Automatic Derivation of Performance Models in the Context of Model-Driven SOA

by

Mohammad Alhaj

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

Ottawa Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

January 2014

Abstract

The thesis proposes a model transformation chain called Performance from Unified Modeling Analysis for Service-Oriented Architecture (SOA) systems (PUMA4SOA), whose purpose is to automatically generate performance models from the UML software design models of SOA systems with performance annotations. The main goal of PUMA4SOA is to enable the analysis of performance properties of software systems in the early software development phases, which helps developing SOA systems that meet their performance requirements. PUMA4SOA extends PUMA, an existing transformation approach from software to performance models developed in our research group. The main differences between PUMA4SOA and PUMA are as follows: a) focus on SOA systems; b) application of Model-Driven Architecture (MDA) principles of considering first software platform-independent models (PIM) which are then transformed into platform-specific models (PSM); c) use of a Platform Completion (PC) feature model to define variability of platform characteristics; d) use of aspect-oriented modeling (AOM) techniques to specify realization of platform features; and e) systematic use of trace-links between different types of models (i.e., software, intermediate and performance models). PUMA4SOA accepts the following input models: the software platform independent model, the deployment model, the PC-feature models and a set of platform aspect models. Similar to PUMA, PUMA4SOA makes use of an intermediate model called Core Scenario Model (CSM). The model transformations chain of PUMA4SOA begins by transforming the UML PIM to a CSM PIM, which in turn is used to generate a CSM PSM using an AOM approach. The third model transformation maps the CSM platform specific model into a performance model (Layered Queuing Network in this case) which is then solved to produce the performance output results.

A traceability model is used in PUMA4SOA to navigate between different kinds of models (i.e., UML, CSM and LQN) in order to propagate changes of properties from one model to another and to feed back the LQN output results into the UML design model.

Acknowledgements

I would like to thank Prof. Dorina Petriu, whose valuable supervision helped me very much and kept the research on track. I would also like to thank my family for their support, my wife Farehan and my kids Aseel, Hanin, Abdullah and Nada, my dear father, my brother and sisters and all the people who were involved in the research.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AOM	Aspect-Oriented Modeling
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CORBA	Common Object Request Broker Architecture
CSM	Core Scenario Model
DCOM	Distributed Component Object Model
EMF	Eclipse Modeling Framework
GQAM	Generic Quantitative Analysis Modeling
GRM	General Resource domain model
GSPN	Generalized Stochastic Petri net
FODA	Feature Oriented Domain Analysis
LQN	Layered Queuing Model
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MDA	Model Driven Architecture
MD-SOA	Model Driven SOA
MDD	Model Driven Development
MOF	Meta-Object Facility
NFP	Non-Functional Property
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model

PUMA	Performance from Unified Model Analysis
PUMA4SOA	Performance from Unified Model Analysis for SOA
REST	Representational State Transfer
SOA	Service-Oriented Architecture
SoaML	Service-oriented Architecture Modeling Language
SOAP	Simple Object Access Protocol
SPT	Schedulability, Performance and Time
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
WSDL	Web Service Definition Language

Chapter 1: Introduction

The chapter presents the context, motivation, objectives and scope of this work, as well as the thesis contributions.

Service-Oriented Architecture (SOA) is a software development paradigm for building and deploying software applications as a set of reusable services [EAR05]. Business processes are realized by composing services through orchestration and/or choreography. Service orchestration is used within an individual business organization to manage the workflow of the invoked services, while service choreography is used to manage the collaboration of services between multiple business organizations. Several modeling languages are used to represent service orchestration and choreography, such as the Business Process Execution Language (BPEL) [OAS07], the Business Process Model and Notation (BPMN) [OMG11] and UML activity diagrams [OMG09]. Designing the service architecture within SOA requires not only describing the functional capabilities and the classification of services but also the service dependencies and contracts between consumers and providers. Service-oriented architecture Modeling Language (SoaML), one of the few languages to model the service architecture is defined as a UML profile which extends UML 2.0 for SOA [OMG09b].

SOA systems can be realized using different middleware technologies such as CORBA, Web services and RESTful services. Common Object Request Broker Architecture (CORBA) is an OMG standard that allows multiple software components written in different languages running on multiple heterogeneous computers to communicate [OMG03a, ROB05]. Web services represent an approach that uses composable services to build distributed applications in a distributed heterogeneous system. Web services technology rely on different XML-based

languages, such as WSDL, which describes how the service can be called; SOAP, which is used as a message protocol between the service consumer and provider; and UDDI, which is used for publishing and discovering information about web services in the registries [OAS02, ALM08]. REST is a distributed system framework that retrieves information from the web sites by defining the source of specific information as a resource and addressing each resource with a URI [RIC07, SCH11]. In this research, we are using web services technology to realize SOA systems. Web services technology is well-defined, standardized, widely used, allows for code reuse and can be easily implemented, since several tools perform code to web service conversion.

Model-Driven SOA (MDSOA) extends Model Driven Development (MDD) to the development of service-oriented systems using models at different levels of abstractions that describe different views of SOA systems, such as the business process view, service structure view, and service behavior view. Model transformation is then applied to refine the design models and to generate either code or other models for the analysis of non-functional properties, such as performance. A major step in MDSOA is the separation between the platform independent model (PIM) and the platform specific model (PSM) of SOA systems. From a performance analysis point of view, this separation helps designers to analyze the performance of the platform independent models, considering that they can be deployed on different platforms.

Aspect Oriented Modeling (AOM) allows software designers to address separately solutions for crosscutting concerns. We use AOM for modeling service platform operations and for transforming platform independent models (PIM) to platform specific models (PSM). We describe platform variability with the help of a Performance Completion (PC) feature model, which allows the modeler to select the desired platform features before performing the aspect composition.

The Unified Modeling Language (UML) is a well-known modeling language for representing software systems during the development phase [OMG09]. UML2.0 provides a graphical notation including 14 kinds of diagrams to create visual models of software systems supporting multiple views. The diagrams are categorized into two groups: structure diagrams such as class and deployment diagram, and behaviour diagrams such as interaction and activity diagram. UML also supports a standard extension mechanism by defining profiles which extends the UML models for different domains. OMG has standardized several profiles for real-time domain, for SOA, for testing and more.

The evaluation of non-functional properties (NFP) (such as performance, schedulability, dependability, security) of software and system designs can be achieved by extending the UML software model with an appropriate profile to add information specific to the property to be evaluated and then transforming the annotated UML model into an analysis model for the respective NFP, which can be analyzed with existing tools [BAL04, WOO05]. There are two standard profiles for annotating the performance properties of UML models: the UML Profile for Schedulability, Performance and Time (SPT) [OMG05] defined for UML 1.X and the UML Performance Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [OMG11a] defined for UML 2.X.

Traceability is a software approach to establishing relationships between various software artifacts during the software development cycle. A traceability model is used during model transformation to create trace-links between the source and target model elements, which can be used to define dependencies between related elements in different models, to propagate changes of properties from one model to another and to analyze the impact of these changes [ALH13].

1.1 Motivation

Service-oriented Architecture is an innovative paradigm which has become a powerful technique for building enterprise applications in the last decade. SOA allows organizations to develop business applications as a set of business processes which invoke reusable services. Several design principles are addressed during the development of SOA, such as reusability, composability, modularity, granularity and componentization [MAR08], which lead to a reducing the time and cost of the software development. A disadvantage is that composing services at runtime introduces a considerable overhead. Hence, SOA systems have relatively poor performance compared with business applications using regular function calls. This may be a problem, especially when SOA systems use services with a fine granularity, which introduce a lot of middleware overhead between the service consumers and the service providers. In our research, we build performance models of SOA systems that include the performance contributions of both the application and the underlying platforms. The goal is to help developers understand the performance effects of their design choices and to build systems that meet their performance requirements. For instance, performance analysis can help developers define the best level of service granularity needed to meet the performance requirements and at the same time maintain the SOA design principles.

Performance from Unified Model Analysis (PUMA) is an ongoing research project in the Department of System and Computer Engineering at Carleton University [WOO2005, WOO13]. The goal of PUMA is to provide model transformations from different kinds of software design models (e.g., UML, UCM) to different kinds of performance models (e.g., queuing networks, Petri nets, simulation) [WOO05, WOO13, XU03]. In order to reduce the number of transformations from $N*M$ to $M+N$ (where M is the number of input software models and N the

number of generated performance models), PUMA uses an intermediate model called *Core Scenario Model* (CSM). From a semantics point of view, CSM is situated between the software models and the performance models, so it reduces the complexity of transforming UML diagrams extended with SPT or MARTE annotations directly into various types of performance models. The design model that is input to PUMA defines two modeling views: structural (including deployment of software to hardware) and behavioral.

In order to analyze the performance of SOA systems, this thesis introduces PUMA4SOA which extends PUMA with new features that consider the nature of SOA systems. As an example, additional levels of abstraction have been defined to model the SOA business processes and the service architecture. Also, we separate the SOA design model from the service platform characteristic. For this purpose, we use two levels of abstraction: a *platform independent model* (PIM), and a *platform specific model* (PSM), as in OMG's MDA [OMG03b]. We address the separation between PIM and PSM by using *aspect oriented modeling* (AOM). In our approach, PIM is considered as the *primary model* that has to be composed with a set of aspect models realizing different features of the SOA platform. After selecting the desired features based on the business requirements from a so-called *performance completion (PC) feature model* and getting the corresponding *platform aspect models* that realize the selected features, a PSM is generated by weaving these aspect models into the PIM. The PC feature model is used to define the variability in platform choices, types of platform realizations, and other external factors that have an impact on the system's performance [ALH13].

The goal of PUMA4SOA is to allow the developers to analyze the performance of SOA systems in the early development phases and to help them make good design decisions regarding the system architecture, design, and configuration [ALH10].

In order to manage the model changes in PUMA4SOA, a traceability model is created to provide trace-links between different kinds of models used in PUMA4SOA (i.e., software, intermediate and performance models). During a given model transformation (e.g., software to intermediate model or intermediate to performance model), trace-links are created that connect the elements of the source and the target models without affecting them or their metamodels. This provides PUMA4SOA the capability to define dependencies between related elements in different kinds of models, to maintain the models consistency, to propagate changes of properties (such as performance parameters or results) from one model to another and to analyze the impact of these changes.

1.2 Thesis Objectives and Scope

The thesis contributes to the generation of performance models from SOA software design models, which can be used to evaluate the runtime performance characteristics of a SOA system in the early phases of the development process. The early performance evaluation helps to choose an appropriate architecture, design, and configuration alternatives to ensure that the final system meets its performance requirements. The research extends the existing Performance from Unified Model Analysis (PUMA) approach in order to address the performance analysis of SOA systems and to separate concerns of the service platform from the SOA design model. The following types of models are used in the thesis: UML software models extended with two profiles, SoaML [OMG09b] and MARTE [OMG11a], CSM intermediate model [PET04, PET07] and Layered Queuing Network (LQN) performance model [FRA12, WOO13].

From the beginning, the focus of this thesis was on the transformation from an annotated UML software model to CSM, while the transformation from CSM to LQN performance model was the concern of a different thesis. However, we realized that in some cases it was necessary

to specify some aspects of the transformation from CSM to LQN and its traceability model, in order to explain how specific SOA characteristics are handled. Therefore, all the transformations and related trace-links from UML to CSM are designed, implemented and tested in the thesis, while those between CSM-and-LQN and LQN-and-UML are only specified, but not implemented.

The input SOA design model to PUMA4SOA is composed, in fact, of three models: a) the PIM representing the workflows, architecture of the underlying components offering services, and service behavior; b) the deployment model describing the allocation of software to hardware; and c) the platform aspect models describing features of the service platform in a generic form. The input design models are described using the UML language [OMG09a] and SoaML profile [OMG09b], while the performance annotations are given with the MARTE profile [OMG11a].

In PUMA4SOA we use a Performance Completion (PC) feature model to capture the variability in platform choices, different types of service realizations, and other external factors with an impact on performance and to represent the dependencies and relationships between them. The PC feature model represents a classification of the different SOA platform aspects and of their relationships in a hierarchical format [KAN90]. The modeler can select from the PC feature model multiple SOA platform aspects needed for the application according to the business requirements.

In our approach, we propose a model transformation chain which begins with an automatic model transformation of the input design models into separate CSM models. Next, an automatic model transformation which derives the CSM PSM by weaving the CSM platform aspect models into the CSM PIM using an aspect oriented modeling approach (AOM). The third

model transformation generates the LQN performance model from the CSM platform specific model. The performance output is produced by executing the LQN model, using an existing LQN solver. The LQN model can be used for different types of performance analysis, such as the pass/fail, sensitivity analysis and finding optimal values.

During the model transformation chain of PUMA4SOA, a traceability model that defines the trace-links between the UML, CSM and LQN models is automatically generated. The traceability approach is based on a traceability metamodel, where classes are used to define the trace-links types. The trace-links are applied in three groups: a) between the elements of the UML and the CSM models, b) between the elements of the CSM and the LQN models and c) between the elements of the LQN and the UML. Trace-links are used in PUMA4SOA to propagate the changes between the different models, to analyze the impact of those changes on the performance of the SOA system and to feed back the performance results obtained by executing the LQN model, to the UML input design models.

1.3 Thesis Contributions

The contributions of this thesis are as follows:

- Develop (i.e., design, implement and test) an automatic model transformation from the Platform Independent Models (PIM) and the platform aspect models of the UML input design models into separate CSM models. This is preceded by selecting the desired features from the Performance Completion (PC) feature model.
- Develop an automatic model transformation from the CSM platform independent model into CSM platform specific model using an Aspect-Oriented Modeling (AOM) approach. This includes performing the AOM steps for multiple aspect compositions.

- Specify in QVT the model transformation from the CSM platform specific model into the LQN performance model to explain the differences from PUMA, such as handling workflows, services and platform elements. (This transformation is not implemented).
- Define the traceability metamodel for PUMA4SOA, used in turn to instantiate the traceability models between UML-and-CSM, CSM-and-LQN and LQN-and-UML. The traceability model between UML-and-CSM was implemented and tested, while the other two were only specified.
- Instantiate a traceability model for a given SOA model, which includes the creation and initialization of trace-links between the elements of UML and CSM models. The instantiation of trace-links between the elements of CSM and LQN and between LQN and UML are only specified.

It is important to mention that the approaches for aspect-based platform modeling, transformation from PIM to PSM and trace-links are applied in PUMA4SOA to service-based systems. However, the concepts for these approaches are not specific to service-based system and can be applied to other classes of software.

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

- M. Alhaj, D. C. Petriu, “Approach for generating performance models from UML models of SOA systems”, CASCON '10 Proceedings of the 2010 Conference of the IBM Center for Advanced Studies on Collaborative Research, 2010.
- M. Alhaj, “Automatic Generation of Performance Models for SOA Systems”, WCOP '11: Proceedings of the 16th International Workshop on Component-oriented Programming, 2011.

- D. C. Petriu, M. Alhaj, R. Tawhid, "Software Performance Modeling", Formal Methods for Model-Driven Engineering, Lecture Notes in Computer Science Volume 7320, 2012, pp. 219-262.
- M. Alhaj, D. Petriu, "Aspect-oriented Modeling of Platforms in Software and Performance Models", In proceeding of: International Conference on Electrical and Computer Systems (ICECS'12), August 2012.
- M. Alhaj, D. Petriu, "Using Aspects for Platform-Independent to Platform-Dependent Model Transformations", International Journal of Electrical and Computer Systems, Volume 1, Issue 1, Year 2012.
- M. Alhaj, D. C. Petriu, "Traceability Links in Model Transformations between Software and Performance Models", SDL 2013: Model-Driven Dependability Engineering, Lecture Notes in Computer Science Volume 7916, 2013, pp. 203-221.
- M. Woodside, D. C. Petriu, J. Merseguer, D. B. Petriu, M. Alhaj, "Transformation challenges: from software models to performance models", Software and System Modeling, (DOI: 10.1007/s10270-013-0385-x); Springer, (published online), 2013.

1.4 Contents of the Thesis

This subsection describes the overall organization of the thesis and the content of each chapter.

Chapter 2: Background and State of the Art. The first section presents a SOA overview; the second section the modeling languages used in PUMA4SOA; the third section the aspect-oriented modeling techniques; and the fourth section an overview on traceability concepts. The final section presents the state of the art, discussing related works in generating performance

models for SOA systems. A group of selected MDA approaches are described and compared to PUMA4SOA.

Chapter 3: PUMA4SOA Approach and Case Study. The first section presents the Performance from Unified Modeling for SOA (PUMA4SOA) approach and its new features. The second section presents the source model for PUMA4SOA based on a purchase order system case study. The third section presents the platform variability using the PC feature model and the final section presents three kinds of platform aspect models: service invocation, service publishing and service discovery.

Chapter 4: Model transformations: The first section discusses the QVT specification. The second section presents the model transformation from the UML design model to the LQN performance model; this includes transformations from UML to CSM, MARTE stereotype to CSM and from CSM to LQN. It also emphasizes the differences of model transformations between PUMA and PUMA4SOA.

