayCredit),CallAction(OrderProcesing::PayDebit),Co nvertToSSL); L20 addSequence(ConvertToSSL,ReplyAction(Shopping::Checkout),Phase1);

Figure 33 shows the part of LQN PModel with the added activities. The action changes in the SModel are mapped to the changes in the activities inside the LQN entries. The added activities inside each entry for the security pattern are highlighted with grey color.

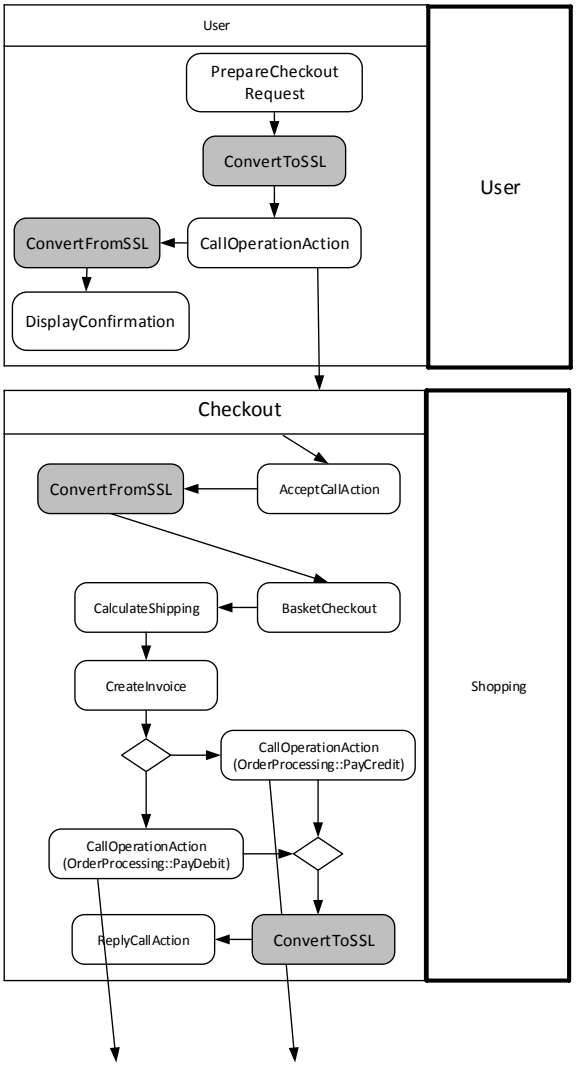

Figure 33: Partial LQN model after application of SSL Security Design Pattern

8.2.3 Service Callback Pattern

In the Shopping and Browsing SOA (partially shown in Figure 34), the "OrderProcessing" class in the SModel is in charge of preparing a credit charge request on behalf of "Shopping" class. The requests are sent to the "PaymentService" for processing. Therefore "OrderProcessing" is an intermediate task between the "Shopping" and "PaymentProcessing". "PaymentProcessing" provides the "OrderProcessing" with a confirmation which will be passed to "Shopping", the initial sender of the request. The

system designer notices that there is no need for the “OrderProcessing” to wait for the reply of the “PaymentProcessing” while the final destination of the confirmation is the instance of the “Shopping” class. This behaviour is a perfect candidate for the Service Callback design pattern. The description of the problem and solution for this pattern is provided in Section 8.1. Using this pattern, “OrderProcessing” forwards the “Shopping” request after some preparation to the “PaymentProcessing” entry, with a callback address of the initial sender which is the “Shopping” entry. “PaymentProcessing” can issue a confirmation directly to the “Shopping” entry. Figure 34 shows the SModel after application of Service Callback to the SOA under study. The actions involved in making this forwarding request happen in SModel are shown with grey color in Figure 34. This requires that “Shopping” makes a Synchronous call to the “OrderProcessing” like before using CallOperationAction(OrderProcessing::PayCredit), meaning it still waits for a reply. Although, the reply will not be coming from “OrderProcessing” as it forwards the “Shopping” request using an asynchronous call to “PaymentProcessing” using CallOperationAction(PaymentProcessing::ProcessCredit).

Finally, ReplyCallAction(PaymentProcessing::ProcessCredit) issues the confirmation directly to CallAction(OrderProcessing::PayCredit).

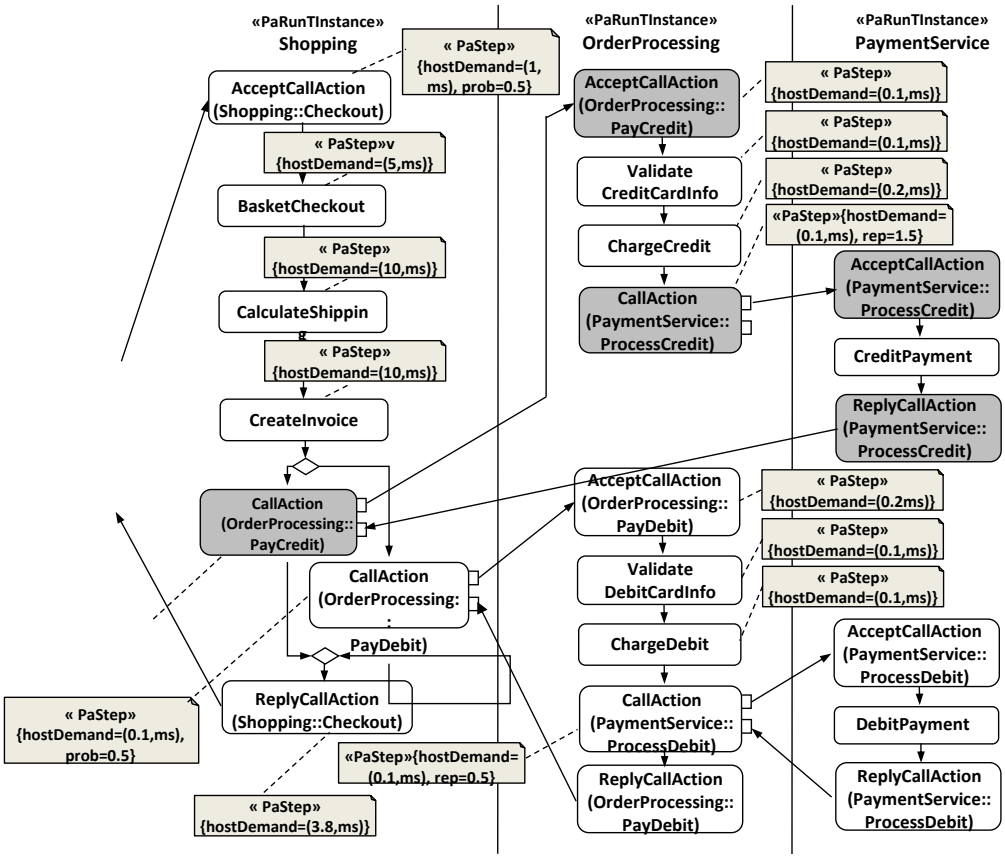

Figure 34: Partial Checkout Business Process Model for the Online Shop with Service Call Back for ProcessCredit

Table 11 shows the recorded SModel transformation rules for application of Service Callback design pattern to the under study SOA. The two synchronous calls by the Shopping and the OrderProcessing classes are deleted and instead a synchronous call is created from the Shopping swimlane with its reply coming from a swimlane which is not the same as recipient of the call and an asynchronous call which forwards requests to PaymentProcessing.