Chapter 5: Generating Platform Specific Model using AOM approach. The first section presents a case study of generating a platform specific model using AOM approach. It describes the steps of AOM modeling for SOA platform at three levels (UML, CSM, and LQN). The second section presents the specifications and techniques used in the model transformation from PIM to PSM at the CSM level using aspect composition.

Chapter 6: Traceability modeling. The first section presents the traceability metamodel of PUMA4SOA, which defines three groups of trace-links: UML-and-CSM, CSM-and-LQN, and LQN-and-UML. The second section presents the trace-links related to aspect models. The third section presents the specifications for applying trace-links in PUMA4SOA. The final section presents a traceability model for the purchase order system case study.

Chapter7: Case studies for testing and validation. The first section presents the testing of PUMA4SOA model transformation using a list of case studies. The second section presents the testing of the model transformation using a case study of an Electronic Prescription System. The final section presents the validation of PUMA4SOA performance analysis using an Insurance Broker case study.

Chapter 8: Conclusions. The conclusions are first presented, then the scope and the limitations of the proposed approach. The final section presents directions for future work.

Chapter 2: Background and State of the Art

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

2.1 Service-Oriented Architecture (SOA) overview

Service-Oriented Architecture (SOA) is a software development paradigm that uses services to develop and deploy business applications. It is an integrated software and design approach that delivers business functions as shared and reusable services [SER13]. SOA builds applications using software services that communicate with each other by either passing simple data or involving two or more services coordinating some activity. Designing SOA systems rely on wide principle rules [ERL08], such as service abstraction, service loose coupling, service reusability, service automation, service stateless, service discoverability, service composability and service contracts.

SOA architecture consists of three elements [DUR11]: 1) the Business Architecture, which defines the business goals, the business objectives and the business workflows, 2) the Infrastructure Architecture, which describes the components that provide the business capabilities, the structure and the behaviours of the underlying services provided by those components, and 3) the Data and Information, which models information handled by the business processes.

2.1.1 Modeling SOA systems

Several modeling languages can be used to model SOA architectures; i.e., the business and the infrastructure architectures. For the Business Architecture of a SOA system, there are modeling languages such the Business Process Model and Notation (BPMN) [OMG11], the Business Process Execution language (BPEL) [OAS07], the UML activity diagram (UML-AD)

[OMG09] and more. In the thesis research, we used the UML activity diagram to model the business architecture of SOA systems. The reason for this is that it is desirable to use a single modeling language, such as UML, for all SOA modeling views; i.e., workflow, structure, and behaviour. Second, UML activity diagrams can be extended with the MARTE profile for performance annotations, which allows us to study the performance properties of SOA systems.

The Infrastructure Architecture of a SOA system describes the components that provide the business capabilities and the behaviours of the underlying services provided by those components. Modeling the Infrastructure architecture requires three views:

1. The Service Structure view describes the configuration of the underlying services, the components and their relationships. It is modeled using SOA modeling language (SoaML) [OMG09b]. SoaML is a standard profile introduced by OMG to extend UML with the ability to define the service structure and dependencies, to specify service capabilities and classification, and to define service consumers and providers [PET12].
2. The Service Behaviour view describes the service operation details. It can be modeled using UML interaction diagrams. In the thesis, we used Sequence Diagrams (UML-SD).
3. The Structure view describes the configuration of the SOA solution. The configuration of the infrastructure of SOA architecture can be modeled using UML Deployment Diagram (UML-DD) [OMG09]. UML-DD models the deploying of software artifacts on nodes. It represents computational resources which are connected in a network through communication paths. The communication path allows the connected nodes to exchange messages.

2.1.2 Model-Driven Architecture for SOA systems

Model-Driven SOA (MDSOA) is a software development approach which is used to model SOA systems at different levels of abstraction. Model transformations are used to generate

other models or code; some models support the analysis of non-functional properties, such as performance [ALH10]. There is a lot of research on automatic model transformations to derive different kinds of models for SOA systems.

The research in [TAG09, RAF09] proposed an approach that uses standard UML profile to produce PIM models which are transformed into 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. In [ZHA09], MDA is used to generate automatically an executable WS-BPEL from different UML model views describing the process model and the service model. In [EMI07] a comprehensive metamodel is proposed that includes the elements of both the conceptual and the deployable model of SOA services. A UML profile is defined which maps the elements of the proposed metamodel to their equivalent in the UML superstructure. The UML profile can be used to generate the executable WS-BPEL. In [LEE09], an MDA-based approach is proposed where the developers select services that may be used in deployment models. The descriptions of these services (in WSDL) are transformed into UML models, and then using WSDL models; a new UML model of a composite web service, with its interfaces and its workflow is generated, which finally is implemented using BPEL.

2.1.3 SOA methodologies

Several SOA methodologies supported by tools are also available in the market. A SOA methodology describes the methods, techniques, and procedures that are used in designing and developing SOA solutions. The survey in [RAM07] presents the current SOA methodologies; three of the most notable are:

- *Service-Oriented Modeling and Architecture* (SOMA) is an IBM SOA methodology [ARS12, IBM04] which consists of three activities: service identification, service specification and

service and component realization. The service identification activity is supported by three kind of processes: a) Top-down process is defined as domain decomposition, where the business domain is decomposed into subsystems and components, while the business processes are decomposed into sub-processes and services, b) Bottom-up process, where functions provided by existing legacy systems are exposed as services, which can be further composed according to the business needs; and c) Middle-out process captures services which have not been identified by the top-down and/or bottom- up process.

The service specification activity is used to classify, and organize services into a hierarchical view, which helps to perform service composition, dependency, and allocation. The service realization is used to make decision regarding the suitable techniques that will be used to realize the service; i.e., top-down, bottom-up or middle-out.

- *Service-oriented modeling framework* (SOMF) provides a service-oriented life cycle modeling and management using a set of modeling views and disciplines [MET12, MET13]. SOMF defines three types of modeling time frames: the *Used-to-Be* modeling which defines the design of a software system that was running in the past, the *AS-Is* modeling which defines the design of software system that is running currently, and the *To-Be* modeling which defines the design of software system that will be deployed in the future. The modeling time frames provide modeling transparency between the legacy system and the future system, and make the business investment traceable, and justifiable. SOMF supports nine transformation models used to select between the project's components: the discovery model describes new software entities providing the solution; the analysis model inspects the software component ability to offer a solution; the design model describes the logical design of the software entities and structure of the components, and the deployment composition;

the technical architecture model describes the conceptual, logical, and physical architecture views; the construction model helps with modeling practices; the quality assurance model checks the ability of the software components for production; the operations model describes the deployment of the software entities; the business architecture model integrates the business concepts with the software entities; and finally the governance model provides standards and policies to all other models.

- *Service-Oriented Analysis and Design* (SOAD) aims to bridge the gap between the business domain and the IT domain. It builds upon three well-known software methodologies: Object-Oriented Analysis and Design (OOAD), Enterprise Architecture (EA) and Business Process Modeling (BPM). SOAD also supports additional SOA-based concepts [ZIM12], such as aggregating and categorizing services, policies and service aspects, meet-in-the middle processes, semantic brokering, service harvesting and knowledge brokering.

2.1.4 Service composition in SOA systems

Service Composition is the process of integrating a group of services to provide composite services that meet the user requirements [HIG12]. Service composition can be classified as a proactive, which represents the off-line composition of available services, and reactive composition, which represents the run-time composition of available services.

Service composition is a complex process if it is performed manually. The complexity of service composition is due to the large number of services to be searched in the service repository, and also because services are being created updated and destroyed continuously on the fly using different technologies and approaches [RAO04]. In [RAO04] two methods of service composition are defined: a) the workflow technique where a modeling graph is used to build the workflow process model, and then the atomic services are selected and composed

automatically; b) Artificial Intelligence (AI) Planning where a set of constraints and conditions are defined to generate the process model automatically using an engine.

In [SKO04], a web service composition is proposed using a UML-based model driven method. The goal of the method is to generate new composite services from the existing services defined in the service registry. In the composition method, a class diagram is used to describe the interface of the web service, and an activity diagram is used to describe the composition of the service operations.

In [MA09], a Fast Web Service Composition Approach is proposed that extends the Ontology Web Language for Services (OWL-S) [OWL13]. OWL-S is a semantic web service language (SWS), which provides an ontological description of web services to allow dynamic and automated discovery, invocation, and composition of web services. OWL-S is extended with new elements to handle the limitation related to dynamic transformations among components of web service compositions.

In [SIR05], the Ontology Web Language for Service (OWL-S) is extended by proposing the addition of abstract processes, which are workflow templates used to write abstract activities. The abstract activities are used to define the specifications of the required services. The paper uses web semantic ontology (OWL-S) to build flexible templates for describing the abstract functionalities, and encoding the QoS of web services within the workflow. The approach allows service discovery based on the required services. The discovered services are then ranked to find the most suitable one based on the encoded QoS defined in the templates.

2.1.5 Service variability in SOA systems

Variability is the ability to extend or configure the usage of a software system in a specific context [SIN06]. Using variability management in SOA systems helps in meeting the

required quality of service. When the current service is unavailable or does not satisfy the required QoS, the service can be replaced with one that meets the requirements, or it can be changed to achieve better performance. Variability also helps to optimize the quality attributes by changing the configuration of the system. It also supports run-time flexibility, where the rebinding of services can be done automatically [KON09].

The [NAR08, PON08, NAR10] researches proposed an end-to-end mechanism to develop SOA solutions by increasing the variants on the developed solution. The proposed mechanism is meant to reduce the development effort and to improve the reusability. The approach called Variation-Oriented Engineering (VOE) aims to explicitly identify the potential changes in the initial SOA solution, and to allow the development of derivative solutions by defining the incremental changes relative to the initial SOA solution [NAR08]. VOE defines three steps of a SOA solution: a) Variation-Oriented Analysis (VOA) where the variable parts are initially defined, b) Variation-Oriented Design (VOD) where the constraints governing the variations are specified, and c) Variation-Oriented Implementation (VOI) where the derived solution is implemented either manually or automatically.

In [KON09], VxBPEL is an extension of BPEL which introduces new activities that define variability in service-centric systems at run time. VxBPEL introduces additional elements to the XML process description that stores variability elements such as variation points, variants, and realizations in BPEL file. The goal of VxBPEL is to support changing services, optimizing service parameters, and addressing adaptive composition of web services.

In [CHA07], a framework is proposed to model a service variability using the elements of the product line engineering. Four types of service variability have been defined: a) Workflow variability defines the variation on the atomic services in business process, b) Composition

variability defines the variation on the interfaces of the composed services, c) Interface variability defines the variation of the service interfaces, and d) Logic variability defines the variation of the logical procedure of the service operations.

2.2 Modeling languages used in PUMA4SOA

We present in this section the modeling languages which are used to describe the source model (the input design models), the intermediate mode and the target model (the performance model) in PUMA4SOA model transformation chain.

2.2.1 Source Model: SoaML and MARTE Profiles

The *Unified Modeling Language* (UML) is a standardized modeling language which provides a set of graphical notations used for different activities of software system development. UML supports three types of extension mechanisms involved in the definition of a profile: 1) stereotypes used to extend different UML metaclasses, 2) tagged values which define the properties of the stereotype and 3) constraints which refine the semantics of a UML model element. The three extension mechanisms help modelers to extend the UML modeling capabilities by adding new concepts, new properties, and new semantics [OMG09a]. In our proposed approach, two UML profiles are being used: the SOA Modeling Language (SoaML) [OMG09b] and the UML Performance Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [OMG11a].

SoaML is a standard UML profile adopted by OMG, which defines service related details such as service structure, dependencies, capabilities and classification. In the SoaML metamodel a *Participant* element represents a person, an organization, a system in the business domain, an application or a component in the system domain. Each *Participant* has a *Role* in the service architecture which can be a *Consumer* or a *Provider* of a service. An *Agent* is a special kind of

participant, which is an autonomous entity that is able to adapt with the environment. A *Port* is a property of a *Participant* that specifies the point where the *Participant* interacts with the surrounded environment. A *Port* can be either a *Service* port when its participant offers a service or a *Request* port when its participant consumes a service. A *ServiceInterface* specifies the interface of a participant to provide or consume a service. The structure of the participant roles is represented as *Collaboration*. Two kinds of collaborations are defined: a *ServiceContract* which specify the agreement between the service providers, and service consumers, and a *ServicesArchitecture* which specifies how participants work together to provide or consume services using service contracts. A *ServiceChannel* represents the communication path between service providers and service requests. The message transferred between the consumers and providers is represented as a *MessageType*. A message may own an *Attachment* attached to it.

The MARTE standard profile [OMG11a] is used to annotate the structural and behavioural views of the UML design model in order to describe real-time characteristics of the system. MARTE extends the UML modeling for describing timing, resources, and other concepts used for performance and schedulability analysis. Ten sub-profiles are defined in the MARTE specification, which extends UML for different domains. In our proposed approach (PUMA4SOA), we are mainly using three of these profiles: a) the Performance Analysis Modeling (PAM) defined for performance analysis of systems with non deterministic behaviour and stochastic characteristics, b) the Generic Quantitative Analysis Modeling (GQAM), describing quantitative analysis concepts shared between the performance and schedulability domains, and b) the Generic Resource Modeling (GRM) which adds the general resource and platform concepts to the software model.

The PAM sub-profile defines concepts such as the *WorkLoadEvent* which describes a stream of arriving events, and different *ArrivalPattern* such as *OpenWorkLoad*, and *ClosedWorkLoad* patterns. The *AnalysisContext* is taken from the GQAM sub-profile to define the system global annotation parameters that can be used in expressions. PAM uses the behaviour-causality model of *Scenarios* and *Steps* and extends the properties of *Steps* to include more properties such as noSync, *extOpDemenad*, *behavDemand* and *ServDemand* [OMG11a]. A *Step* in PAM is further specialized into the *CommStep,* which expresses communication delays, and the *RequestedService* which models the service operations. Since performance analysis is determined by how the system behaviour uses system resources [PET09b], the GQAM domain model defines different kinds of resources such as *Computational Resource*s (the *ExecHost*, and *CommHost*) and *SchedulableResource* which describes concurrent processes with threads. In the PAM domain, there is *RunTimeInstance,* and the *LogicalResource*.

2.2.2 The Intermediate Model: Core Scenario model (CSM)

The Core Scenario Model (CSM) captures the performance properties of a UML design model into a scenario-based unified intermediate model [PET04, PET07]. The CSM metamodel represents performance concepts which are close to those contained in the performance sub-profile of the SPT profile [OMG05] and of the MARTE PAM as well [OMG11a]. A *CSM* instance model is a collection of scenarios and different kinds of resources. A *Scenario* is described as a collection of steps linked together by connectors, and a *Resource* is an element which executes operations. A *Step* is an action within a scenario, and a *PathConnection* is a connector which is linked to at least one step. A Step can be either primitive when it performs a single action or refined when it performs multiple actions (sub-scenario). A *Step* is also specialized into two types: *ResourceAcquire* step which describes the acquired component and

ResourceRelease Step which describes the released component. The *PathConnection* is specialized into seven types: *Start*, *End*, *Sequence*, *Branch*, *Merge*, *Fork* and *Join* [PET07]. There are also various types for resources. The super class *GeneralResource* is specialized into two types: *ActiveResource* and *PassiveResource*. The *ActiveResource* defines the *ProcessingResource* and the *ExternalOperation*, while the *PassiveResource* defines the *Component*. A *Scenario* has a workload, which defines the intensity of usage of the scenario.