Table 11: SModel Transformation rules for Shopping and Browsing SOA and Service Callback pattern

SModel Rules
BPM (UML AD) Diagram

B1 deleteActionCallReplyEdge (CallOperationAction (OrderProcessing::PayCredit), AcceptCallAction (OrderProcessing::PayCredit), ReplyCallAction(OrderProcessing::PayCredit));
B2 modifyActivity(OrderProcessing, PayCredit)
{
B2.1 deleteControFlow(CallAction(PaymentProcessing::PayCredit), ReplyAction(OrderProcessing::ProcessCredit));
B2.2 deleteAction(ReplyAction(OrderProcessing::ProcessCredit));
}
B2 deleteActionCallReplyEdge (CallOperationAction (PaymentService::ProcessCredit), AcceptCallAction (PaymentService::ProcessCredit), ReplyCallAction(PaymentService::ProcessCredit));
B3 addActionCallReplyEdge (CallOperationAction (OrderProcessing::PayCredit), AcceptCallAction (OrderProcessing::PayCredit), ReplyCallAction(PaymentService::ProcessCredit));
{
B3.1 addActionCallEdge(AcceptCallOperation(PaymentService::ProcessCredit), AcceptCallOperation (PaymentService::ProcessCredit)) ;
}

Table 12 shows the derived PModel transformation rules from the SModel transformation rules in Table 11. Although there is no change to the incoming call into the PayCredit entry in the PModel, it needs to be deleted as the consequence of rule B1 in the SModel transformation rules (Table 11). The reason is that in the SModel, the ReplyAction and its reply call needs to be deleted from "OrderProcessing" (B2.1 and B2.2 in Table 11) and the call and the reply call in SModel are both mapped to a call from a CallOperationAction to that entry. As a result the call from the activity to the entry is deleted from the PModel (L1 in Table 12) and recreated during the process of forwarding call creation (L5 and L6 in Table 12). Figure 35 shows the part of LQN PModel with the changes due to application of Service Callback design pattern.

Table 12: Derived PModel Transformation rules for Shopping and Browsing SOA and Service Callback Pattern

PModel Rules
LQN
L1 deleteCall(CallOperationAction (OrderProcesing::ProcessCredit),PayCredit); L2 deleteCall(CallOperationAction (PaymentService::PayCredit),PayCredit); L3 deleteSequence(CallAction(PaymentProcessing::PayCredit), ReplyAction(OrderProcessing::ProcessCredit)); L4 deleteActivity(ReplyAction(OrderProcessing::ProcessCredit)); L5 addCall(CallOperationAction (OrderProcesing::ProcessCredit),PayCredit); L6 addForwardingCall(PayCredit,ProcessCredit);

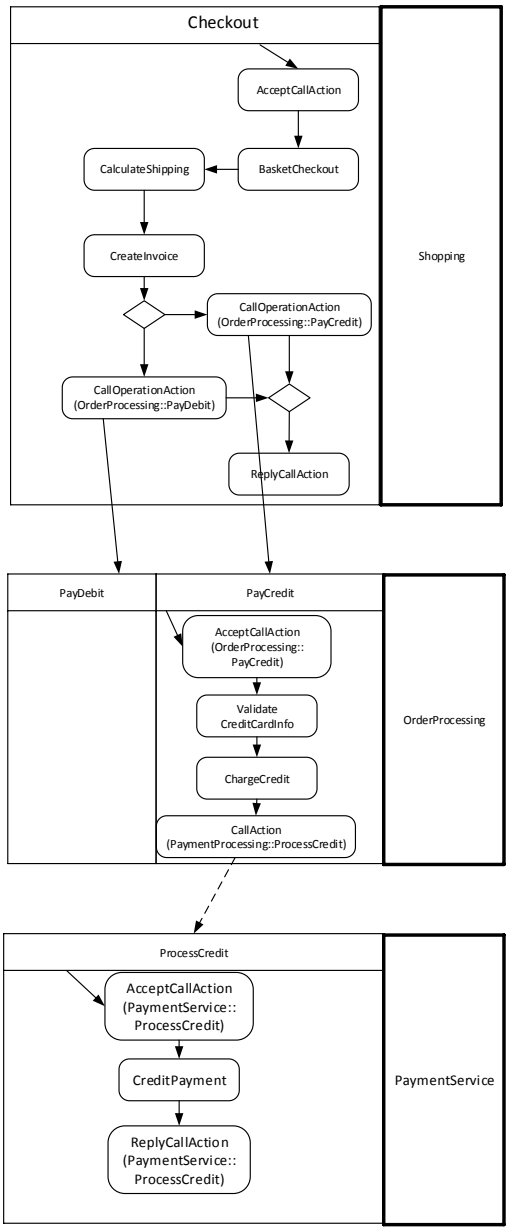

Figure 35: Partial LQN model after application of Service Call Back Design pattern

8.2.4 Redundant Implementation Pattern

It is not unusual for a system designer to create redundant instances of the critical services in the SOA to avoid a single point of failure or a bottleneck. Therefore, if one service is down or overloaded due to high demand (i.e. saturated task), then the redundant instance can help. Often this design pattern can be implemented with a load balancer a component which intelligently balances the load between the two duplicate instances. In this case study, it is assumed that system designer decided that the critical service for the SOA is Browsing and Shopping service and it should be implemented redundantly. To implement this in the SModel, a new swimlane is created in the BPM diagram for the redundant Browsing and Shopping service. Then the same activities which are inside the original Browsing and Shopping service are recreated in the newly created swimlane. Also, new calls are created from the user swimlane to the redundant Browsing and Shopping service. The existing call probabilities (given by MARTE annotations) are divided by half, meaning that the load is balanced equally between the original and redundant services. In the deployment diagram, a processing node is created with an artifact corresponding to the newly added swimlane in the BPM. The artifact is deployed to the processing node.

These changes are mapped to the following PModel changes: The new swimlane in the BPM is mapped to the creation of a new task and the new BPM activity leads to the creation of a new entry for that task. The deployment diagram processing node is mapped to a new processor in the PModel and the task is deployed to it. Also, new BPM calls are mapped to the calls of the activities of the User task in PModel with the divided probabilities. The performance results for this pattern are discussed in Section 8.3.4.

8.2.5 Partial State Deferral Pattern

Partial state deferral proposes that even when services are required to remain stateful, a subset of their state data can be temporarily deferred. To apply the pattern to the Browsing and Shopping SOA, the system designer decides that data for short-form specifications of products is cached at the service while the detailed specification is kept in the database. Therefore, fewer calls to the database give faster response time for the users. In the SModel, some activities are added to the Product swimlane to store and manage the popular short-form data locally. Also, the probability of calls to the DB swimlane is reduced (i.e. specified by MARTE annotation) as there will be less chance for the calls to the DB to retrieve the detailed specification of products. In PModel, few activities are added to the Product Catalogue entry of Product task. Also, the call probability from activities in Product Catalogue entry to DCT entry in DB task is reduced. The performance results for this pattern are discussed in Section 8.3.5.

8.2.6 Asynchronous Queuing Pattern

This patterns suggests that the wherever possible, a service capability should be used asynchronously by its consumers. In case of Browsing and Shopping SOA, it is assumed that the system designer decides that the Order Processing task can be used asynchronously by the Shopping service. This means that Order Processing entry does not wait for a reply from the Payment Processing entry as Order Processing entry forward the Shopping entry request to the Payment Processing task entries. The entries inside the Payment Processing task will issue replies directly to the Shopping entry, the initial requester. The implementation of this pattern is very similar to Service Callback Pattern discussed in Section 8.2.3. For implementing this design pattern in SModel, the

synchronous calls from the activities inside Order Processing to the Payment Processing swimlane and also from activities in the Shopping swimlane to Order Processing are both deleted. This includes the call from CallOperationAction to AcceptCallAction and the Reply call from ReplyAction to the caller, CallOperationAction. Also, the ReplyAction in the Order Processing swimlane needs to be removed as Order Processing does not provide replies anymore. Instead, a synchronous call from CallOperationAction in Shopping swimlane to the Order Processing (i.e. Sync Call from CallOperationAction to AcceptCallAction), and also an asynchronous call from Order Processing to Payment Processing and a Reply call from Payment Processing to Shopping processing are created. In the PModel, the ReplyAction activity inside the Order Processing task is deleted. A forwarding call is also created from the CallOperationAction inside the Order Processing to the entry of Payment Processing task.

8.2.7 Concurrent Contracts

Concurrent Contracts is one of the ways of implementing multi-channel user support. Using this design pattern, the system designer can create dedicated service capabilities (i.e. contracts) inside each service for each user category. In case the SOA needs to be modified to support a new group of users with specific processing needs, a new service capability can be created to fulfill the demand. The implementation of Concurrent Contracts in the SModel is done by creating a new activity subgraph inside the swimlanes. This is mapped to the creation of new entries with the activities inside the tasks in the PModel.

8.2.8 Event-Driven Messaging

In some systems a client application needs to learn from a service when a certain event has happened. Rather than calling the service repeatedly to check if the event happened or not, the consumer can subscribe to a notification service and the service will notify the consumer in case of an event. In SModel, activity subgraphs are created inside the swimlines to handle the subscription and also to send notifications in case of events. The consumers subscribe using asynchronous calls and the notifications are also sent via the asynchronous calls. In PModel, the entries are created inside tasks to accept asynchronous subscription requests. Also the entries make asynchronous calls through the activities to the consumers in case of an event.

8.2.9 User Interface Mediator

An SOA can sometimes enhance the user experience if it informs the user about the progress of the handling a time consuming request. But sending the progress messages from the components inside the SOA should not slow down their progress. This design pattern can be implemented in the SModel by adding actions inside the swimlane subgraphs to make asynchronous calls to a user interface mediator to update the mediator with the progress of a request. In the PModel, activities are created inside the entries to make asynchronous calls to the user interface mediator entry in the User task.

8.3 Using the Coupled Transformation for Performance Analysis

In this section, the coupled transformation technique is used in the performance analysis of the Shopping and Browsing SOA, when a system designer tries five design patterns to study their impact on its performance. The performance results are discussed and compared.

First, a performance analysis scenario which involves the application of the façade design pattern is discussed in Section 8.3.1. Then, some results for application of 1) Service Decomposition 2) Security Design Pattern 3) Redundant Implementation and 4) Partial State Deferral to the Shopping and Browsing SOA are discussed in Sections 8.3.2 to 8.3.5 and results are compared in Figure 36, Figure 37, and Figure 38. Each figure has the name of the pattern and the name of resource that the results are shown for at the bottom. The initial system for these experiments had a small difference from that shown in Figure 30, with the Shopping and Browsing task having a thread pool size of 10. A summary is also provided in Section 8.3.6.

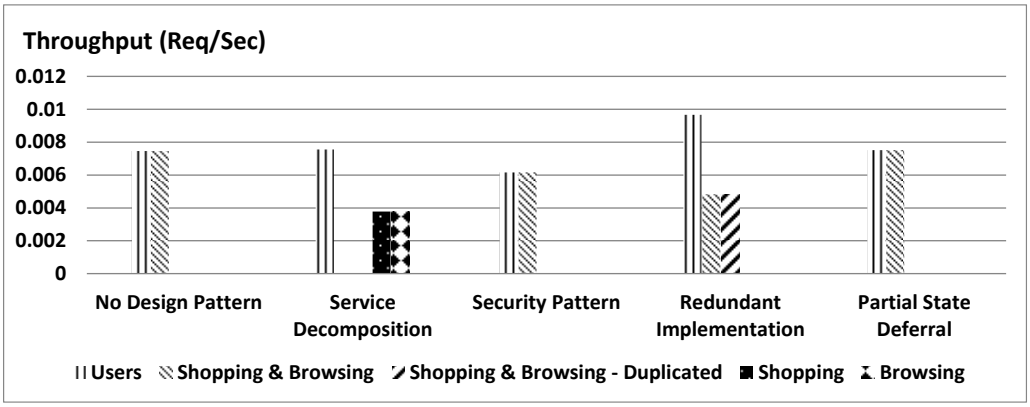

Figure 36: Throughput values based on 50 users for the four Design Patterns

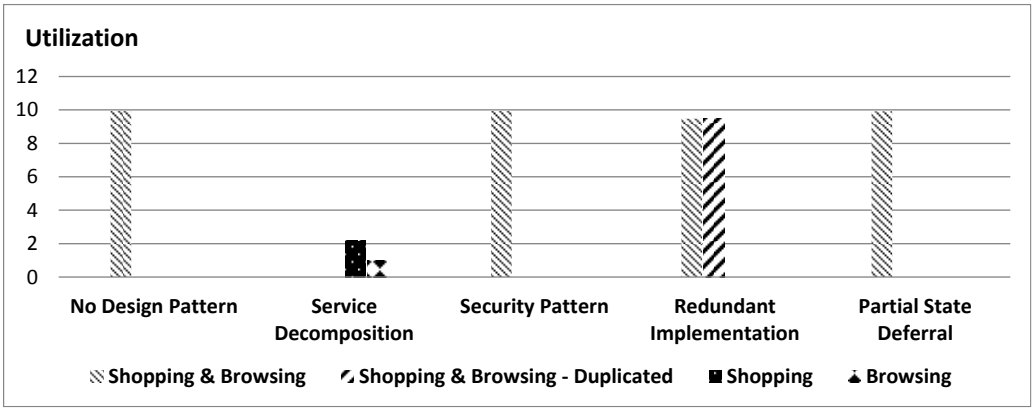

Figure 37: Utilization values based on 50 users for the four Design Patterns

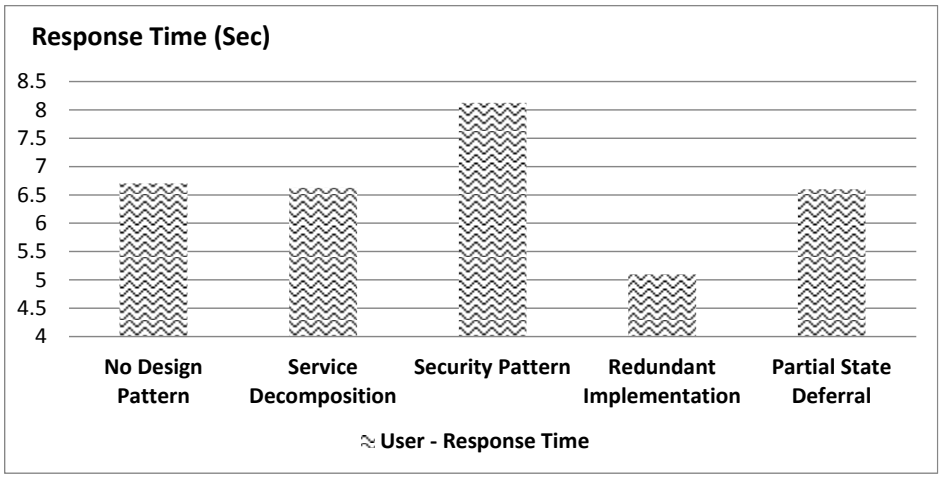

Figure 38: Response time value based on 50 users for the four Design Patterns

8.3.1 Façade Design Pattern

For the façade pattern applied as described in Chapter 3 , Chapter 5 , and Chapter 6 , with three groups of different types of users and four scenarios (named A, B, C and D), the system throughput and response times were determined for a range of user populations. For N users in User Group 1, there were 2N in User Group 2, and N/2 in User Group 3. N ranged from 2 to 220, so the total number of users ranged from 7 to 770.