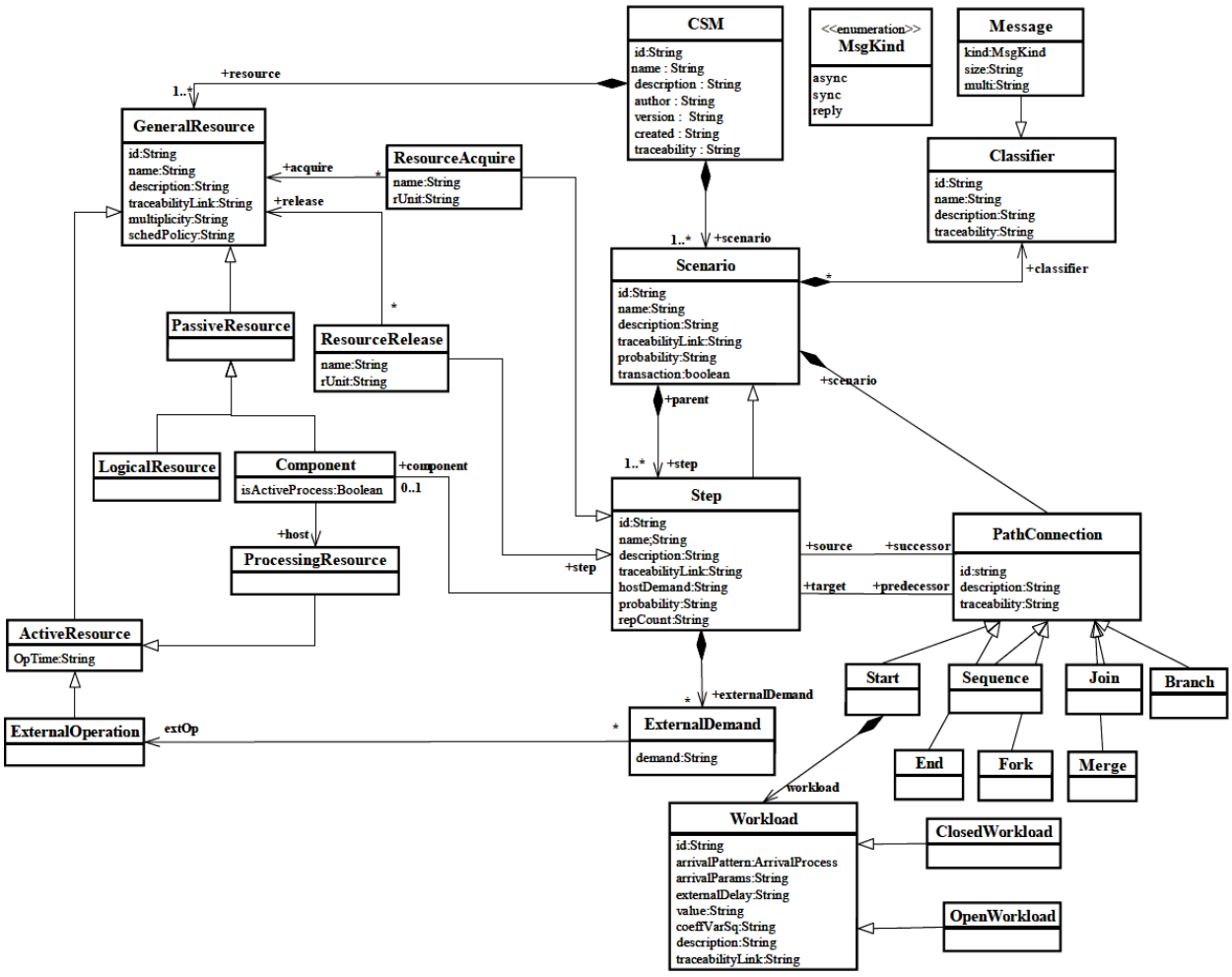


Figure 2.1: The metamodel for the Core Scenario Model

There are two types of workloads: the *ClosedWorkload* which defines a fixed number of requests, and the *OpenWorkload* which defines an unlimited number of requests. A *Message* is associated with path connections and it defines the message type sent between system

components. A *Message* can be of three kinds: *async, sync* and *reply* [ALH08]. Figure 2.1 describes the CSM metamodel.

2.2.3 Target Model: LQN

The LQN model contains software resources called *tasks* which are allocated to hosts called *processors*. A *task* provides a set of services called *entries*. A *task* also owns a queue where requests for entries are waiting for the service based on scheduling type. The process of an *entry* is described using a set of *activities* connected by different precedence relationship. An *activity* represents the smallest operation unit in LQN model.

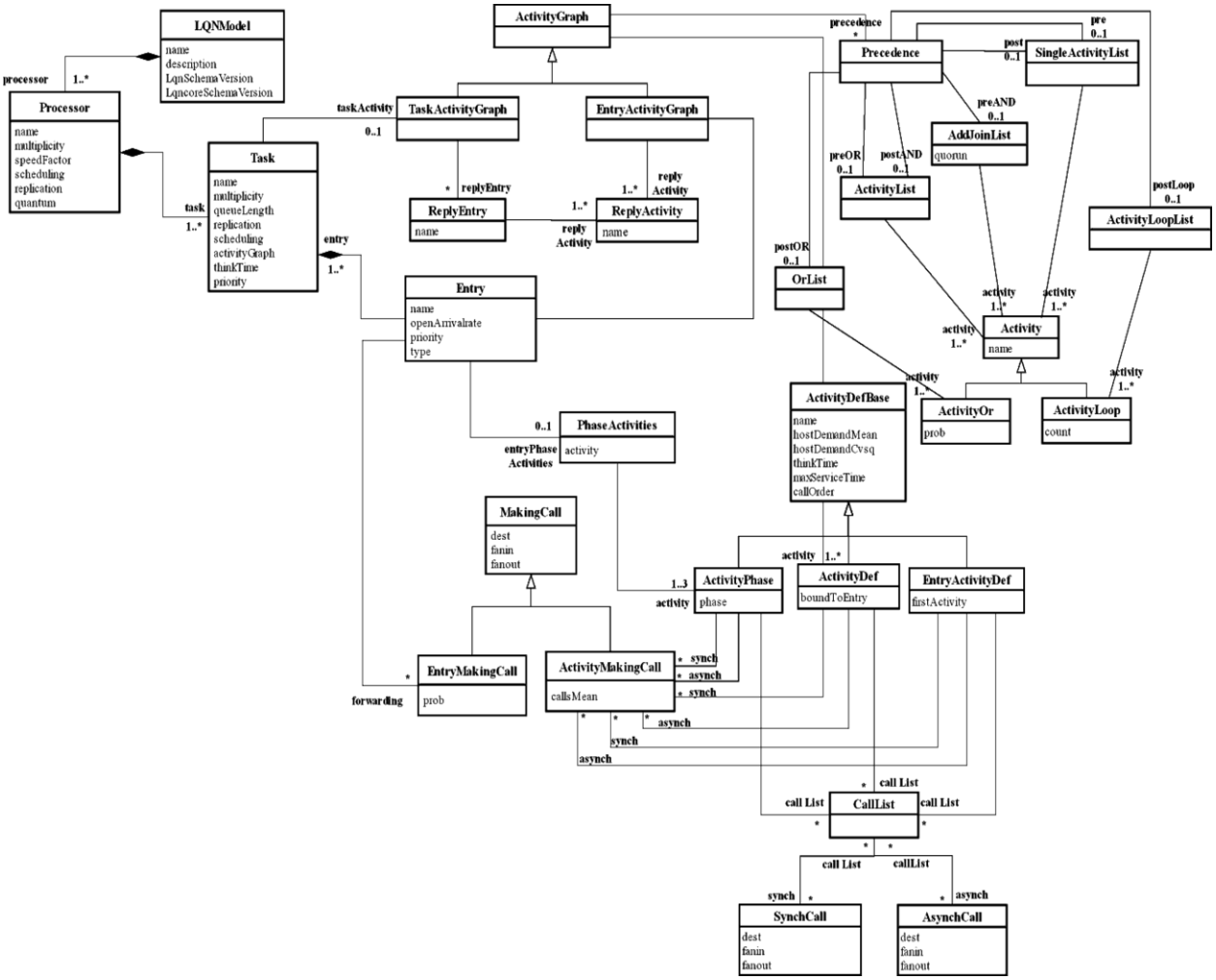


Figure 2.2: The LQN metamodel

Activities can also be described in the model in the form of *phases*, where they are constrained to be executed in a sequence of one to three. The requests from an activity (or phase) to an entry can be *synchronous* (where the client is blocked, waiting for a reply), *asynchronous* (there is no reply) and *forwarding* (which allows for a client request to be processed by a chain of servers: the first server in the chain will forward the request to the second, etc., and the last server in the chain will reply to the client). Figure 2.2 describes the LQN metamodel as in [GRA08].

2.3 Aspect-Oriented Modeling Techniques

Aspect oriented modeling (AOM) is a technique for modeling and managing crosscutting concerns. The AOM architecture contains a *primary model* which describes the core design model, and a set of *aspect models* which represent the addressed concerns. A set of rules, called *point-cuts*, identify the *join-points* (the location in the primary model where the aspect will be inserted or "woven"). An aspect model, initially defined in a generic form, can be instantiated multiple times - once for each join-point - to produce multiple context-specific aspect models. The context-specific aspect models are then composed into the primary model by weaving them into their corresponding join-points [ALH08].

In [SIN05], an aspect-oriented web service approach (AOWS) is proposed, which addresses problems with current web service aspects, especially related to description, discovery, and integration mechanisms. A framework developed using aspect oriented component engineering (AOCE), describes the AOWS component and systemic capabilities from different aspect-oriented point of views.

In [XU07], an approach was presented to address service composition weaknesses, such as lack of support for dynamic adaptation of the composition, specifications which cannot explicitly modularize concerns, and composition specifications without a standardized visual

representation. The UML language is used to model the web service composition and aspect oriented modeling (AOM) is used to perform the composition of the aspects into the core model.

In [ABU07], another aspect oriented approach aims to improve process flexibility when applying features to web services and to reduce resource consumption. To achieve the goals, the proposed approach defines an aspect oriented extension module, called aspect binding, which modularizes the logic of applied feature. Aspect bindings hold the feature's application logic. The aspect oriented extension architecture allows for applying aspect bindings to different web services on different web service engines.

2.4 Traceability Model

The software development in MDE is model-oriented, where various types of models are used to describe the software under development at different levels of abstractions during the software life cycle. During the software development, models can be created, updated, transformed and/or deleted. This dynamic nature of the models raises challenges related to the ability of managing, configuring and maintaining consistency between the software models. One of the solutions is to establish relationships between the elements of the different models using trace-links. Traceability is a software mechanism for establishing relationships between different software artifacts during the software life cycle. It is used to understand the relationships and dependencies between the software artifacts, to maintain consistency and to propagate the changes between them.

Various traceability approaches have been introduced in the software modeling literatures. The authors of [GAL07] classified the traceability approaches based on the target model into three types: a) *requirement-driven approach*, where traceability is defined as the ability to trace a requirement during its life cycle (i.e., initiation, refinement, iteration, use and

deployment), 2) *modeling approach*, where traceability is defined as the ability to follow the relationship between elements in different models which represent the same concept; and 3) *transformation approach*, where traceability is defined as the trace-links between the elements of the source and the target model in a model transformation. In [KOL06], traceability approaches have been classified based on storage and manageability into two types: a) *embedded traceability*, where the trace-links are defined as new model elements within the same models they are linked to; and b) *external traceability*, where the trace-links are stored in an external new model to maintain traces separately from the model they are linked to. In [PAE02], traceability approaches have been classified based on the types of the trace-links into two approaches: a) *explicit trace-links,* where traceability is defined as the ability to apply trace-links directly in the models using their concrete syntax, as example UML dependencies, and b) *implicit trace-links,* where traceability is defined as the ability to apply trace-links using model management operations such as a model transformation.

2.5 The state of the art

We discuss in this section a set of related works which are similar to the proposed PUMA4SOA, in the sense that they focus on the derivation of performance models from software design specifications. We defined a list of criteria for the assessment of these works and evaluated each one of them against the defined criteria.

2.5.1 Criteria for comparison

We present the criteria we selected for evaluating and comparing related works which derive performance models from software design models. The criteria take into account the modeling languages used and additional features provided by each approach, such as automation, traceability and variability.

1. Source model properties
   1.1. Source domain classification: component-based, distributed, real-time, SOA, automotive and avionics.
   1.2. Modeling language(s) used for the source model and whether they are standard or non standard: UML (standard), CSM (non-standard).
   1.3. Modeling profiles used (if any), such as MARTE and SoaML.
   1.4. Modeling views for the source model: structural, behavioural.
2. Target model properties:
   2.1. Performance modeling language used for the target model.
3. Automation of model transformation: specifies whether the automation of the transformation is implemented or not.
4. Modeling of service platform: specifies the strategy used to represent the service platform in the software model, such as PIM/PSM separation and variability.
5. Traceability: specifies whether traceability between the software model and the performance model is implemented or not.
6. Feedback of performance results to design model: specifies if the approach is capable of presenting the output result of the performance model in the software models.

2.5.2 Related works for deriving performance models from software models

We present in this section the most recent research works focused on the derivation of performance model from software models. These papers were selected by searching in several resources such as Google Scholar, IEEE Xplore Digital Library, ACM Digital Library and [BAL04]. The papers are evaluated, compared and summarized using the criteria defined in Section 2.5.1.

2.5.2.1 Performance from Unified Model Analysis (PUMA)

Performance from Unified Model Analysis (PUMA) is a performance analysis approach, developed in the Department of SCE at Carleton University, which uses an intermediate model for transformations between different kinds of design models to different kinds of performance models [WOO05, WOO09]. The goals of PUMA are to reduce the complexity of transformations between many design models to many performance models, and make the transformations consistent. Initially a UML design models annotated with MARTE is developed by a software engineer, and then this model is transformed into a CSM intermediate model, which may be transformed in turn to different performance models such as LQN, QN, Petri nets, simulation. Using an existing performance solver, the performance model is analyzed and the performance results are obtained.

2.5.2.2 User Requirements Notation (URN)

The User Requirements Notation (URN) is an ITU-T standard which combines two notations modeling languages: the Goal-Oriented Requirement Language (GRL) for modeling the intentional concepts and the Use Case Maps (UCM) for modeling scenario interactions, behaviours and architecture. URN is used for requirements elicitation, analysis, specification and validation [ITU12, MUS09]. In [ZEN05] URN was used to evaluate the performance model of a software system by transforming the Scenario-based model (UCM) into a performance model (LQN). URN is supported by an open-source Eclipse plugin called UCMNav, which is able to automatically generate a performance model suitable for various kinds of performance analysis [PET03] and jUCMNav in [MUS09] which transforms UCM into CSM models.

2.5.2.3 Palladio component model (PCM)

PCM [BEC09] is a component-based approach which supports the development process of Component-Based Software Engineering (CBSE) by dividing the creation of the software model into four development roles: component developer, system architect, system deployer and domain expert. PCM is defined by a dedicated metamodel. Its graphical notation is similar to UML class, component and activity diagrams. Several models can be instantiated from a PCM metamodel; i.e., repository model, system model, resource environment model, allocation model and usage model, which are used to predict the QoS properties, such as performance and reliability, early in the software life cycle. PCM modeling is supported by open source tools, which includes model editor, simulator, experiment controller and performance completion library.

2.5.2.4 Architectural UML Software Models to Performance Model

In [VER05], an automatic model transformation framework is used to map the UML platform independent model (PIM) into a platform specific model (PSM). The framework is used to study the impact of the middleware on the design and performance of distributed software system. The input PIM model is a UML activity diagram describing the system's behavior, a UML deployment diagram describing the system's configuration, UML collaboration diagram describing the architectural patterns of the system and a description of the middleware usage in XML format. The generated UML platform specific model is transformed into a performance model, such as LQN, and performance analysis is used to extract the final software system.

2.5.2.5 P-WSDL

The approach in [DAM07] introduces an extension to the Web Services Description Language (WSDL) with performance characteristics of a web service called P-WSDL. The extended

WSDL is used to integrate the performance characteristics of the invoked services within BPEL process. The BPEL process is mapped into an annotated UML activity diagram, and an automatic model transformation is used to build an LQN performance from the annotated UML activity diagram where the performance of the service-oriented application is evaluated.

2.5.2.6 UML+CSM+GSPN

The approach in [GOM06] uses the PUMA approach to study the performance of the SOAP toolkit and XML parsers used by the web service platform on which the service-oriented applications are running. The design model, UML sequence diagram annotated with SPT, is transformed to the Core Design Model (CSM), which is turn is transformed into Generalized Stochastic Petri net (GSPN) used for performance analysis.

2.5.2.7 UML4SOA+MARTE

The approach uses UML4SOA [TRI10, MAY10] which is an extension of UML2.0 to describe the behavior of the individual services and service orchestrations using activity diagrams and composite structure diagrams. MARTE is used to describe the performance properties of the modeled system. The deployment diagram describes the system configuration. The design models are then transformed manually to a LQN model.

2.5.2.8 Performance Evaluation Process Algebra (PEPA)

PEPA [HIL96] is a performance modeling paradigm which uses stochastic process algebra for specifying the software system. PEPA extends the classical process algebra such as Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP). It provides a set of conventions for describing the behavior of a system such as Component, Activity, Action type and Activity rate [HIL96]. A system is described in PEPA as a set of active agents who collaborate to perform certain behavior. Expressions are used in PEPA to model the

behavior of components based on their activities and their combinations. There are several combinations such as: Prefix *(α, r).P*, Choice *P1+P2*, Cooperation *P1><P2*, Hiding *P/L* and Constant $A \stackrel{\text{def}}{=} P$, where *α* represents an activity, *r* represents the activity rate and *P, L, A* represents components [GIL06]. Different tools have been implemented to support PEPA, such as PEPA Plug-in for Eclipse [ECL13] and IPC, the Imperial PEPA Compiler [IPC13]. Model transformations from the software design model to PEPA model have been introduced in several papers. As an example, an automatic transformation from UML Sequence Diagram to PEPA is presented in [TRI08a] and an automatic transformation from UML Activity Diagram to PEPA in [TRI08b].

2.5.2.9 Sensoria Reference Markovian Calculus (SRMC)

SRMC [CLA09] is a process calculus which is used to model services. It is extended from PEPA to allow three levels of uncertainties: a) uncertainty of the system configuration, b) uncertainty of the parameters of some components and c) uncertainty of the duration of the events. The cause of those uncertainties is due to the lack of details of the modeled system. The SRMC language is suitable for sensitivity analysis, since it allows the modelers to define the values of the parameters of the three uncertainties on the top of the initial model, which is typically a PEPA model. There are many tools can be used to model design and simulate SRMC models, such as the SMC compiler [SMC13].

2.5.2.10 Quality Impact Predictions for Evolving Service-oriented Systems (Q-ImPrESS)

Q-ImPrESS [KOZ11] is a model driven approach which aims to predict the performance, reliability and maintainability of SOA systems. The development process of the Q-ImPrESS is based on an abstract design model called Service Architecture Model (SAM). SAM is used to capture the details of the software system such as system structure, behavior deployment and

usage scenarios [IMP13]. The development process of Q-ImPrESS defines the following steps: 1) gather new requirement, 2) define change scenarios, 3) model change scenario, 4) predict system quality based on performance, reliability and maintainability, 5) tradeoff analysis, 6) implement SAM models, 7) validate models and 8) deploy system. The main advantage of the Q-ImPrESS approach is that it combines the prediction analysis for several non-functional properties of the software system. This allows performing tradeoff analysis and provides a wide range of alternative designs.

2.5.2.11 UML+QN based performance model

The approach proposed in [COR00] uses the Software Performance Engineering (SPE) methodology to evaluate the performance properties of the software architecture by generating a queuing network based performance model from UML software diagrams. Different UML diagrams are used: a UML use case diagram to describe the user profile and the software scenarios, a UML sequence diagram to describe the Execution Graph (EG), a UML deployment diagram to describe the extended queuing network model (EQNM) and produce an instance of EG. The combined EG-instance and EQNM are used to obtain the performance model.

2.5.2.12 Design models + KLAPER+ Analysis models

A model driven approach introduced in [GRA08] uses a kernel language called KLAPER which captures the non-functional characteristics of component-based systems. KLAPER is an intermediate model which bridges the gap between the design models and analysis models. The design models are described using UML diagrams annotated with SPT. The KLAPER model is first generated from the annotated design models. Then, different kinds of performance models (such as LQN and Petri net) can be generated from the KLAPER model. The automation of model transformations in this approach is still under development.

2.5.3 Comparison between the presented approaches and PUMA4SOA

The comparison between the related works presented previously and PUMA4SOA is based on the criteria described in Section 2.5.1. The source models are used for different modeling domains: PUMA and UML+QN based performance model for the distributed real-time systems; PCM and Design models + KLAPER+ Analysis models for component-based systems; URN, PEPA and SRCM can be used for any kind of software domain; the rest are used for the SOA domain (P-WSDL, UML+CSM+GSPN and Q-ImPrESS are specialized in web service-based systems). Different modeling languages are being used to describe the source model: UML with its profiles (SPT, MARTE) are used in most of the above approaches; the URN is an ITU-T standard which supports GRL and UCM; the PCM and Q-ImPrESS are using user-defined languages (PCM and SAM) respectively. Different approaches are also using different views to describe the software systems, such as behavior model, workflow model, architecture model, goal-oriented model, collaboration diagram, use case diagram and deployment diagram.

The related works described above use different target models, such as the LQN, which is common between most of the approaches. The target model of PEPA and SRMC approaches is using stochastic process algebra, while UML+CSM+GSPN uses Generalized Stochastic Petri nets (GSPN). Most of the approaches previously discussed do perform automatic model transformations, except for UML+CSM+GSPN, UML4SOA+MARTE and UML+QN. Our proposed approach (PUMA4SOA) partially implements model transformation, as some parts of the transformations are still under development in another thesis.

Most of the related works described above do not provide the ability to model the service platform. PUMA4SOA, PCM and Architectural UML Software Models to Performance Model are the only ones which describe the service platform structure and behavior using Performance

Completion (PC) feature modeling. PCM uses the concept of performance completion to address the separation between the software system and the service platform [HAP08]. In PUMA4SOA, we separate the platform independent model (PIM) from the platform specific model (PSM) and use AOM to model the service platform.

Finally, PUMA4SOA is the only approach which uses a traceability model to define the trace-links between the elements of the software model and the performance model, and to present the feedback of performance results to the software model.

Table 2.1 summarizes the comparison between the related works and PUMA4SOA.

Table 2.1: Comparison between model driven approaches

		PUMA	URN	PCM	Architectural UMLtoPerformance Model	P-WSDL	UML+CSM+GSPN	UML4SOA+MARTE
Source model	**Domain**	Distributed RT	Generic	Component based	Distributed systems	Web services	Web services	SOA
	Modeling language(s)	UML (standard)	URN (standard)	PCM (non standard)	UML (standard)	UML(Standard)	UML(Standard), WSDL (Standard)	UML4SOA (Standard)
	Modeling profile(s)	SPT, MARTE	NA	NA	SPT	SPT	SPT	MARTE
	Modeling views	1) Behavior model(s) 2) Deployment diagram	1) Goal-Oriented model 2) Use case map	1) Repository model 2) System model 3) Resource environment model 4) Allocation model 5) Usage model	1)Behavior models 2) Deployment diagram	1) Behavior model(s) 2) Deployment diagram	1) Behavior model(s) 2) BPEL process	1) Workflow model 2) Deployment diagram
Target model	**Modeling language(s)**	QN, LQN, Petri net	LQN	PCM, LQN	LQN	GSPN	LQN	LQN
Automation of model transformations		Yes	Yes	Yes	Yes	Yes	No	No
Modeling of Service platform		NA	No	Performance completions	PIM/PSM	NA	NA	NA
Traceability between models		No	No	No	No	No	No	No
Feedback of performance results to design model		No	No	No	No	No	No	No

Table 2.2: Comparison between model driven approaches (continued)

		PEPA	SRMC	Q-ImPrESS	UML+QN based performance model	Design models + KLAPER+ Analysis models	PUMA4SOA
Source model	**Domain**	Generic	Web services	Generic	Generic	Component-based	SOA
	Modeling language(s)	UML (standard)	SAM (non standard)	UML (standard)	UML (standard)	UML (standard)	UML (standard)
	Modeling profile(s)	MARTE, SoaML	NA	MARTE		SPT	SPT, MARTE, SoaML
	Modeling views	1) Behavior model(s) 2) Service architecture model	1) service architecture model	1) Behavior model(s)	1) Behavior model 2) Use case diagram 3) Deployment Diagram	1) Behavior model 3) Deployment Diagram	1) Workflow model 2) Service architecture model 3) Service behavior model 4) Deployment diagram.
Target model	**Modeling language(s)**	stochastic process algebra	PCM, LQN, simuCom	Stochastic Process Algebra	Extended queuing network model (EQNM)	LQN, Petri net	QN, LQN, Petri nets
Automation of model transformations		Yes	Yes	Yes	No	No	Yes (partially)
Modeling of Service platform		NA	NA	NA	NA	Klaper	1) PIM/PSM separation 2) Platform variability
Traceability between models		No	NA	No	No	No	Yes
Feedback of performance results to design model		No	NA	No	No	No	Yes

Chapter 3: PUMA4SOA Approach and Case Study

Chapter 3 presents our proposed approach PUMA4SOA which generates performance models from the UML design model of the SOA systems. The approach aims to evaluate the runtime performance characteristics of such systems in early development phases, before the entire system is built and can be deployed and measured. Early performance evaluation helps to choose an appropriate architecture, design and configuration alternatives, so that the final system meets its performance requirements [ALH10].

Section 3.1 of this chapter presents the new features of PUMA4SOA compared with PUMA. Section 3.2 presents the input design models of PUMA4SOA using a Purchase Order system case study. Section 3.3 presents the PC feature model used to select the platform aspect models. Section 3.4 presents the platform aspect model which defines the middleware structure and behaviour of the platform aspects in a generic format.

3.1 PUMA4SOA approach

Performance from Unified Model Analysis (PUMA) is a research project in the Department of System and Computer Engineering at Carleton University [WOO05, WOO13, XU03]. The project aims to provide a unified interface between different kinds of software design models, and different kinds of performance models. The importance of the project came from the complexity of performance analysis starting from software designs in UML annotated with SPT [OMG05] or MARTE [OMG11a], which had to be transformed to various types of performance models. Figure 3.1 describes the PUMA architecture [WOOD05].

The proposed approach of this research extends PUMA using Model-driven SOA (MDSOA). The approach is called PUMA4SOA and is described in Figure 3.2.

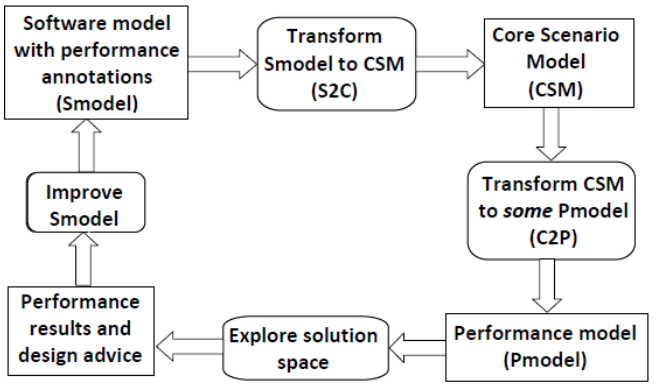


Figure 3.1: PUMA Approach

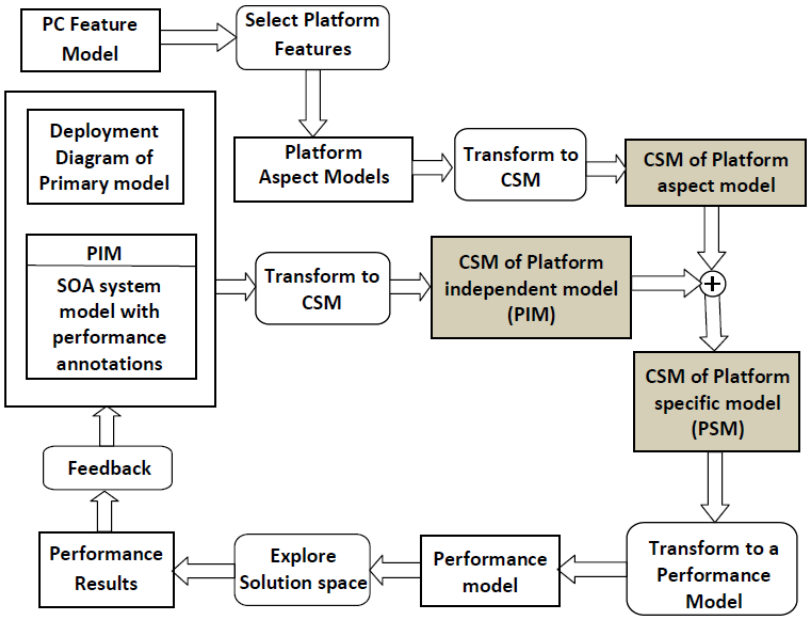


Figure 3.2: PUMA4SOA Approach

MDSOA is a software approach which focuses on models to represent service-oriented systems at different levels of abstractions. Model transformations are also used in MDSOA to generate other kinds of model for the analysis of non-functional properties, such as performance. The extension provides new capabilities to PUMA that allows evaluating the performance characteristics of SOA systems. The differences between PUMA and PUMA4SOA stems from:

1. Kinds of input design models: the top leftmost of Figure 3.2 represents three input models to PUMA4SOA: a) the platform independent model (PIM) of SOA systems, b) the deployment diagram, and c) a set of Platform aspect models.
2. Separation between the platform independent model (PIM) and the platform specific model (PSM) of SOA systems: the input design model is initially built as a platform independent model. After selecting the platform aspect models, PIM is transformed into a platform specific model using AOM aspect composition. The separation of platform models from the application model allows reusing the performance characteristics of different platforms, making the building of the performance model more efficient [ALH10].
3. Use of platform aspect models: Platform aspect models describe the structure and behavior of the service platform in a generic format. They are used during AOM composition to generate the platform specific model (PSM).
4. Use of a Performance Completion (PC) feature model: it represents the variability in the service platform. Each possible feature is realized by an aspect model. The PC feature model allows the modeler to select multiple platform features based on the business requirements, and consequently to identify the aspect models to be composed with the PIM.
5. Traceability between the elements of the UML, CSM and LQN is used in PUMA4SOA. We have developed a traceability metamodel which is used to instantiate the traceability models. A traceability model describes the trace-links between elements of UML, CSM and LQN models during the model transformation chain of PUMA4SOA. Trace-links can be used to define the dependencies between related elements in different models, to propagate changes of properties from one model to another and to analyze the impact of these changes.

Traceability between LQN2UML is also used to feed back the performance results from the LQN model to the UML model, which is not available in PUMA.

3.2 Input design model: Purchase Order System

This section presents the UML input design model to PUMA4SOA. A purchase order system example is used to illustrate these models.

3.2.1 Platform-independent model

The platform independent model in PUMA4SOA contains a UML software model with three levels of abstractions. The UML model is annotated with performance information using the standard UML profile MARTE. Each level represents a part of the performance details that will be used together with the other parts to build the performance model. The three abstractions levels are as follows: a) *Workflow Model*, b) *Service Architecture Model*, and c) *Service Behaviour Model* [PET12].

3.2.1.1 Workflow model

The Workflow model is the top level view in PIM which describes the business process of SOA design model. A workflow contains a sequence of actions linked by control flows and controlled by conditions, iterations, and concurrency. The Workflow model can be represented by different graphical modeling notations, such as the BPMN, or the UML-AD. The advantage of BPMN is that it supports different types of events, choreography, transactions and tasks. BPMN may seem better suited because of its richer business semantic. However, in our proposed approach, we use UML activity diagrams due to two reasons: First, it is desirable to use a single modeling language for all SOA levels; i.e., workflow, structure, and behaviour. Second, UML activity diagrams can be extended with the MARTE profile for performance annotations.

UML activity model does not support the concept of choreography workflow, where peer to peer external interactions occur between services. Choreography workflow allows several participants (Organizations), which can be either internal or external, to collaborate with each other to achieve the business goals. The internal participant belongs to the local enterprise, whose business process details (such as service invocation and performance properties) are managed by the designer. An external participant, however, operates outside the boundaries of the local enterprise, and its business process details are usually opaque. When an internal participant (service consumer) requests a service from an external participant (service provider), a service level agreement (SLA) is used to negotiate the level of performance properties at run time. The UML activity diagram allows for modeling of a single workflow to describe the workflow collaboration between the internal and external participants. It does not support the choreography business process which is modeled by separating between the workflows of the internal and external participants. We use two new stereotypes to extend the *ActivityPartition* element of UML activity model: a) ‹‹*internal*›› stereotype to indicate the *ActivityPartition(s)* representing the local participant(s); and b) ‹‹*external*›› stereotype to indicate the *ActivityPartition(s)* representing the non-local participant(s) boundaries. Please note that an ‹‹*external*›› stereotype has been introduced in UML version 2.2 [OMG09].

Figure 3.3 describes the workflow of a purchase order system which starts when *POProcess* engine runs the *ReceivePO* action, followed by a parallel process of invoicing and scheduling actions. Then, the *ShipProduct* action is processed by the external shipper server. The UML activity diagram shows different swimlanes that contain the actions and activities of different participants. An activity can be either atomic when it contains a single action or composite when it is refined to a scenario of actions.

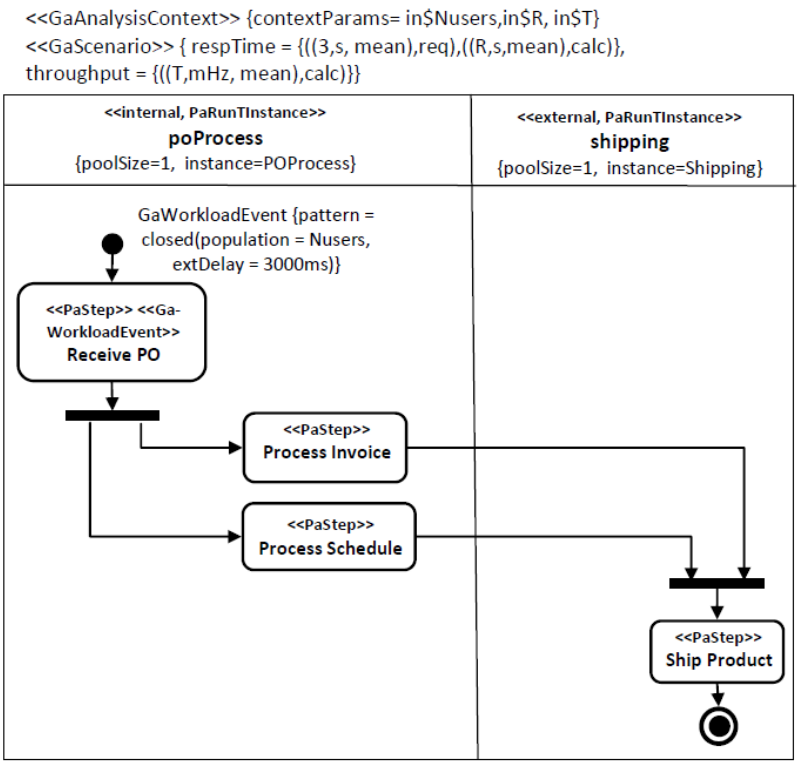


Figure 3.3: The Workflow Model of PO System

The workflow shows two swimlanes: the ‹‹*internal*›› *ActivityPartition* which describes the server that runs the purchase order process engine (*POProcess*) and the ‹‹*external*›› *ActivityPartition* which describes the shipping server that runs outside our target enterprise. Different elements from the MARTE profile are also used to describe the performance characteristics of SOA systems. A Swimlane is stereotyped as ‹‹*PaRunTInstance*›› to indicate a concurrent participant that will be executed by a specified process or task instance. Such process may have a *poolsize* attribute giving its number of threads and instance for the ‹‹*SchedulableResource*›› that this swimlane represents. Each action is stereotyped as ‹‹*PaStep*›› to indicate a unit of an operation. The first action, *ReceivePO,* has a stereotype called ‹‹*GaWorkloadEvent*›› which defines the stream of events that produces the workload to the system. The workload in the example has a *closed* arrival pattern with a *population* attribute that

defines the number of concurrent users, and *extDelay* defining the delay between the end of one cycle and the start of the next.