In scenario A, the service façade pattern is not applied. Instead the multi-channel capability is implemented by providing separate shopping and browsing operations for each group giving the performance model in Figure 39. 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 25.

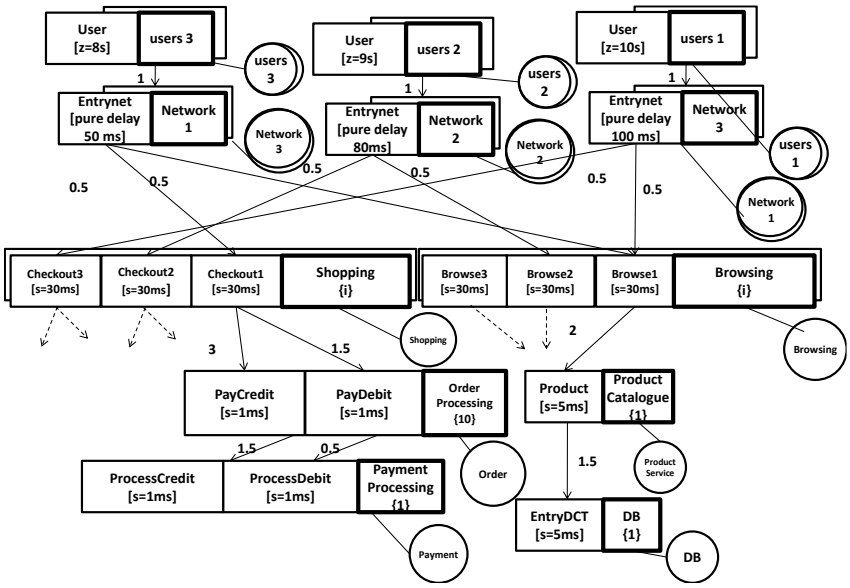

Figure 39: Performance Model after applying the multi-channel capability to each service without applying any Service Façade Design Pattern

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 40 compares the system throughputs and Figure 41 the response times of scenarios A and B for each user group of the system. Figure 40 and Figure 41 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.

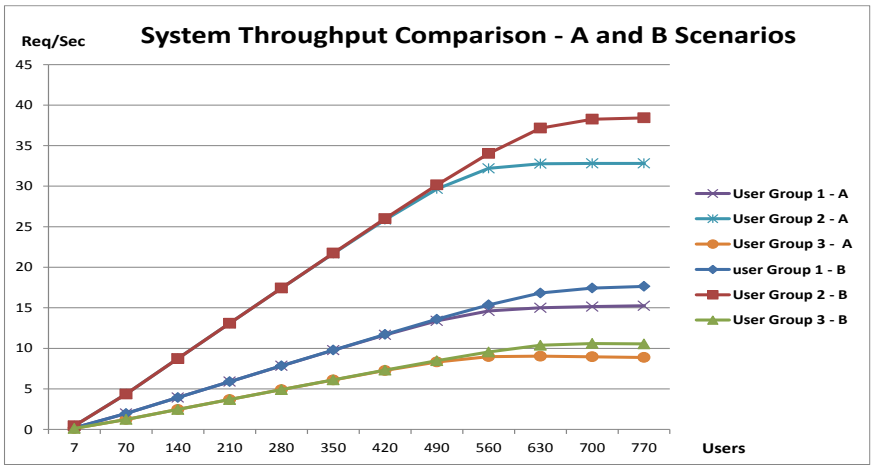

Figure 40: System Throughput for Scenarios A and B

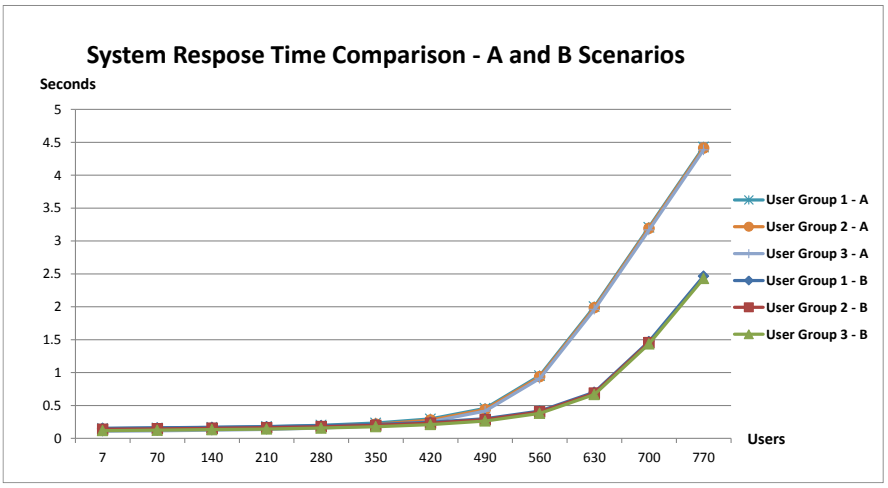

Figure 41: System Response Time for Scenarios A and B

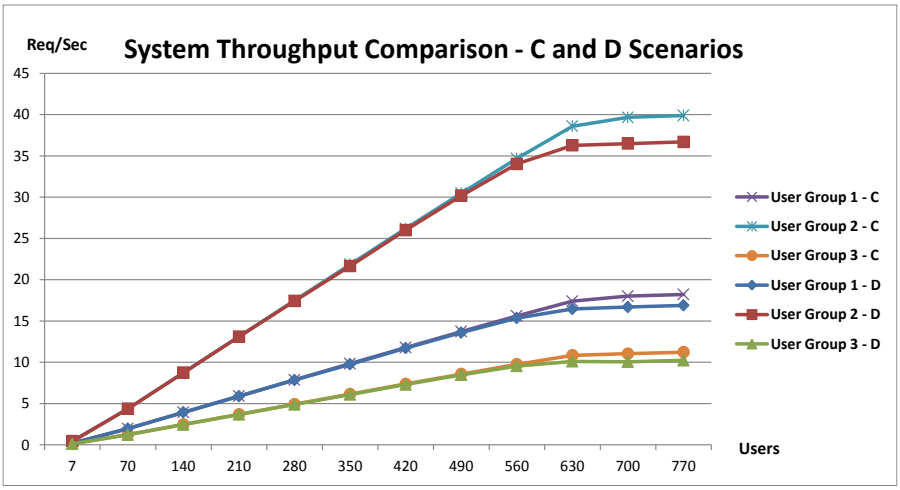

Figure 42: System Throughput for Scenarios C and D

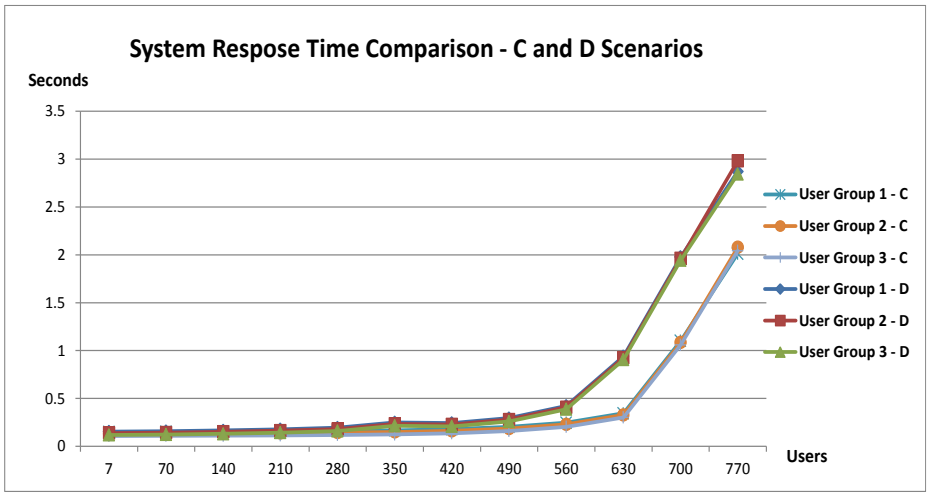

Figure 43: System Response Time for Scenarios C and D

Scenario C is obtained from A by changing the processors for the browsing and shopping services (which are the bottleneck for large populations) from single to dual-core. 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 an existing processor that is 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 42 and Figure 43.

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) is only slightly worse than scenario C (the modified version of A) even though now C and D are running on identical hardware resources. By exploiting our understanding of the resource utilizations and deploying the service façade in D on an existing but under-utilized processor the facade provides almost the same performance with the same resources. Thus we can 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).

8.3.2 Service Decomposition Pattern

The designer decision is to apply Service Decomposition (a popular SOA design pattern) and decompose this one service into two smaller services. To apply it, the designer must decide (1) whether the decomposed services are running on one host or two (2) the thread pool size of each decomposed task. The choices of one host, and 10 threads per task, gave the performance results shown in Figure 36, Figure 37, and Figure 38. There is a slight improvement in the system response time and utilization, due to the increased concurrency level (20 threads compared to 10 before). If the total thread pool size is kept at 10 (5 each), then no improvement is seen.

The SModel refactoring introduced a new ActivityPartition and a new service participant (along with its associated contract) to the SModel BPM and SEAM respectively. Also a new artifact for the second service is added to «GaExecHost» of the SModel deployment diagram. This leads to following PModel refactoring rules: (1) a new task is added for the second service, (2) one of the entries in the Shopping and Browsing task is moved to the newly added task, with its calls.

8.3.3 Security Design Pattern

Next we consider the impact of a security design pattern. The designer must choose the type of security technology, which here is the SSL encryption technology. The results after the SModel and PModel were refactored are shown in Figure 36, Figure 37, and

Figure 38. They show the reduction in performance caused by the additional workload of encryption.

To refactor the SModel, new activities were added to the BPM for the encryption and decryption (estimated by the designer, or found by experiment). These additional processing times increase the host demands in the PModel and give the performance degradation.

8.3.4 Redundant Implementation

A service which is actively in use introduces a potential single point of failure that may impair the reliability of the entire system. The Redundant Implementation design pattern addresses this issue with a failover support. To apply the pattern the Shopping & Browsing service in the system is duplicated and all the calls to this service are equally distributed between the original and the duplicate service. The performance results for normal operation are shown in Figure 36, Figure 37, and Figure 38. Figure 36 shows that the user throughput is improved by about 25% because of the additional service provider. Figure 37 shows that the utilization of the Shopping and & Browsing service is slightly improved in compare with before as it has less load on it in the new system. Figure 38 shows a 50% reduction in the response time. Furthermore, if one of the services fails the system still can serve the users which means increased reliability. But, all these improvements have the cost of adding and maintaining a second copy of the service.

8.3.5 Partial State Deferral

Services may be required to store and manage large amounts of state data, causing increased memory consumption, slow response, and reduced scalability. Partial state deferral proposes that even when services are required to remain stateful, a subset of their

state data can be temporarily deferred. In the present case, the Browsing service provides information for the users for browsing. To apply the pattern, data for popular products is cached at the service while the rest is being kept in the database. Fewer calls to the database give faster response time for the users. Figure 36, Figure 37, and Figure 38 show that the performance improvement obtained here is a small improvement only, and the impact on the utilization values is minimal.

8.3.6 Summary of the results

Overall, the approach proposed in the thesis was able to cope with a range of SOA design patterns. The performance evaluation by the PModel provides timely advice on the choice of the pattern and its alternatives. The traceability of the PModel changes provides insight into how the pattern affects the performance.

Chapter 9 Conclusions

This chapter summarizes the thesis contributions and discusses the limitations and directions for future work.

The thesis presents a successful coupled transformation approach for exploring the impact of SOA design patterns on the performance of a SOA system. It interprets a software pattern in terms of the corresponding changes in a performance model of the software by providing the following approaches:

1. RBML was tailored for modeling SOA design patterns (both problem and solution) in three views: structural, behavioral and deployment. RBML is essential as it formalizes and narrows down the pattern definition, and makes it concrete, with specific named artifacts to be added or changed, and in some cases it defines a specific variation of the pattern (such as the choice of SSL for security). This is essential for supporting the designer in specifying the SModel refactoring (useful in enforcing constraints on how the roles are bound). A role binding technique for the elements defined as roles and a derived binding technique for elements which are not playing any role were proposed for systematically identifying the SOA design problem and the model elements involved in the pattern application. Furthermore, a systematic approach was proposed for recording the SOA design changes in the form of SModel refactoring transformation rules.
2. The propagation of the SOA design refactoring operations to the corresponding performance model requires the knowledge of the cross-model relationships between the SModel and PModel elements. Therefore, a structure was proposed for the mapping table which holds the mapping between SModel and PModel elements. The

mapping table is generated from the traceability links produced in the process of deriving the LQN performance models from a SOA design. A mapping extension involving sets of UML Actions and Calls was used to overcome the semantic gap between the SModel and PModel.

3. The automatic propagation of the design pattern changes to the performance model is using a coupled transformation technique. According to this, the recorded SOA design refactoring rules and the mapping table are used for the derivation of the PModel refactoring transformation rules. Coupling the transformations ensures that the performance analysis remains in sync with the software changes, and relates the resource and performance changes back to the pattern.
4. For a generic PModel refactoring process that is able to propagate all kind of changes produced by a variety of design patterns, an LQN annotation technique was proposed to add transformation directives to the PModel elements. An extension to the LQN metamodel was used to provide Transformation Directives annotations that convey the PModel refactoring transformation rules. Then, a QVT-based transformation engine was designed and implemented to process the directives and to refactor the PModel. A performance model conforming to the regular LQN metamodel is constructed as output.
5. The following tools were designed and developed to support the transformation:
    - A tool for helping the system designer to record the SModel refactoring transformation rules. Some constrains are designed into the tool to help with the correctness and sanity of the recorded operations

- A tool for automatic derivation of PModel refactoring transformation rules from the recorded SModel refactoring transformation rules.
- A tool for automatic annotation of a PModel with derived PModel refactoring transformation directives.
- A QVT based transformation engine tool to process a PModel based on LQN+TD metamodel, refactor it and produce a regular LQN PModel.

The approach proposed in this thesis also supports successive applications of different patterns in combination, proceeding through a series of model versions. The technique is demonstrated and evaluated on the case study system, Browsing and Shopping SOA. The following steps were taken for validation and verification:

a. A set of unit test and test scenarios were designed to validate the methods in the two main functionality of the proposed approaches:
    i. Derivation of the PModel refactoring transformation rules from the recorded SModel refactoring transformation rules.
    ii. QVT-based refactoring of the LQN PModel by processing the automatically derived refactoring rules.

a. The proposed techniques were used in the application of ten SOA design patterns to the Browsing and Shopping SOA. The approaches from Chapter 3 and Chapter 4 were used for recording the SModel refactoring transformation rules and the methods described in Chapter 5 and Chapter 6 were used to derive the PModel refactoring transformation rules and to refactor the PModel.

b. The efficiency and effectiveness of the approaches proposed in the thesis were evaluated in the process of performance analysis process usually performed by the

system designer to evaluate the performance impact of 10 design pattern on Browsing and Shopping SOA. The performance results were generated using the LQN solver tool and compared to evaluate their impacts.