3.2.1.2 Service Architecture Model

A new OMG profile called the Service-Oriented Architecture Modeling Language (SoaML) [OMG09b] is used to model the service structure. It extends UML with the ability to define the service structure and dependencies, to specify service capabilities and classification, and to define service consumers and providers [ALH12a]. A Service structure model also helps the modeler to specify the level of service granularity. The level of service granularity defines the size of the provided services, and it has a substantial effect on the system performance [EAR05]. In SOA systems, where the environment is heterogeneous and distributed, the cost of marshalling/unmarshalling data increases the messaging overhead. To reduce the effect of marshalling/unmarshalling overheads, it is preferred to use services with a large granularity. On the other side, large granularity produces unnecessary coupling between the components of a SOA system, which makes them hard to implement, or to evolve and change over the time [OMG09b, ALH10]. Service Architecture Modeling helps the modeler to manage the trade-off between granularity and performance.

Figure 3.4 describes the service structure model of the purchase order system. There are four components: Sales, Invoicing, Scheduling, and Shipping. Components are stereotyped as ‹‹*Participant*›› from SoaML profile to indicate parties that provides or requires services, and they also can be stereotyped as ‹‹*SchedulableResource*›› from MARTE profile to indicate a process or a thread pool. The Sales, Invoicing and Scheduling are three internal participants which are defined within the local enterprise. The Shipping participant is the external participant. A component can be either a service consumer (requester) or a service provider or both of them.

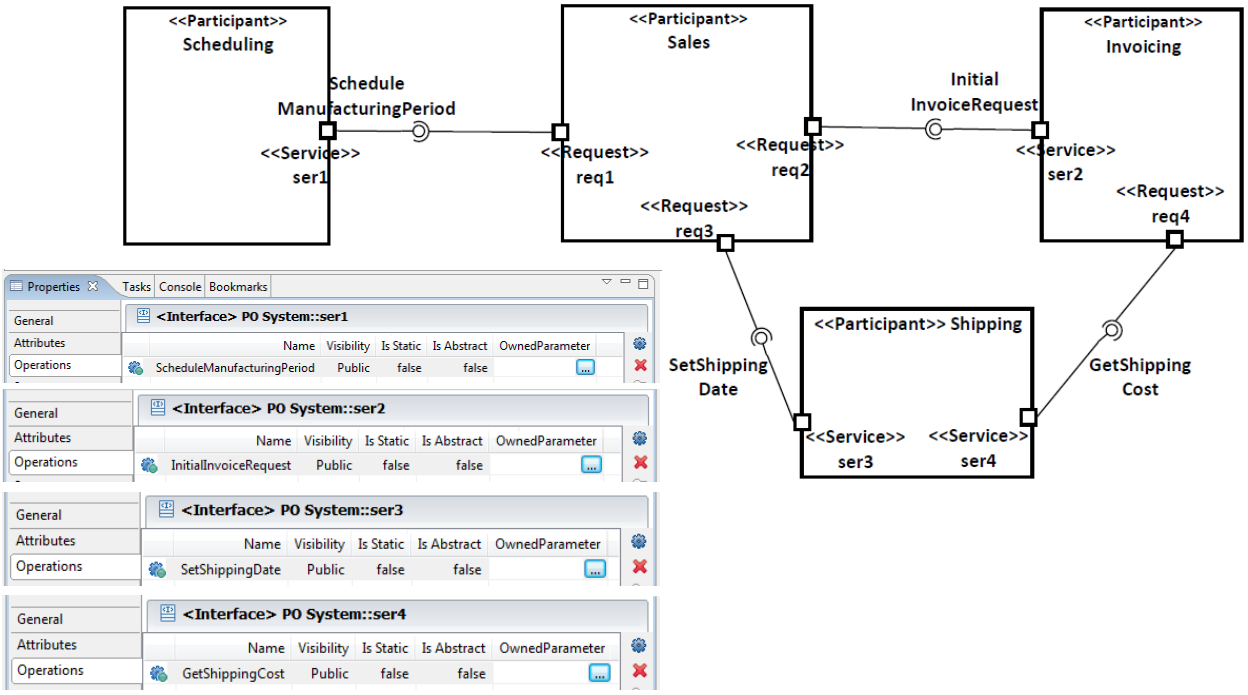


Figure 3.4: The Service Architecture Model of PO System

When the component provides a service, its port which is providing, is stereotyped as ‹‹*Service*›› from the SoaML profile to indicate the provided service. The service port can be associated with one of the provided interfaces represented with the lollipop notation “o— “. Within the interfaces, properties, attributes and operation signatures are defined as shown in Figure 3.4. When a component consumes a service, its port requesting the service is stereotyped as ‹‹*Request*›› (with a SoaML stereotype indicating the request of a service). The request port can also be associated with one of the required classes represented as “)— “. The tool window showing the properties of the required interfaces is not shown in the figure.

3.2.1.3 Service Behavior Model

The Service Behavior Model refines the workflow behavior, giving more details about the services invoked. Each workflow action or activity is refined using a UML sequence diagram which represents its detailed behavior, including the invocations of other services and the interaction between participants [PET12]. Figure 3.5 describes the behavior of the

ProcessSchedule action of the purchase order system. Lifelines are stereotyped as ‹‹*PaRunTInstance*›› to indicate a concurrent participant that will be executed by a specified process or task instance. Messages are stereotyped as ‹‹*PaStep*›› to indicate a unit of operation. A step has different attributes, such as *hostDemand* for the required execution time, *prob* for its probability of execution, and *rep* for the number of repetitions. Messages are also stereotyped as ‹‹*PaCommStep*›› to indicate the conveyance of a message, which has a *msgSize* attribute to indicate the amount of data transmitted by the sender. *MsgSize* can be used to calculate the overheads at the transmission and receiving points using *commRcvOvh*, and *commTxOvh* attributes defined for the execution host of the respective participant [ALH10].

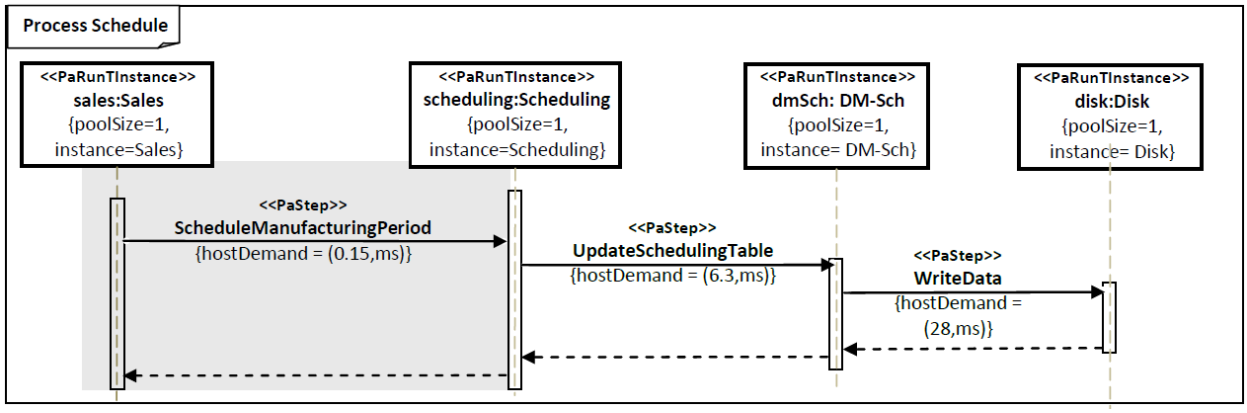


Figure 3.5: The Sequence Diagram of *ProcessSchedule*

3.2.2 Deployment diagram

The deployment diagram defines the allocation of software to the hardware. A UML Deployment diagram defines different types of Nodes and Artifacts. Figure 3.6 describes the deployment diagram of the purchase order system. The *SalesHost*, *InvoicingHost, and SchedulingHost* are distributed over one LAN, while the *ShippingHost* is installed on a different LAN. Both LANs communicate through a wide area network (WAN). Several servers are installed; the *POProcess* is the business process engine that runs the purchase order process, the

Sales, *Invoicing*, and *Scheduling* servers run on hosts that belong to the local enterprise, while the Shipping server run on an external host. The Disk is used to store and update a database that is related to the Purchase order, such as client details, PO history, stock inventory, etc. Data management applications are used to manage the database; examples of the databases are *DM-Sales and DM-Ship*.

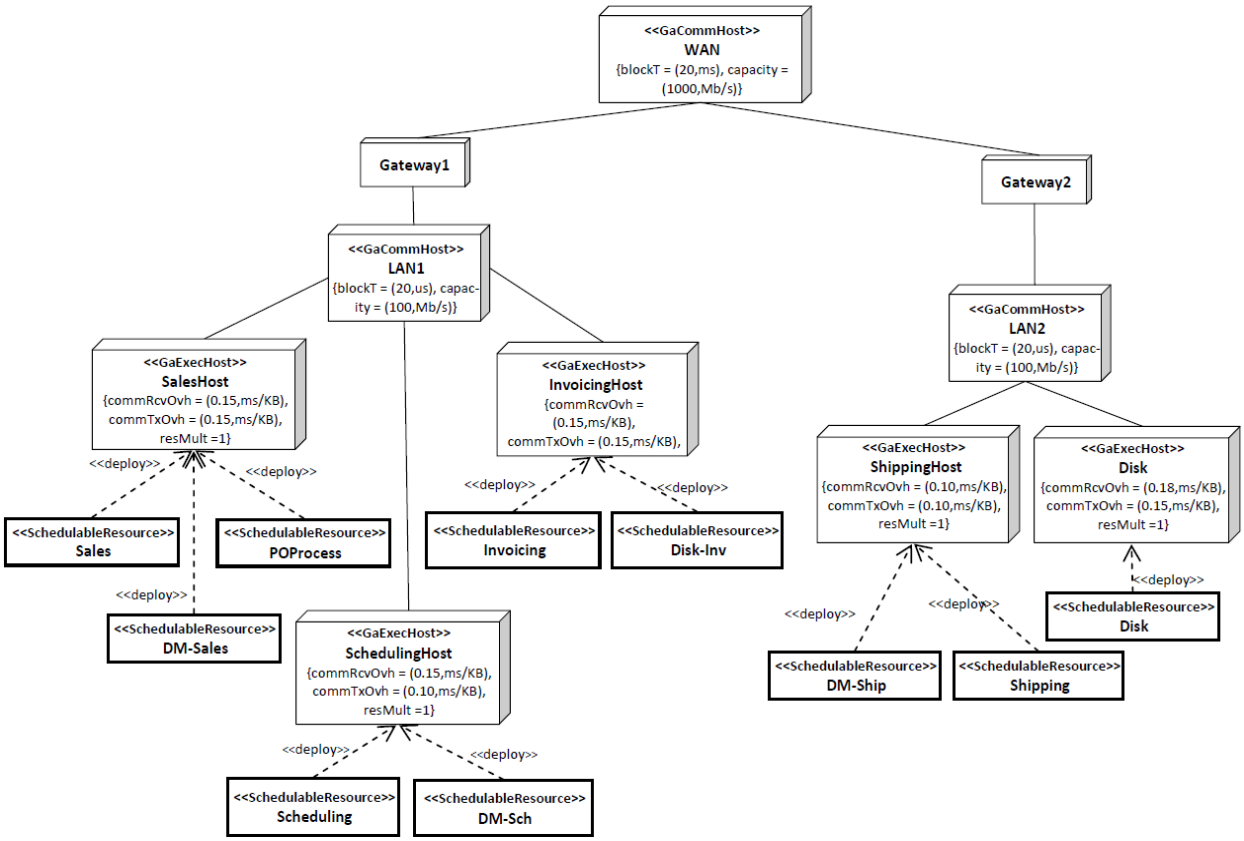


Figure 3.6: The Deployment Diagram of PO System

Different MARTE annotations are also used; the ‹‹*GaCommHost*›› stereotype identifies a physical communication link, which may have a *blockT* attribute for the network latency, and *capacity* for the maximum throughput rate. The ‹‹*GaExecHost*›› stereotype identifies a processor resource hosting the components deployed to it, which may have *commRcvOvh* and *commTxOvh* attributes for transmitting and receiving communication overheads. The ‹‹*SchedulableResource*›› stereotype indicates a process with a thread pool [ALH10].

3.3 Platform variability: the Performance Completion (PC) feature model

The PC feature model describes the variability in the structural and behavioural models of the service platform which may affect the system performance. It provides the alternatives and choices to select between multiple aspects based on the business requirements. The concept of "performance completions" was first introduced in [WOO02] to narrow the gap between the design models and the external platform factors. It was also used in [HAP10] and in [TAW11], to define the variability in platform choices, types of platform realizations, and other external factors that have an impact on the system performance.

Since the regular notation for feature diagrams is not part of UML, we use a UML extension proposed in [CLA01, CLA01a], where the feature diagram is represented as a UML class diagram extended with stereotypes to represent the PC feature model. Each feature is represented as a class element. The relationships between a feature and its sub-feature are defined using stereotypes *‹‹mandatory››*, *‹‹optional››* and *‹‹alternative››* which are used to annotate class representing mandatory, optional and alternative feature respectively. The PC feature model in our approach does not support complete constraints between features, grammars and consistency checks.

Each feature in the feature model represents a platform aspect. A platform aspect model describes the structure and the behavior of the service platform in a generic format. PUMA4SOA supports a PC feature model which allows the modeler to select between different platform aspect models that are most appropriate for the business requirements [PET12]. Figure 3.7 describes the features which may affect the performance of our example, the Purchase Order System. There are three mandatory feature groups which are required by any service platform: the operation, message protocol and realization. There are also two optional feature groups:

communication and data compression. The relationship between the feature groups and their sub-features are alternative with exactly-one-of the feature selected. In order to select a service platform aspect, at least one feature from the mandatory feature groups (operation, message protocol and realization) must be selected [PET12]. As an example, selecting one of the operation features, such as invocation, requires selecting one of the message protocol (HTTP or SOAP) and the realization (WebService, REST, etc.).

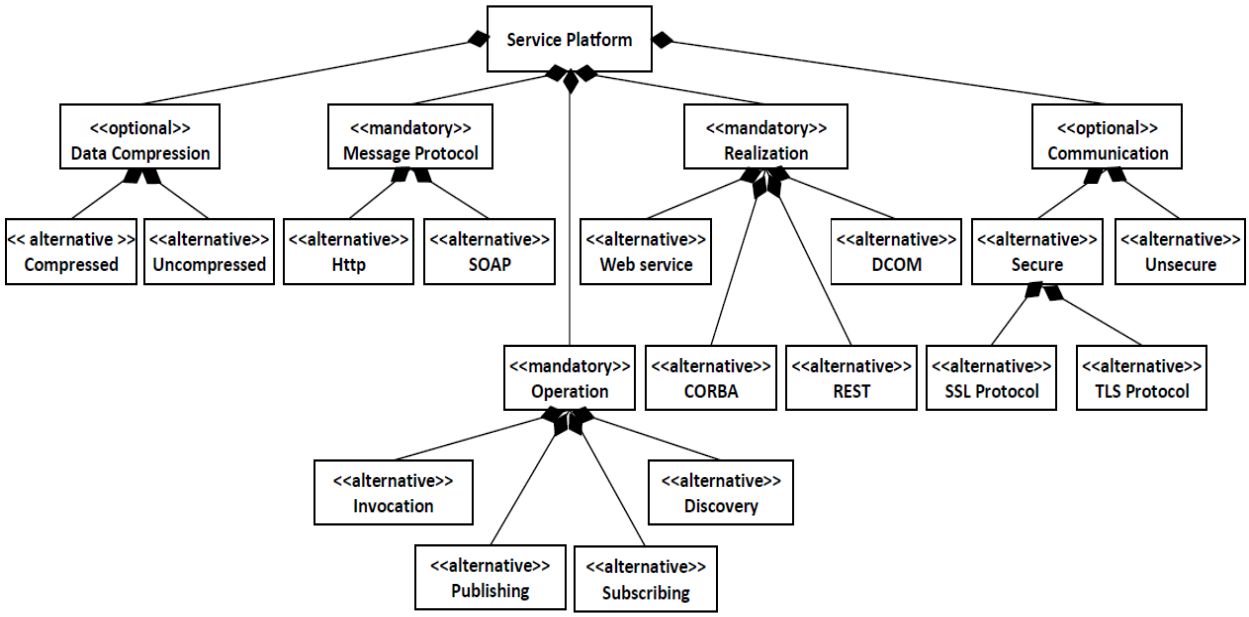


Figure 3.7: The Platform Completion (PC) Feature Model

3.4 Platform Aspect Model

The Platform Aspect Model defines the middleware structure and behaviour of the platform aspects in a generic format. It provides the underlying software infrastructure that allows the business layer to utilize SOA. This enables the separation of services from the implementation, and allows the heterogeneous components to offer services that have no implementation dependency [COL11].

The platform architecture of SOA system defines two components: the service consumer, and the service provider. The service consumer invokes services offered by the service provider

and the service provider responds to the service invocation requested by service consumer. The service consumer also discovers and subscribes whenever there is a demand for a new service. Some technologies, such as the Universal Description, Discovery and Integration (UDDI), add a third component called a service broker. UDDI is a set of specifications that is used to publish information for describing and discovering web services, finding businesses, building registries [UDD04]. The service broker acts as an intermediate role between consumer and provider to store and allocate services. Figure 3.8 describe the conceptual view of web service roles.

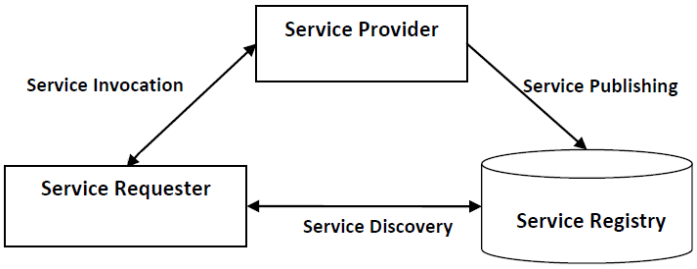


Figure 3.8: The conceptual view of Web Service roles

SOA platform supports different aspects that describe the behavior interaction between SOA components; i.e., Consumer, Provider, and Registry. These platform aspects are used to generate the PSM from the PIM. The next sections presents the generic aspect models of three platform aspects: service invocation, service publishing and service discovery.

3.4.1.1 Service Invocation

Service invocation describes the platform behaviour when a client requests a service. The deployment diagram in Figure 3.9 describes the generic Hosts and Artifacts used in Service Invocation aspect. Two generic hosts, |*ClientHost* and |*ProviderHost*, are linked through a generic network (|*Network*) which can be |LAN, |Gateway, |WAN or combination. The |*ClientHost* deploys three components: |*client* (the service consumer), |*XMLParserC* and |*SOAPClient*. The |*ProviderHost* deploys three components: |*Provider* (the service provider),

|*XMLParserP* and |*SOAPProvider*. The parser component converts the message to XML format, and the SOAP component transfers the service request/response. The service behaviour invocation is described in Figure 3.10. The interaction model begins with a service request message, followed by the middleware operations such as message marshalling and serializing. The service is then invoked at the provider side, and the response is returned back to the client.

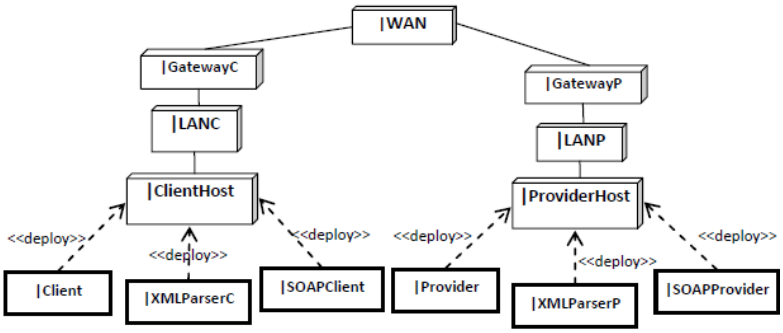


Figure 3.9: The Service Invocation Aspect: Deployment View

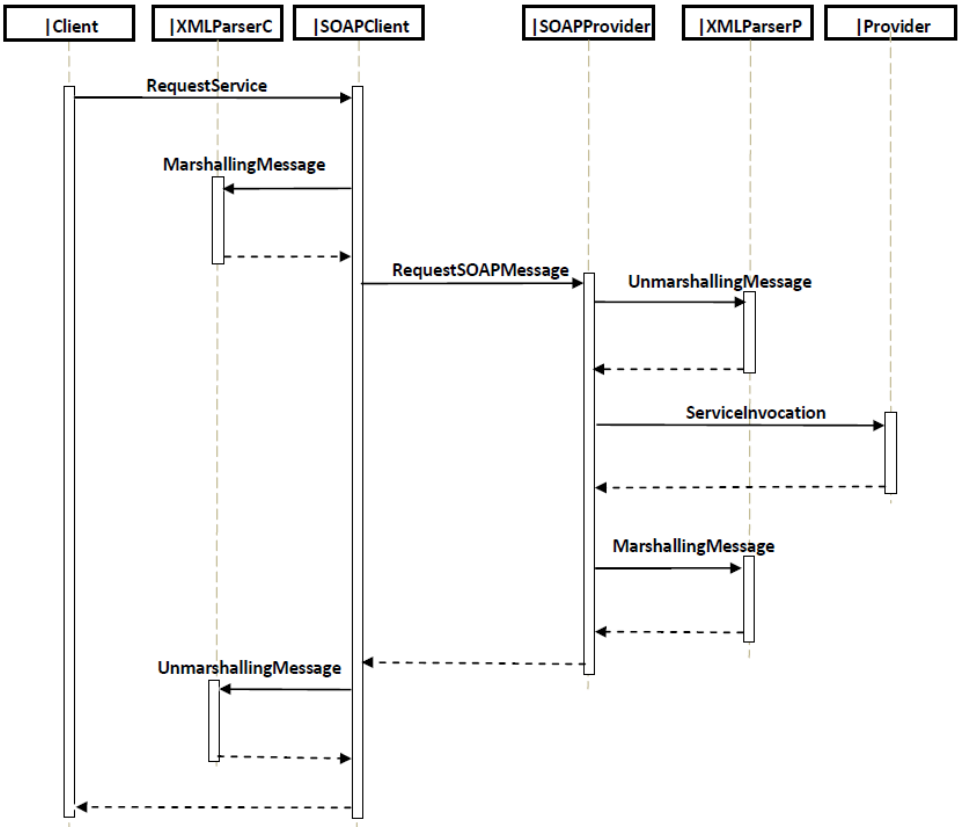


Figure 3.10: The Service Invocation Aspect: Behavior View

3.4.1.2 Service Publishing

Service publishing allows web services to be shared over a distributed system. The service providers publish the required components in a service registry, which has a known location to service consumers [KAS08]. In order to publish a service at a UDDI registry, the service provider first must build the service WSDL, and then deploy it in a public location. The service provider must then send a SOAP message to the registry to indicate the location of the WSDL using a URI. Figure 3.11 describes the deployment diagram of the generic Hosts and Artifacts used in service publishing aspect. Three generic hosts are defined: |*ProviderHost*, |*RegistryHost* and |*Disk*. The |*ProviderHost* deploys three components: |*Provider* (the service publisher), |*XMLParserP*, and |*SOAPProvider*. The |*RegistryHost* also deploys three components: |*Registry* (the service registry), |*XMLParserR*, and |*SOAPRegistry*. The |Disk host saves the information of the published service, such as URI. Figure 3.12 describes the service publishing behavior interaction where it begins when the |*provider* publishes a service into a registry. The registry receives the publishing request and stores the service details into a disk.

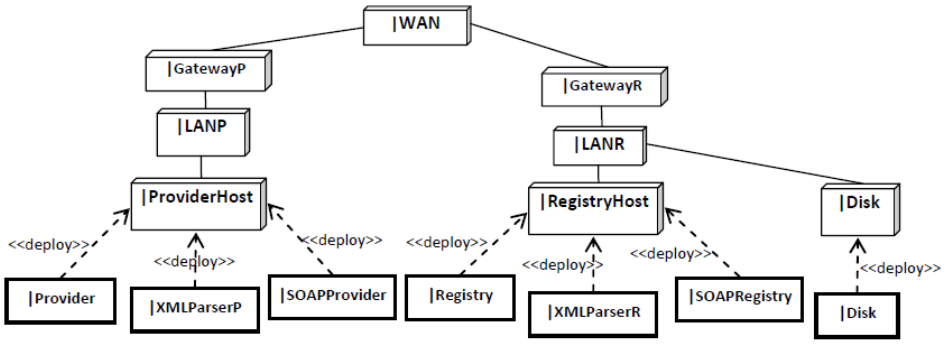


Figure 3.11: The Service Publishing Aspect: Deployment View

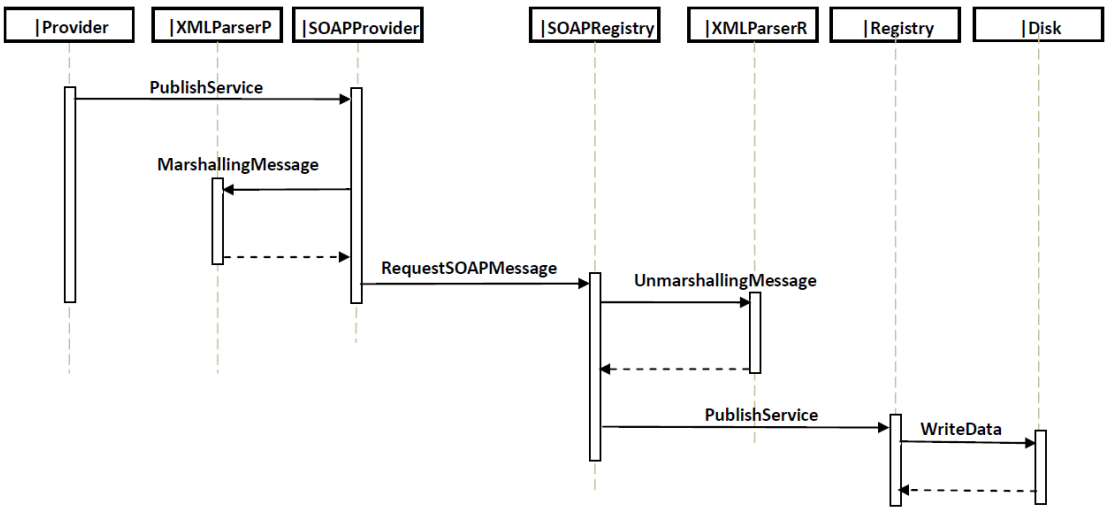


Figure 3.12: The Service Publishing Aspect: Behavior View

3.4.1.3 Service Discovery

Service Discovery is the process of querying services within the registries in order to meet the business requirements [SAP06]. In service discovery, the web service description must be discovered, and then it must be matched with the user requirements. There are three types of web service discovery models based on the availability of web service descriptions: static, centralized and decentralized. Web service description is static when it is stored locally; it is centralized when it is published to a directory service, and it is decentralized when it is published in a distributed directory.

Figure 3.13 describes the deployment diagram of the generic Hosts and Artifacts used in the service discovery aspect. Three generic hosts are defined: |*ClientHost,* |*Registry* and |*Disk*. The |*ClientHost* deploys three components: |*Client* (the service consumer), |*XMLParserC,* and |*SOAPClient*. The |*RegistryHost* also deploys three components: |*Registry* (the service registry), |*XMLParserR*, and |*SOAPRegistry*. The |*Disk* is used to retrieve the information of the queried service. Figure 3.14 describes the service discovery behavior. It begins when a client queries a

service; the registry receives the query and searches for the service details, reads from the disk and returns the details back to the client.

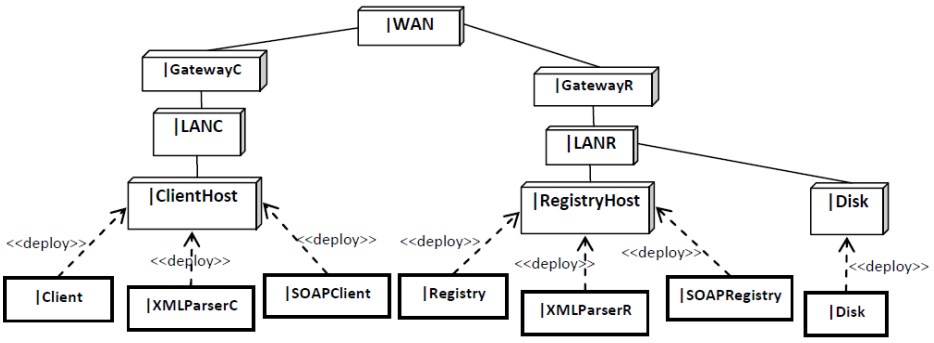


Figure 3.13: The Service Discovery Aspect: Deployment View

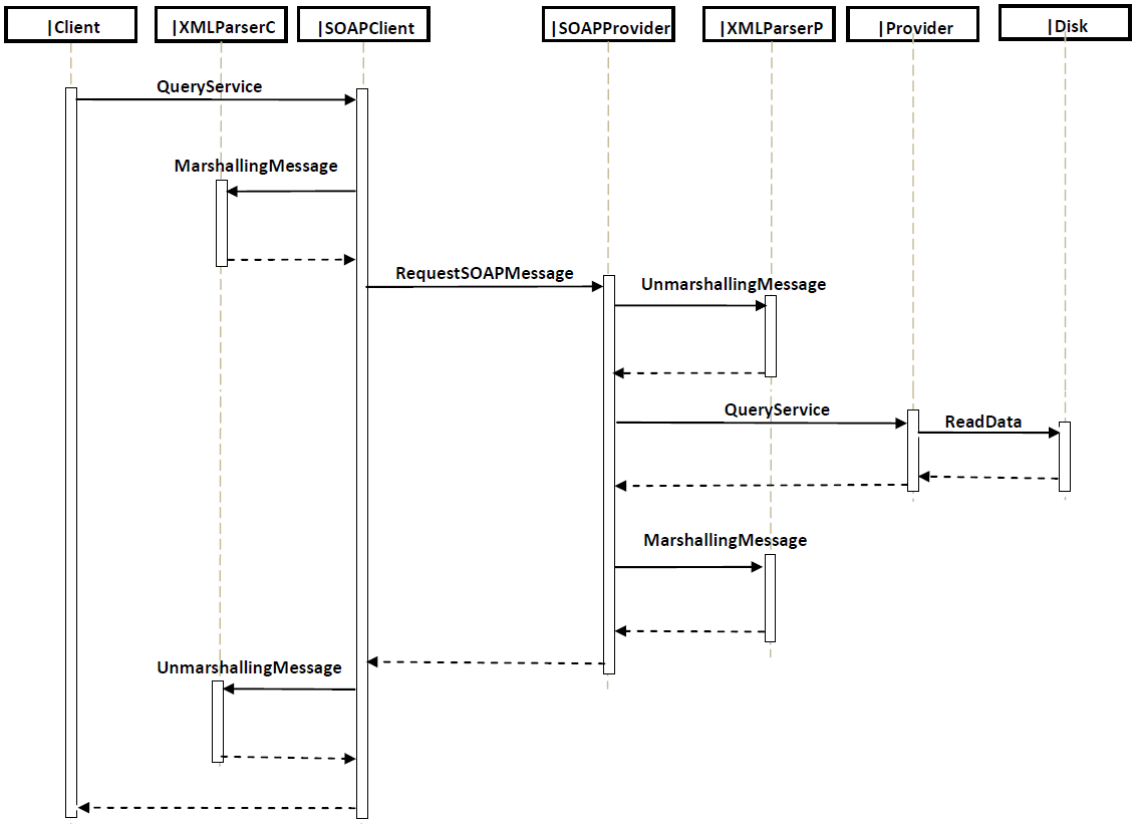


Figure 3.14: The Service Discovery Aspect: Behavior View

Chapter 4: Model transformations

Chapter 4 presents specifications and techniques used in PUMA4SOA which describe the model transformation chain: a) from the UML design model to the Core Scenario Model (CSM), and b) from the Core Scenario Model (CSM) to the LQN Performance model.

The actual implementation of the model transformations presented in this chapter is in Java using the Eclipse environment, similar to PUMA which PUMA4SOA is extending. However, we use OMG's QVT language [OMG08] for describing the pseudo-code of the transformations in a precise and succinct way.