9.1 Limitations

Although the approaches developed in this thesis have achieved their goals, some limitations have been found due to the modeling features, model transformation capabilities and the usage of traceability links:

- In the SModel design using SoaML, the UML activity diagram is used to represents the business workflow model or BPM, which is one of the modeling views defining the platform independent model. The UML activity diagram is not the best choice for modeling business processes, as its metamodel does not support a wide range of business process meta-elements compared to other modeling languages, such as BPMN. However, advantages are that the UML activity diagram supports performance annotations (i.e. MARTE) and the fact that other views of the input models are represented using UML.
- Identification of the activity subgraph, synchronous calls and forwarding calls in the SModel are pretty much dependent on the consistent use of standard UML operational calls (i.e. three types of actions: *CallOperationAction, AcceptCallAction, ReplyAction*) in the construction of the UML activity diagram. At the time of writing this thesis, these types of actions are not very popular and there is limited tool support for drawing UML activity diagram with these types of actions.

- The approach in this thesis provides only the tools to the system designer for studying performance impact of a SOA design pattern on a SOA system. It does not support the choice of appropriate design patterns, which is left outside the scope of this work.
- The successive application of SOA design patterns to a SOA case study using the approach in this thesis requires that the SOA design also to be refactored to reflect the application of a pattern. The thesis assumes that the system designer performs such a refactoring manually or takes advantage of existing SOA design refactoring methods and tools to apply the design pattern changes for each iteration of the process.
- The approach in this thesis is dependent of the RBML notation for visual and systematic pattern specification technique. Without the RBML specification of SOA design patterns, it would be difficult to bind the pattern to the SModel. However the RBML is not considered as a fully formal approach for design pattern specification and this is one of the reasons that the process of recording SModel changes is not automated in this thesis. Formal pattern specification languages using a mathematical notation (e.g., see [73, 74, 113, 114]) provide the concepts needed to precisely describe design patterns and possibly automating the process of creating SModel refactoring transformation rules, but applying them requires sophisticated mathematical skills which a software engineer may lack.
- The approach in this thesis is dependent on the production of the traceability links from the initial process of generating the performance model from the SOA design for the mapping table, as one its key inputs. PUMA and PUMA4SOA are

capable of creating the traceability links and therefore the technique in this thesis is dependent on them. However, none of them creates the extended mapping table that we use in our approach.

- Although the approach in this thesis can be tailored to be used to refactor other types of performance models, at this moment it is only used in this research to generate an LQN performance model.

9.2 Future Work

Here is a list of future work directions for extending this research:

- The tool for recording the SModel refactoring transformation rules can potentially be enhanced to provide more assistance to the system designer in the process of sellecting the appropriate patterns, identifying where to apply a pattern, and refactoring the model.
- When the QVT languages implementation has been improved to support languages features such as reading files and enhanced list processing, the approach in this thesis can be enhanced to communicate the transformation directives to the QVT-based transformation engine through a list or file. This way, there will be no need for PModel annotation and the process of refactoring will be simpler.
- The approaches proposed in the thesis can be applied to other kinds of systems and patterns. Although it is designed to accept a SModel prepared with SoaML and to propagate SOA design pattern changes, the conceptual approach can be tailored to work on other types of software design and other type of software design patterns. This extended application could have widespread impact.

- The approach in this thesis was designed to only propagate the SOA design changes to a LQN performance model. But it can be enhanced to support other types of performance model notations such as Petri nets.

  Role Based Modelling Language (RBML) was used for visual and systematic presentation of SOA design patterns. The technique in this thesis can be adapted to support other modelling languages for design pattern specifications such as mathematical notations (e.g., see [73, 74, 113, 114]) or graphical constraints [85] .

References

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

[2] M. Kassab, G. E. Boussaidi, and H. Mili, "A quantitative evaluation of the impact of architectural patterns on quality requirements," *Springer's Studies in Computational Intelligence Book Series,* vol. 337, pp. 173-184, October 2011.

[3] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by Unified Model Analysis (PUMA)," *WOSP '05 Proceedings of the 5th international workshop on Software and performance*, Palma de Mallorca, Illes Balears, Spain, 2005, pp. 1 - 12

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

[5] N. Mani, D. C. Petriu, and M. Woodside, "Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture," Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Oulu, Finland, 2011.

[6] R. B. France, D.-K. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique," *IEEE Trans. Software Eng.,* vol. 30, pp. 193-206, 2004.

[7] D.-K. Kim, R. B. France, S. Ghosh, and E. Song, "Using Role-Based Modeling Language (RBML) to characterize model families," Proceedings. 8th IEEE Int. Conference on Engineering of Complex Computer Systems, 2002.

[8] Dae-Kyoo Kim, Robert France, S. Ghosh, and E. Song, "A role-based metamodeling approach to specifying design patterns," 27th Annual Int. Conferrence on Computer Software and Applications COMPSAC, 2003.

[9] Object Management Group, "Service oriented architecture Modeling Language (SoaML) " Version 1.0.1, formal/2012-05-10

[10] B. Elvesæter, C. Carrez, P. Mohagheghi, A. Berre, S. G. Johnsen, and A. Solberg, "Model-driven Service Engineering with SoaML," in *Service Engineering*, S. Dustdar and F.Li, Eds., Springer, 2011, pp. 25-54.

[11] Object Management Group, "A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems)," Version 1.1, formal/2011-06-02.

[12] Object Management Group, "Query/View/Transformation (QVT) " Version 1.2 ,formal/2015-02-01.

[13] Object Management Group, "Query/View/Transformation (QVT) URL : http://www.omg.org/spec/QVT/1.1/ [Last time accessed Feb, 2013]," Version 1.1, formal/January 2011.

[14] M. Woodside, D. Petriu, J. Merseguer, D. Petriu, and M. Alhaj, "Transformation challenges: from software models to performance models," *Software & Systems Modeling,* vol. 13, pp. 1529-1552, 2014.

[15] C. Trubiani, A. D. Marco, V. Cortellessa, N. Mani, and D. Petriu, "Exploring Synergies between Bottleneck Analysis and Performance Antipatterns," *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE'14)*, Dublin, Ireland, 2014.

[16] N. Mani, D. Petriu, and M. Woodside, "Propagation of Incremental Changes to Performance Model due to SOA Design Pattern Application," *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE'13) , Research Papers Track* Prague, Czech Republic, 2013, pp. 89-100.

[17] N. Mani, D. Petriu, and M. Woodside, "Towards Studying the Performance Effects of Design Patterns for Service Oriented Architecture," *Proceedings of the ICPE'11 Second Joint WOSP/SIPEW International Conference on Performance Engineering*, Karlsruhe, Germany, 2011.

[18] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*: Addison-Wesley Professional, 2011.

[19] E. Newcomer and G. Lomow, *Understanding SOA with Web Services*: Pearson Education, 2005.

[20] OASIS Ref Model, "Reference Model for Service Oriented Architectures, Committee Draft 1.0," February 7,2006 URL: http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf. [Last time accessed March , 2015].

[21] S. Durvasula, M. Guttmann, A. Kumar, J. Lamb, T. Mitchell, B. Oral, *et al.*, "SOA Practitioners," Guide Part 2 SOA Reference Architecture", available from http://www.soablueprint.com/whitepapers/SOAPGPart2.pdf, Jan 2011 [Last time accessed Feb 2015].

[22] Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*: Prentice Hall, 2001.

[23] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, *et al.*, *Patterns: Service-Oriented Architecture and Web Services*: IBM Redbooks, 2004.

[24] P. Allen, *Component-Based Development for Enterprise Systems*: Cambridge University Press, 1998.