Section 4.1 presents a brief overview of the model transformation language QVT. Section 4.2 describes the model transformation from UML design models to a CSM model (UML2CSM) and from a CSM model to an LQN model (CSM2LQN).

4.1 QVT specification overview

QVT is a model transformation language that mainly focuses on model query, view and transformation. It supports two levels of declarative language (the Relations and the Core) and an imperative language (the Operational mapping). The Relations language is used in matching object patterns, tracing model elements and creating object templates. The Core language is considerably simpler than the Relations language and is used in matching patterns where a set of variables are evaluated against a set of models [OMG08]. The Operational mappings language is used for invoking the imperative representations of transformations from the Relations or the Core.

The QVT specification contains different modeling concepts which are used in matching patterns, creating templates and tracing objects. A *Relation*, as an example, defines the relationships which must hold between the elements of a set of models. Within a *Relation* block,

there can be multiple *domains*, a *where* clause and a *when* clause. A *domain* is used to specify the related elements of the candidate models. A *when* clause (acting as a pre-condition or a guard) defines the conditions that make the *Relation* hold. A *where* clause (acting as post-condition) defines the conditions that need to be satisfied by the related model elements. An *Operational Transformation* concept is specified by the Operational mappings metamodel, which specifies the imperative description of a unidirectional transformation using model signatures of QVT semantic concepts. An operation, called *main*, is defined to execute the transformation. A *MappingOperation* is another concept in the Operational mapping which specifies the mapping between the elements of the source and target models. A *MappingOperation* may have a *MappingParameter* and a *MappingBody*. A *MappingParameter* specifies the parameters used and their directions which can be *in*, *out* or *inout*. A *MappingBody* defines the block of a mapping operation, which is consists of three sections: the *initialization*, the *population* and the *termination*. A *helper* is another QVT concept which acts as a method that computes several source objects and provides a result.

4.2 Model transformation from the UML design model to the LQN Performance model

The platform independent model of UML design model defines two modeling layers, as seen in Chapter 3: the business layer represented by the workflow model, and the service layer, represented by the service architecture model and the service behaviour model. Each model represents a part of performance model that will be used together with the other parts to build the entire performance model.

During PUMA4SOA model transformation, the UML activity diagram which represents the workflow model needs to be transformed separately from the SoaML diagram which

represents the service architecture diagram, and UML sequence diagram which represents the service behavior diagram.

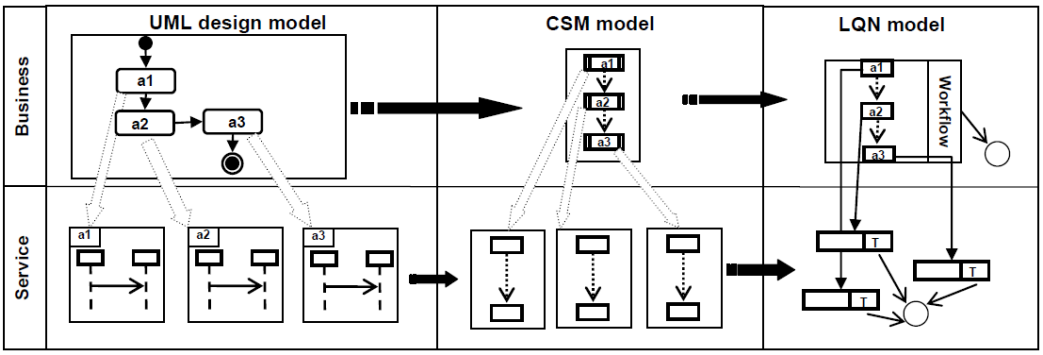


Figure 4.1: The Model Transformation in PUMA4SOA for the Business and Service layers

Several specifications have been used to describe the model transformation chain of PUMA4SOA. Specification 1 and 2 describes a high level view of the model transformation from UML to CSM (UML2CSM) and from CSM to LQN (CSM2LQN) which separates the mapping of the business layer from the service layer, as in figure 4.1.

In Specification 1, a *RefinementType* is a CSM element, not shown in the metamodel, which is aggregated to *StepType* and used to link the refined step in the CSM top scenario to the subscenario that describes its behaviour details.

Specification 1: UML2CSM Model transformation: Business and Service layers

```
transformation UML2CSM (in uml:UML, out csm:CSMType){
    main(){
        //Specification 1a:  UML2CSM model transformation for the business layer
        csm:= map activityToTopCSM(uml);
        //Specification 1b: UML2CSM model transformation for the service layer
        subScenario:= map interactionToSubScenarios(uml);
        //Linking the subscenarios to the CSM top scenario.
        constructor RefinementType::refinement(i:String){Id:=i}
        csm.getSteps() → forEach(step){
            if(step.getName() = subScenario.getName()){
```

Specification 1: UML2CSM Model transformation: Business and Service layers

```
            refinement.setSub(subScenario)
            refinement.setParent(step) } }  }
  return csm}.
```

Specification 2: CSM2LQN Model transformation: Business and Service layers

```
  transformation CSM2LQN (in csm:CSMType, out lqn:LQN){
    main(){
      //Specification 2a: CSM2LQN model transformation for the business layer
      lqn:= map topCSMToActivityGraph(csm);
      //Specification 2b: CSM2LQN model transformation for the service layer
      lqn:= map subScenariosToUnderlyingTasks(csm, lqn);    }
  return lqn}
```

The next sections present specifications 1a and 1b and specifications 2a and 2b in more detail. These specifications describe the mapping between the elements of the UML, CSM, and LQN.

4.2.1 Transformation from UML to CSM

The model transformation from UML design models to a CSM model occurs first at the business layer, where the UML workflow model is transformed into a CSM top scenario model. The workflow contains a sequence of activities and actions controlled by conditions, iterations, and concurrency [ALH10]. An activity represents a behavior that is composed of individual actions [OMG09]. Each action is refined into a sequence diagram called a service behavior model, which represents its detailed behavior, including the invocations of services and the interactions between participants. The model transformation occurs next at the service layer where each sequence diagram of a refined activity is transformed into a set of CSM subscenario models that describe the services.

Model transformations from UML to CSM have been described by two previous works. In [LUI08], a model transformation from UML activity diagram to CSM (UMLAD2CSM) has been described by mapping a subset of UML2.0 activity diagram elements annotated with a subset of MARTE stereotypes. A similar approach has been introduced in [ISR05], where model transformation from a UML sequence diagram to CSM (UMLSD2CSM) using a subset of UML2.0 sequence diagram elements annotated with a subset of SPT profile. The model transformation of [ISR05] is used in the original PUMA where traditional software systems are applied. In our approach, we extend the model transformation from UML to CSM by mapping most of the UML activity diagram and sequence diagram elements annotated with additional MARTE stereotypes. We also integrate the two model transformations (AD2CSM and SD2CSM) to handle the mapping of the business and service layers of the target model (UML design models) of SOA systems.

Specification 1a describes the details of the mapping presented in Specification 1 (*activityToTopCSM)*, where the elements of the workflow model are mapped to the elements of the CSM top scenario model. In Specification 1a, the elements of the activity diagram are classified into five groups based on their generalization elements, as follows:

1. The *ControlNode* group describes the elements which coordinate the flow in the activity diagram (such as *InitialNode*, *ForkNode*, *DecisionNode*). The elements of type *ControlNode* are mapped into CSM elements of type *PathconnectionType* (such as *StartType*, *ForkType*, *BranchType*).
2. The *ActivityEdge* group describes the elements which make the connections between two activity nodes (such as *ControlFlow* and *ObjectFlow*). The elements of type *ActivityEdge* are mapped into a CSM element of type *SequenceType*.

3. The *Action* group describes the elements which define a single step within the activity diagram (such as *AcceptEventAction*, *SendSignalAction*). The elements of type *Action* are mapped into a CSM element of type *StepType*.
4. The *StructuredActivityNode* group describes the portion of the activity that is not shared by other activity nodes (such as *ConditionalNode*, *SequenceNode*). The elements of type *StructuredActivityNode* are mapped into a CSM element of type *ScenarioType*.
5. The *ObjectNode* group describes the elements which describe the instance of a particular classifier (such as *CentralBufferNode*, *DataStoreNode*). The elements of type *ObjectNode* are mapped into a CSM element of type *StepType*.

The specifications 1a(1) to 1a(5) describe the mapping of the elements of the five groups. The elements within the same group inherit most of their properties from the parent element. Special properties are defined for each individual element. To differentiate between the elements of the same group which are mapped to the same CSM element type, we use the new *metadata* element, defined in the CSM metamodel, to tag the CSM element with a stereotype corresponding to the UML model element type (as generated by the RSA tool). *Metadata* is a light weight profiling mechanism which extends a CSM metamodel by defining name-value pairs to capture stereotypes (discussed in more detail in the next chapter).

Specification 1a: Mapping UML elements to CSM elements: Business layer

```
mapping Activity::activityToTopCSM(in uml:UML) :CSMType
    when{uml.getPackagedElement() <> null} {
        uml.getPackagedElement()→forEach(activity){
            constructor CSMType::csm(i:String, n:String){ id:=i, name:=n};
            constructor ScenarioType::scenario(i:String, n:String){id:=i, name:=n};
            csm.add(scenario);
            activity. getOwnedElement()→forEach(ownedElement){
```

Specification 1a: Mapping UML elements to CSM elements: Business layer

```
//Specification 1a(1): Mapping AD ControlNode to CSM PathconnectionType
pathConnection:= ownedElement.objectsOfType(ControlNode) → map
controlNodeToPathConnectionType(ownedElement);
scenario.add(pathConnection);
//Specification 1a(2): Mapping AD ActivityEdge to CSM SeqeunceType
sequence:= ownedElement.objectsOfType(ActivityEdge) → map
activityEdgeToSequenceType(ownedElement);
scenario.add(sequence);
//Specification 1a(3): Mapping AD Action to CSM StepType
step:= ownedElement.objectsOfType(Action) → map actionToStepType(ownedElement);
scenario.add(step);
//Specification 1a(4): Mapping AD StructuredActivityNode to CSM ScenarioType
scenario:= ownedElement.objectsOfType(StructuredActivityNode) → map
structuredActivityNodeToScenarioType(ownedElement);
scenario.add(scenario);
//Specification 1a(5): Mapping AD ObjectNode to CSM StepType
step:= ownedElement.objectsOfType(ObjectNode) → map
objectNodeToStepType(ownedElement);
scenario.add(step);
//Specification 1a(6): Mapping AD ActivityPartition to CSM ComponentType
component:= ownedElement.objectsOfType(ActivityPartition) → map
activityPartitionToComponentType(ownedElement);
scenario.add(component);
} }
return csm; }
```

Specification 1a(1): Mapping AD ControlNode to CSM PathconnectionType: Business layer

```
mapping controlNodeToPathConnectionType(in node:ControlNode):PathConnectionType{
    pathconnection:= node.objectsOfType(InitialNode) → map intialNodeToStartType(node);
    pathconnection:= node.objectsOfType(ForkNode) → map forkNodeToForkType(node);
```

Specification 1a(1): Mapping AD ControlNode to CSM PathconnectionType: Business layer

pathconnection:= node.objectsOfType(JoinNode) → map joinNodeToJoinType(node);

pathconnection:= node.objectsOfType(DecisionNode) → map decisionNodeToBranchType(node);

pathconnection:= node.objectsOfType(MergeNode) → map mergeNodeToMergeType(node);

pathconnection:= node.objectsOfType(FinalNode) → map finalNodeToEndType(node);

return pathconnection; }

Specification 1a(2): Mapping AD ActivityEdge to CSM SeqeunceType: Business layer

mapping activityEdgeToSequenceType(in edge: ActivityEdge): SequenceType{

sequence:= edge.objectsOfType(ControlFlow) → map controlFlowToSequenceType(edge);

sequence:= edge.objectsOfType(ObjectFlow) → map objectFlowToSequenceType(edge);

return sequence;}

Specification 1a(3): Mapping AD Action to CSM StepType: Business layer

mapping actionToStepType(in action:Action): StepType{

//Specification 1a(3.1): Mapping AD OpaqueAction to CSM StepType

step:= action.objectsOfType(OpaqueAction) → map opaqueActionToStepType(action);

step:= action.objectsOfType(AcceptEventAction) → map acceptEventActionToStepType(action);

step:= action.objectsOfType(SendObjectAction) → map sendObjectActionToStepType(action);

step:= action.objectsOfType(SendSignalAction) → map sendSignalActionToStepType(action);

step:= action.objectsOfType(CallBehaviorAction) → map callBehaviorActionToStepType(action);

step:= action.objectsOfType(CallOperationAction) → map callOperationActionToStepType(action);

return step; }

Specification 1a(4): Mapping AD *StructuredActivityNode* to CSM ScenarioType: Business layer

mapping actionToStepType(in activityNode: StructuredActivityNode): ScenarioType{

scenario:= activityNode.objectsOfType(ConditionalNode) → map

conditionalNodeToScenarioType(activityNode);

scenario:= activityNode.objectsOfType(ExpansionRegion) → map

expansionRegionToScenarioType(activityNode);

Specification 1a(4): Mapping AD *StructuredActivityNode* to CSM ScenarioType: Business layer

scenario:= activityNode.objectsOfType(LoopNode) → map
loopNodeToScenarioType(activityNode);
scenario:= activityNode.objectsOfType(SequenceNode) → map
sequenceNodeToScenarioType(activityNode);
return scenario; }

Specification 1a(5): Mapping AD *ObjectNode* to CSM StepType: Business layer

mapping objectNodeToStepType(in objectNode:ObjectNode): StepType{
step:= objectNode.objectsOfType(ActivityParameterNode) → map
activityParameterNode ToStepType(objectNode);
step:= objectNode.objectsOfType(CentralBufferNode) → map
centralBufferNodeToStepType(objectNode);
step:= objectNode.objectsOfType(DataStoreNode) → map
dataStoreNodeToStepType(objectNode);
step:= objectNode.objectsOfType(ExpansionNode) → map
expansionNodeToStepType(objectNode);
return step; }

All specifications presented before are describing the mappings between the elements of the UML-AD and the CSM at a high level of abstraction. Further specification details are presented in the technical report [ALH14], which specifies the mapping between the properties of the UML-AD and CSM elements. We present below two of the mapping specifications described in [ALH14]: a) the mapping from the UML *OpaqueAction* to the CSM *StepType*, and b) the mapping from the *UML ActivityPartition* to the *CSM ComponentType*.

The first mapping between the UML *OpaqueAction* to the CSM *StepType* is defined in Specification 1a(3.1). The specification begins with a constructor command which creates an instance of a *StepType*. The parameters of the constructor are assigned to the step instance (if

they are not null). The stereotyped of the *OpaqueAction* element adds more details to the step element. Two stereotypes are defined in the algorithm: the *PaStep* and the *GaWorkloadEvent* which is defined to the first action in the activity model.

Specification 1a(3.1): Mapping AD OpaqueAction to CSM StepType: Business layer

```
mapping opaqueActionToStepType(in action: OpaqueAction): StepType {
 constructor Metadata::actionTypeMetadata(){name:=_ACTIONTYPE, value=opaque};
 constructor StepType::step(i:String, ra:ResourceAcquireType, rr:=ResourceReleaseType,
r:=RefinementType, c:ComponentType, p:ScenarioElementType, s:ScenarioElementType) {id:=i,
 name=action.name, resourceAcquire:=ra, resourceRelease:=rr, refinement:=r,
component:=c, predecessor:=p, successor:=s, metadata:= actionTypeMetadata } ;
 //When OpaqueAction is stereotyped with PaStep
 if(action.isStereotyped("PaStep"){
    object step:StepType{
       hostDemand=: action.getHostdemand();
       probability=: action.getProp();
       repCount:= action.getRep();
        action.getBehavDemands() → forEach(behavDemand) {
          constructor StepType::step(i:String){id:=i, name:=behavDemand, repCount:=behavCount}; }
        action.getServDemands() → forEach(servDemand) {
          constructor StepType::step(i:String){id:=i, name:=servDemand, repCount:=servCount}; }
        action.getExtOpDemands() → forEach(extOpDemand) {
          constructor StepType::step(i:String){id:=i, name:=extOpDemand, repCount:=extOpCount}; }
      //the rest of the properties are mapped in a similar way.
    }   }
  //When OpaqueAction is the first action element in workflow model.
 if(action.isStereotyped("GaWorkloadEvent"){
   constructor WorkloadType::worload(i:String) {id:=i} ;
       object workload:WorkloadType{
       arrivalPattern=: action.getPattern();
       externalDelay=: action.getExternalDelay();    }
     if(action.pattern.contains(Closed){
```

Specification 1a(3.1): Mapping AD OpaqueAction to CSM StepType: Business layer

```
            population:= action.getPattern().getPopulation(); }
        var start:= step.getParent().getStart();
        start.add(workload); }
  return step;   }
```

The second mapping from the *UML ActivityPartition* to the *CSM ComponentType* is defined in specification 1a(6). The specification begins with a constructor command which creates an instance of a *ComponentType*. The parameters of the constructor are assigned to the component instance (if they are not null). Three stereotypes are defined in the algorithm: the *PaRunTInstance,* the *internal* which represents the orchestration workflow and the *external* which represents the choreography workflow.