[25] N. M. Josuttis, *SOA in Practice, The Art of Distributed System Design*: O'Reilly Media, 2007.

[26] 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.

[27] Cory Casanave (Model Driven Solutions), "Enterprise Service Oriented Architecture Using the OMG SoaML Standard URL : http://www.omg.org/news/whitepapers/EnterpriseSoaML.pdf [Last time accessed March, 2015]," December, 2009.

[28] D. C. Petriu, M. Alhaj, and R. Tawhid, "Software Performance Modeling," in *Formal Methods for Model-Driven Engineering*, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds., Springer Berlin Heidelberg, 2012.

[29] 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 March, 2015].

[30] Object Management Group, "Unified Modeling Language (UML)," Version V2.4.1,formal/2011-08-05

[31] Object Management Group, "Business Process Model and Notation (BPMN) Version 2.0," Version 2.0 formal/2011-01-03

[32] OASIS Standard, "Web Services Business Process Execution Language Version 2.0," 2007 URL:http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf [Last time accessed March, 2015].

[33] F. Taghizadeh and S. R. Taghizadeh, "A Graph Transformation-Based Approach for applying MDA to SOA," *International Conference on Frontier of Computer Science and Technology*, Shanghai, 2009, pp. 446 - 451.

[34] V. Rafe, R. Rafeh, P. Fakhri, and S. Zangaraki, "Using MDA for Developing SOA-Based Applications " *International Conference on Computer Technology and Development*, Kota Kinabalu, 2009, pp. 196 - 200.

[35] C.Emig, K. Krutz, S. Link, C. Momm, and S. Abeck, "Model-Driven Development of SOA Services," Cooperation & Management, Universität Karlsruhe (TH), Germany, 2007 URL: http://cm.tm.kit.edu/CM-Web/05.Publikationen/2007/[EK+07]_Model_Driven_Development_of_SOA_Services.pdf [Last time accessed March, 2015].

[36] C. Zhao, Z. Duan, and M. Zhang, "A Model-Driven Approach for Dynamic Web Service Composition," *WCSE '09 Proceedings of the 2009 WRI World Congress on Software Engineering*, 2009, pp. 273- 277.

[37] P. Kruchten, *The Rational Unified Process: An Introduction*: Addison Wesley, 2003.

[38] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, Oliver Laitenberger, R. Laqua*, et al.*, *Component-based Product Line Engineering with UML*: Addison Wesley, 2002.

[39] E. Ramollari, D. Dranidis1, and A. J. H. Simons2, "A Survey of Service Oriented Development Methodologies," *2nd European Young Researchers Workshop on Service Oriented Computing*, 2007.

[40] O. Zimmermann, P. Krogdahl, and C. Gee, "Elements of Service-Oriented Analysis and Design - An interdisciplinary modeling approach for SOA projects," Technical article, IBM (2 June 2004) URL: http://www.ibm.com/developerworks/library/ws-soad1/ [Last time accessed March ,2015].

[41] A. Arsanjani, "Service-Oriented Modeling and Architecture - How to identify, specify and realize services for your SOA.," Technical article, SOA and Web Services Center of Excellence, IBM, Software Group (9 November 2004). Online: http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/ [Last time accessed Feb ,2015].

[42] J. Floch, C. Carrez, P. Cieślak, M. Rój, R. Sanders, and M. M. Shiaa, "A comprehensive engineering framework for guaranteeing component compatibility," *Journal of Systems and Software & Systems Modeling,* vol. 83, 2010.

[43] 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 Feb ,2013].

[44] A. D. Marco and P. Inveradi, "Compositional Generation of Software Architecture Performance QN Models," *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2004, pp. 37-46.

[45] D. C. Petriu and X. Wang, "From UML description of high-level software architecture to LQN performance models " in *Applications of Graph Transformations with Industrial Relevance AGTIVE'99*. vol. 1779, A. S. Manfred Nagl, Manfred Münch, Ed., Springer Berlin Heidelberg, 2000, pp. 47-62.

[46] V. Cortellessa and R. Mirandola, "Deriving a Queueing Network based Performance Model from UML Diagrams," WOSP '00 Proceedings of the 2nd international workshop on Software and performance, 2000.

[47] Object Management Group, "UML Profile For Schedulability, Performance, and Time," v1.1, formal/2005-01-02.

[48] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*: Prentice Hall, 1984.

[49] D. C. Petriu, "Software Model based Performance Analysis," in *Model Driven Engineering for distributed Real-Time Systems: MARTE modelling, model transformations and their usages*, J. P. Babau, M. Blay-Fornarino, J. Champeau, S. Robert, and A.Sabetta, Eds., ISTE Ltd and John Wiley & Sons Inc., 2010.

[50] D. C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications," TOOLS

'02 Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, 2002.

[51] P. Haas, *Stochastic Petri Nets: Modelling, Stability, Simulation* Springer-Verlag, New York, 2002.

[52] R. German, *Performance Analysis of Communication Systems: Modeling with Non-Markovian Stochastic Petri Nets* John Wiley and Sons, 2000.

[53] S. Becker, H. Koziolek, and R. Reussner, "Model-Based performance prediction with the palladio component model," *Proceedings of the 6th international workshop on Software and performance*, Buenes Aires, Argentina, 2007, pp. 54-65.

[54] D. B. Petriu, "CSM2LQN – transformations for the generation of performance models from software designs"," PhD Thesis, Dept of Systems and Computer Engineering, Carleton University, Sept 2014.

[55] M. Alhaj and D. Petriu, "Using Aspects for Platform-Independent to Platform-Dependent Model Transformations," *International Journal of Electrical and Computer Systems,* vol. 1, pp. 35-48, 2012.

[56] M. Alhaj and D. C. Petriu, "Aspect-Oriented Modeling for Performance Evaluation with UML+MARTE, LQN, and CSM," *The Second International Comparing Modeling Approaches (CMA) workshop*, Innsbruck, Austria,, 2012.

[57] M. Alhaj and D. C. Petriu, "Aspect-oriented Modeling of Platforms in Software and Performance Models," *the International Conference on Electrical and Computer Engineering ICECS' 2012*, Ottawa, Canada, 2012.

[58] M. Alhaj and D. Petriu, "Traceability Links in Model Transformations between Software and Performance Models," in *SDL 2013: Model-Driven Dependability Engineering*. vol. 7916, F. Khendek, M. Toeroe, A. Gherbi, and R. Reed, Eds., Springer, 2013, pp. 203-221.

[59] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz, "Layered Queueing Network Solver and Simulator User Manual Revision: 11145 URL:http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf [Last time accessed March, 2015]," Feb 2012.

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

[61] V. Cortellessa, A. D. Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into UML models to remove performance antipatterns," *Proceeding of ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, Cape Town, 2010, pp. 9-16

[62] D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, Bertinoro, Italy, 2012.

[63] T. Parsons and J. Murphy, "Detecting Performance Antipatterns in Component Based Enterprise Systems," *Journal of Object Technology,* vol. 7, 2008.

[64] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex, and J. P. Sousa, "A framework for utility-based service oriented design in SASSY," WOSP/SIPEW '10

Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering, 2010.

[65] J. Xu, "Rule-based automatic software performance diagnosis and improvement," *Proceeding of 7th Intl Workshop on Software and Performance*, Princeton, NJ, USA, 2008, pp. 1-12.

[66] I. Galvao and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineerin," *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, 2007, p. 313.

[67] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, " On-Demand Merging of Traceability Links with Models," *3 rd ECMDA Traceability Workshop*, 2006.

[68] A. v. Knethen and B. Peach, "A Survey on Tracing Approaches in Practice and Research," Technical Report IESE Report Nr. 095.01/E, Fraunhofer - Institute of Experimental Software Engineering, 2002.

[69] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns: Pattern-Oriented Software Architecture*: Wiley, 1996.

[70] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley, 1995.

[71] W. Pree, *Design Patterns for Object-Oriented Software Development*: Addison Wesley, 1995.

[72] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Oriented Software Architecture: Patterns for Concurrent and Networked Objects*: Wiley, 2000.

[73] A. H. Eden, A. Yehudai, and J. Y. Gil, "Precise specification and automatic application of design patterns," ASE '97 Proceedings of the 12th international conference on Automated software engineering, Incline Village, NV, 1997.

[74] K. Lano, J. C. Bicarregui, and S. Goldsack, "Formalising Design Patterns," Proceeding of BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science, 1996.

[75] V. Cortellessa, A. D. Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into UML models to remove performance antipatterns," Proceeding of 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems, Cape Town, 2012.

[76] C. W. Bachman and M. Daya, "The role concept in data models," he third international conference on Very large data bases, 1977.

[77] L. A. Stein and S. B. Zdonik, "Clovers: The Dynamic Behavior of Types and Instances," Technical Report CS-89-42, Department of Computer Science, Brown University, Providence, RI,Nov. 1, 1989.

[78] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," *Data Knowl. Eng.,* vol. 35, pp. 83-106, 2000.

[79] A. Albano, G. Ghelli, and R. Orsini, "Galileo: A strongly-typed, interactive conceptual language," *In ACM Transactions on Database Systems,,* vol. 10, 1985.

[80] B. Rock, G. Gottlob, and S. Michael, "Extending Object - Oriented Systems with Roles," *ACM Transactions on Information Systems,* vol. 14, pp. 268-296, 1996.

[81] B. Pernici, "Objects with roles," *In Proceedings of the conference on Office information systems*, Cambridge, Massachusetts, 1991, pp. 25-27.

[82] F. Steimann, "On the representation of roles in objectoriented and conceptual modelling," *In Data and Knowledge Engineering,* vol. 35, pp. 83-106, 2000.

[83] E. Sciore, "Object Specilazation," *In ACM Transactions on Information Systems,* vol. 7, pp. 103-122, 1989.

[84] R.Wieringa and W. D. Jonge, "The identification of objects and roles: Object identifier revisited," Technical Report lR-267, Vrije University, Amsterdam,1991.

[85] A. Lauder and S. Ken, "Precise Visual Specification of Design Patterns," *Proc. 12th European Conf. Object-Oriented Programming*, 1998, pp. 114-136.

[86] G. E. Boussaidi and H. Mili, "Understanding design patterns - what is the problem?," *Journal of Software: Practice and Experience,* vol. 42, pp. 1495–1529, December 2012.

[87] A. L. Guennec, G. Sunye, and J. Jezequel, "Precise Modeling of Design Pattern," *Proc. Third Int'l Conf. Unified Modeling Language*, 2000, pp. 482-496.

[88] M. Fowler, *Refactoring: improving the design of existing code*: Addison-Wesley Professional, 1999.

[89] A. Ghannem, G. E. Boussaidi, and M. Kessentini, "Model Refactoring Using Examples: A Search-based Approach," *The Journal of Software: Evolution and Process,* vol. 26, pp. 692–713, July 2014.

[90] M. Woodside, D. C. Petriu, and K. H. Siddiqui, "Performance-related Completions for Software Specifications," *In Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002*, Orlando, Florida, USA, 2002, pp. 22-32.

[91] D. C. Petriu, C. M. Woodside, D. B. Petriu, J. Xu, T. Israr, G. Georg, *et al.*, "Performance Analysis of Security Aspects in UML Models," *Proceedings of the 6th International Workshop on Software and Performance, WOSP 2007*, Buenes Aires, Argentina, 2007, pp. 91–102.

[92] M. Woodside, D. C. Petriu, D. B. Petriu, J. Xu, T. Israr, G. Georg, *et al.*, "Performance analysis of security aspects by weaving scenarios extracted from UML models," *Journal of Systems and Software,* vol. 82, pp. 56-74, 2009.

[93] eclipse, "Model-to-Model Transformation (MMT) " Version 3.2.0 2012-06-27 URL : http://projects.eclipse.org/projects/modeling.mmt.qvt-oml [Last time accessed March, 2015].

[94] Object Management Group, "Object Constraint Language (OCL)," Version 2.4 formal/2014-02-03.

[95] Object Management Group, "Meta Object Facility (MOF™) Core," Version 2.4.2 formal/2014-04-03.

[96] H. Þ. Einarsson, "Refactoring UML Diagrams and Models with Model-to-Model Transformations, A M.Sc. thesis," M.Sc., Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland, 2011.

[97] J. Dong, S. Yang, and K. Zhang, "A Model Transformation Approach for Design Pattern Evolutions," ECBS '06 Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006.

[98] J. Dong, Y. Zhao, and Y. Sun, "Design pattern evolutions in QVT," *Software Quality Control,* vol. 18, pp. 269-297, June 2010

[99] J. Dong, S. Yang, Y. Sun, and W. E. Wong, "QVT based model transformation for design pattern evolutions," the Tenth IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA 2006), Honolulu, Hawaii, US, 2006.

[100] T. Baar and S. a. Markovi, "A graphical approach to prove the semantic preservation of UML/OCL refactoring rules.," the 6th international Andrei Ershov memorial conference on Perspectives of systems informatics, Berlin, Heidelberg, 2007.

[101] M. Elaasar, L. Briand, and Y. Labiche, "VPML: an approach to detect design patterns of MOF-based modeling languages," *Software & Systems Modeling,* pp. 1-30, 2013/03/03 2013.

[102] S. Becker, "Coupled Model Transformations," *Proceedings of the 7th international workshop on Software and performance*, Darmstadt, Germany, 2008, pp. 103-114

[103] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice," vol. 17, pp. 309-332, 2005.

[104] A.Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," *12th International IEEE Conference on Enterprise Distributed Object Computing Conference, EDOC'08*, 2008, pp. 222-231.

[105] A.Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Managing Dependent Changes in Coupled Evolution," in *Theory and Practice of Model Transformations*, R. F. Paige, Ed., Springer Berlin Heidelberg, 2009.

[106] M. Alhaj, "Automatic generation of performance models for SOA systems," *Proceedings of the 16th International Workshop on Component-Oriented Programming (WCOP 2011)*, 2011, pp. 33 - 40.

[107] M. Alhaj and D. C. Petriu, "Approach for generating performance models from UML models of SOA systems," *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2010)*, 2010, pp. 268–282.

[108] Telecommunication Standardization Sector of ITU, "User Requirement Notation (URN) standard," Z-151 2012-10 URL: https://www.itu.int/rec/T-REC-Z.151-201210-I/en [Last time accessed March,2015].

[109] Eclipse, "Papyrus graphical editing tool for UML2 as defined by OMG," Version 1.0.0 , http://www.eclipse.org/papyrus/ [Last time accessed March ,2015].

[110] eclipse, "MagicDraw UML modelling tool by No Magic," http://www.nomagic.com/products/magicdraw.html [Last time accessed March ,2015].

[111] eclipse, "Eclipse Modeling Tools - Kepler Packages," http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/lunasr2 [Last time accessed March,2015].

[112] eclipse, "QVT Operational Component," http://projects.eclipse.org/projects/modeling.mmt.qvt-oml [Last time accessed March ,2015].

[113] T. Mikkonen, "Formalizing Design Patterns," *Proceedings 20th International Conference of Software Engineering*, 1998, pp. 115-124.

[114] L. Lamport, "The Temporal Logic of Actions," *ACM Trans. Programming Languages and Systems,* vol. 16, pp. 872-923, 1994.