Specification 1a(6): Mapping AD ActivityPartition to CSM ComponentType: Business layer

```
mapping activityPartitionToComponentType(in partition: ActivityPartition): ComponentType {
    constructor ComponentType::component(i:String, d:String, a:Boolean, s:String) {id:=i,
    description:=d, isActiveProcess:=a, schedPolicy:=s} ;
    if(partition.isStereotyped("PaRunTInstance"){
        object component:ComponentType{
            host=: partition.getHost();
            instance=: partition.getName();
            multiplicity=: partition.getPoolSize();
      //the rest of the properties are mapped in a similar way.
        }   }
    if(partition.isStereotyped("internal"){
        constructor Metadata::orgTypeMetadata(){name:=_ORGTYPE, value=internal}
        object component:ComponentType{
            metadata:= orgTypeMetadata;    }   }
    if(partition.isStereotyped("external"){
        constructor Metadata::orgTypeMetadata(){name:=_ORGTYPE, value:=external}
```

Specification 1a(6): Mapping AD ActivityPartition to CSM ComponentType: Business layer

```
object component:ComponentType{
    metadata:= orgTypeMetadata;   }   }
return component;   }
```

The CSM scenario is described as a collection of steps linked together by connectors. After each Step is mapped to the CSM model, a link is established between this step and the preceding/succeeding pathconnections. A Step defines two association roles: a) a *predecessor* which identifies the preceding connector that the *step* is linked with, and b) a *successor* which identifies the succeeding connector that the *step* is linked with. A *pathconnection* also defines two association roles: a) a *source* which identifies the list of the preceding steps that the *pathconnection* is linked with and b) a *target* which identifies the list of the succeeding steps that the a *pathconnection* is linked with. The number of the *source* and *target* depends on the kind of the *pathconnection*. For example, the *Start* is linked to one target *Step*, the *End* is linked to one source *Step*, the *Sequence* is linked to one target *Step* and one source *Step*, the *Fork* and *Branch* are linked to one source *Step* and two or more target *Steps*, the *Join* and *Merge* are linked to one or more source *Steps* and one target *Step* [PET07].

Specification 1a(7) describes how the links are established between two steps and different kind of pathconnections.

Specification 1a(7): Establishing links between two steps and a pathconnection

```
helper CSMType::establishingLinks(in precedStep: StepType, in succeedStep: StepType, in connector:
PathConnectionType) {
    //Linking Start to Step
    If(connector.objectsOfType(StartType)){
      succeedStep.getPredecessors().add(connector);
      connector.getTargets().add(succeedStep);  }
```

Specification 1a(7): Establishing links between two steps and a pathconnection

```
//Linking Step to End
If(connector.objectsOfType(EndType)){
   precedStep.getSuccessors().add(connector);
   connector.getSources().add(precedStep); }
//Linking Step to (Sequence, Fork, Join, Branch or Merge) to Step
If(connector.objectsOfType(SequenceType) or connector.objectsOfType(ForkType) or
connector.objectsOfType(JoinType) Or connector.objectsOfType(BranchType) Or
connector.objectsOfType(MergeType)){
   succeedStep.getPredecessors().add(connector);
   connector.getTargets().add(succeedStep);
   precedStep.getSuccessors().add(connector);
   connector.getSources().add(precedStep); } }
```

The next specification describes the details of *interactionToSubScenarios* mapping presented in specification 1. In specification 1b, the elements of the service behavior models, which are represented as UML-SD, are mapped to the elements of the CSM Subscenario models.

The UML sequence diagram elements (such as *CombinedFragment*, *InteractionUse* and *Continuation*) are mapped to CSM *ScenarioType*. The *Message*, *ExecutionSpecification* and *Event* elements are mapped to CSM *StepType*. The *LifeLine* is mapped to *ComponentType* and *MessageSort* is mapped to *MessageKind*. The kind of *CombinedFragment* depends on an operator selected from an enumeration called *InteractionOperatorKind*. An *InteractionOperatorKind* enumeration contains different operators (such as parallel, alternative, loop, optional) which are used to define the type of the *CombinedFragment*. The type of the operator is used during the mapping to define the *PathconnectionType*; i.e., *Fork*, *Join*, *Branch*, *Merge*, between the operands of the *CombinedFragment*.

Specification 1b: Mapping UML elements to CSM elements: Service layer

```
mapping Interaction:: interactionToSubScenarios(in uml:UML) :ScenarioType
    when { uml.getPackagedElement() <> null} {
        uml.getPackagedElement()→forEach(interaction){
            subscenario:= map interactionToScenario(interaction)
            interaction.getOwnedElement()→forEach(ownedElement){
            scenario:= ownedElement.objectsOfType(CombinedFragment)→map
            combinedFragmentToScenarioType(ownedElement)
            scenario:= ownedElement.objectsOfType(InteractionUse)→map
            interactionUseToScenarioType(ownedElement)
            scenario:= ownedElement.objectsOfType(Continuation)→map
            continuationToScenarioType(ownedElement);
            step:= ownedElement.objectsOfType(Message)→map
            messageToStepType(ownedElement, subscenario)
             //Specification 1b(6): mapping the SD Event to StepType
            step:= ownedElement.objectsOfType(Event)→map
            eventToStepType(ownedElement)
            component:= ownedElement.objectsOfType(LifeLine)→map
            lifeLineToComponentType(ownedElement);
            end:= map endInteractionToEndtype(ownedElement)    }    }
return subscenario }
```

Specification 1b(6): mapping the UML Event to StepType

```
mapping Interaction::eventToStepType(in event:Event){
    step:= event.objectsOfType(CreationEvent)→map creationEventToStepType(event);
    step:= event.objectsOfType(DestructionEvent)→map destructionEventToStepType(event);
    step:= event.objectsOfType(ReceiveOperationEvent)→map
    receiveOperationEventToStepType(event);
    step:= event.objectsOfType(ReceiveSignalEvent)→map receiveSignalEventToStepType(event);
    step:= event.objectsOfType(SendOperationEvent)→map sendOperationEventToStepType(event);
   step:= event.objectsOfType(SendSignalEvent)→map sendSignalEventToStepType(event);
return step;  }
```

All specifications presented before describe the mappings between the elements of the UML-SD and the CSM Subscenarios at a high level of abstraction. Further specification details are presented in the technical report [ALH14], which specifies the mapping between the properties of the UML-SD and CSM Subscenario elements. We present below two of the mapping specifications described in [ALH14]: a) the mapping between the UML *CombinedFragment* of type parallel to the CSM *ScenarioType*, b) the mapping between the *UML Lifeline to the CSM ComponentType*.

The first mapping between the UML *CombinedFragment* of type parallel and the *CSM ScenarioType* begins with two constructor commands which create an instance of a *ForkType and JoinType*. The *CombinedFragment* with a parallel operator is mapped into Fork/Join behaviour, where the operands are first forked for parallel execution and then joined back to the main path. A *PaStep* stereotype is defined where the *noSync* property is used to identify an operation which is forked but never joins to the main path.

Specification 1b(2.3): mapping the SD CombinedFragment of type parallel to CSM ScenarioType

```
mapping Interaction:: parCombinedFragmentToScenarioType(in fragment:CombinedFragment, in
mainScenario:ScenarioType):ScenarioType{
    constructor ForkType:fork(i:String) {id:=i};
    constructor JoinType:join(i:String) {id:=i};
    mainScenario.add(fork);
   mainScenario.add(join);
   fragment. getOperands()→forEach(operand){
        constructor ScenarioType:scenario(i:String) {id:=i, name:=operand.getName()};
       if(operand.isStereotyped("PaStep"){
            object scenario:ScenarioType{
               propability=: operand.getProp();
               repCount=: operand.getRep();
          if(operand.getNoSynch() = true){
```

Specification 1b(2.3): mapping the SD CombinedFragment of type parallel to CSM ScenarioType

```
            constructor EndType::end(i:String) {id:=i};
            scenario.getSuccessor().add(end);   }
      //the rest of the properties are mapped in a similar way.
        }   }
        scenario.getPrecessor().add(fork);
        scenario.getSuccessor().add(join);
        mainScenario.add(scenario);    }
  return mainScenario;    }
```

The second mapping between the *UML Lifeline* and *the CSM ComponentType* begins with a constructor command which creates an instance of a *ComponentType*. The parameters of the constructor are assigned to the component instance (if they are not null). A *PaRunTInstance* stereotype is defined in the specification.

Specification 1b(7): mapping the SD LifeLine to CSM ComponentType: Service layer

```
  mapping Interaction:: lifeLineToComponentType(in lifeline:LifeLine):ComponentType{
    property:= lifeline.getRepresents();
    constructor ComponentType::component(i:String, d:String, a:Boolean, s:String) {id:=i,
    name:=property.getName(), description:=d, isActiveProcess:=a, schedPolicy:=s} ;
    if(lifeline.isStereotyped("PaRunTInstance"){
       object component:ComponentType{
          host=: lifeline.getHost();
          instance=: lifeline.getName();
          multiplicity:= lifeline.getPoolSize();
    //the rest of the properties are mapped in a similar way.
       }   }
    return component;    }
```

Tables 4.1 and 4.2 present the graphical mapping between UML and CSM metamodels.

Table 4.1: The graphical mapping of UMLAD2CSM

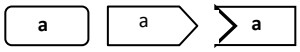
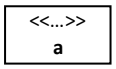
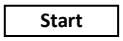
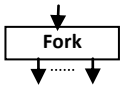
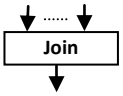
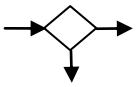
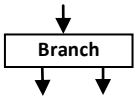
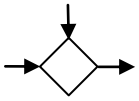
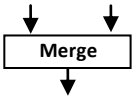
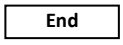
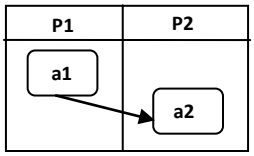
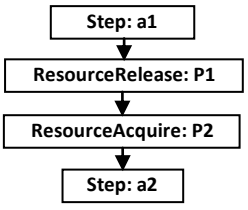
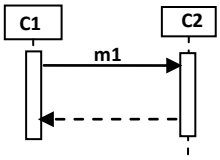
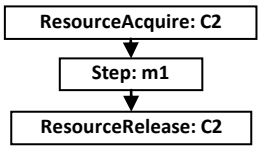
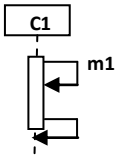
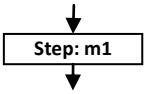
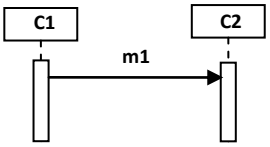
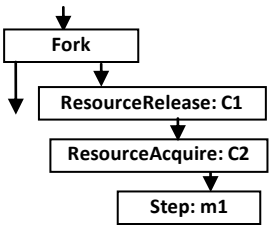
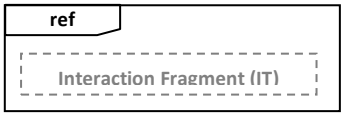
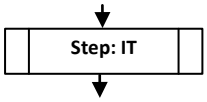
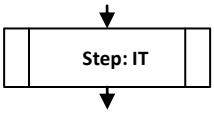
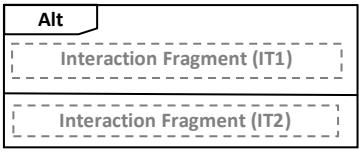
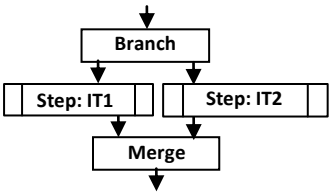
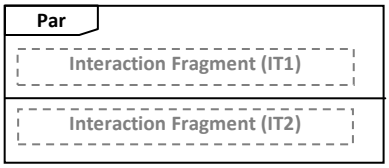
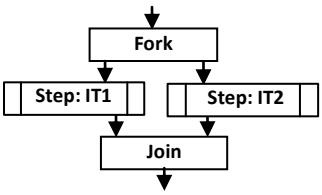
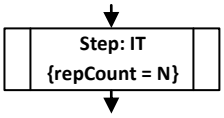
Elements of Activity Diagram		Elements of CSM Diagram
Action {OpaqueAction, AcceptEventAction, SendObjectAction, SendSignalAction, CallBehaviorAction, CallOperationAction} 		*StepType* 
ObjectNode { ActivityParameterNode, CentralBufferNode, DataStoreNode, ExpansionNode} 		*StepType* 
StructuredActivityNode { ConditionalNode, ExpansionRegion, LoopNode, SequenceNode} 		*ScenarioType* (refined) 
ActivityEdge {ControlFlow, ObjectFlow} 		Sequence 
ControlNode	*InitialNode* 	*StartType* 
	ForkNode 	ForkType 
	JoinNode 	*JoinType* 
	DecisionNode 	*BranchType* 
	MergeNode 	*MergeType* 
	FinalNode 	*EndType* 
Control flow cross Partition P1 to Partition P2 		CSM order: Step a1, the resource P1 is released, the resource P2 is acquired, Step a2 is called 

Table 4.2: The graphical mapping of UMLSD2CSM

Elements of Sequence Diagram		Elements of CSM Diagram
Synchronous Message: Component C1 sends a message to component C2		
Synchronous message: local message		
Asynchronous message: Component C1 sends A message to component C2		
Interaction use Fragment		
Combined Fragment — Option combined fragment		
Alternative Combined Fragment		
Parallel Combined Fragment		
Loop Combined Fragment		

4.2.2 Transformation of MARTE stereotypes to CSM

The UML profile for Modeling and Analysis of Real Time and Embedded systems (MARTE) defines ten sub-profiles which extend UML for different analysis domains. In our proposed approach (PUMA4SOA), we are using three of these profiles: a) the Generic Quantitative Analysis Modeling (GQAM) which supports the analysis with quantitative extension units to determine the performance capability of a system with non-deterministic behavior, b) the Generic Resource Modeling (GRM) which add the general platform concepts to the software model, c) the Performance Analysis Modeling (PAM) which defines the extension units; i.e., stereotypes, that are used to evaluate the performance of soft real time embedded systems. The results generated from the performance analysis are statistical output measures; i.e., mean of response time, delay, throughput, service time [OMG11a]. The following are stereotypes of GQAM, GRM and PAM which are used to annotate the UML input design models in PUMA4SOA:

GQAM::GaStep::PaStep: represents a unit of an operation.

GQAM::GaStep::PaStep::PaCommStep: represents the conveyance of a message.

PaRunTInstance: represents a run-time instance of a process resource.

GRM::Resource::PaLogicalResource: represents a resource which can be acquired and released by the *AcqStep* and *RelStep* respectively. It can be a single unit resource, a buffer pool, or a token access pool [OMG11a].

GQAM::GaStep::PaResPassStep: it is applied after a fork to indicate that the resource which acquired before the fork is passed to the annotated branch.

GQAM::GaStep::PaStep::PaRequestedService: it is a special type of steps, which indicates an operation of an interface.

VSL::Expressions::GaAnalysisContext: represents the global variables which are defined within the model context.

GaWorkloadEvent: represents the stream of events which trigger the behavior of a scenario.

GRM::ComputingResource::GaExecHost: represents a computational process

GRM::CommunicationMedia::GaCommHost: represents the physical communication link.

GRM::ResourceUsage::GaScenario: represents the behavior at the system level.

GQAM::GaStep:: GaAcqStep: represents a step that acquires a resource.

GaStep:: GaRelStep: represents a step that releases a resource.

GRM::ResourceTypes::ConcurrencyResource::SchedulableResource: represents a kind of *ConcurrencyResource* with logical concurrency [OMG11a].

The MARTE profile is used to annotate the UML design model in order to describe performance characteristics of the system. It extends the UML modeling for describing timing, resources, and quantitative performance and schedulability. The mapping of MARTE stereotypes and their properties to a CSM model occurs during the mapping of the annotated UML design models to CSM models. The mapping between the previous MARTE stereotypes and their equivalence in the CSM metamodel is described in Appendix A.

The mapping between the elements of UML and CSM models is not always performed as a one-to-one relationship. Some of the MARTE elements when annotated to UML elements introduce new features to the behavior model. The mapping of UML element with these features requires defining two or more CSM elements. We define four MARTE Stereotypes which add new features to the UML behavior: a) *PaCommStep››*, b) *PaStep*, c) *GaAcqStep* and d) *GaRelStep*.

The ‹‹*PaCommStep*›› stereotype is a special kind of ‹‹*PaStep*›› which represents message conveyance [OMG11a]. Mapping an element annotated with ‹‹*PaCommStep*›› in the UML design models (such as *ControlFlow* or *Message*) to CSM model introduces three steps: the *TxOhStep, RcvOhStep* to describe the overhead delays at the sender and the receiver respectively, and the *CommStep* to describe the network delay. Figure 4.2 shows the model transformation of a message m1 annotated with *PaCommStep*.

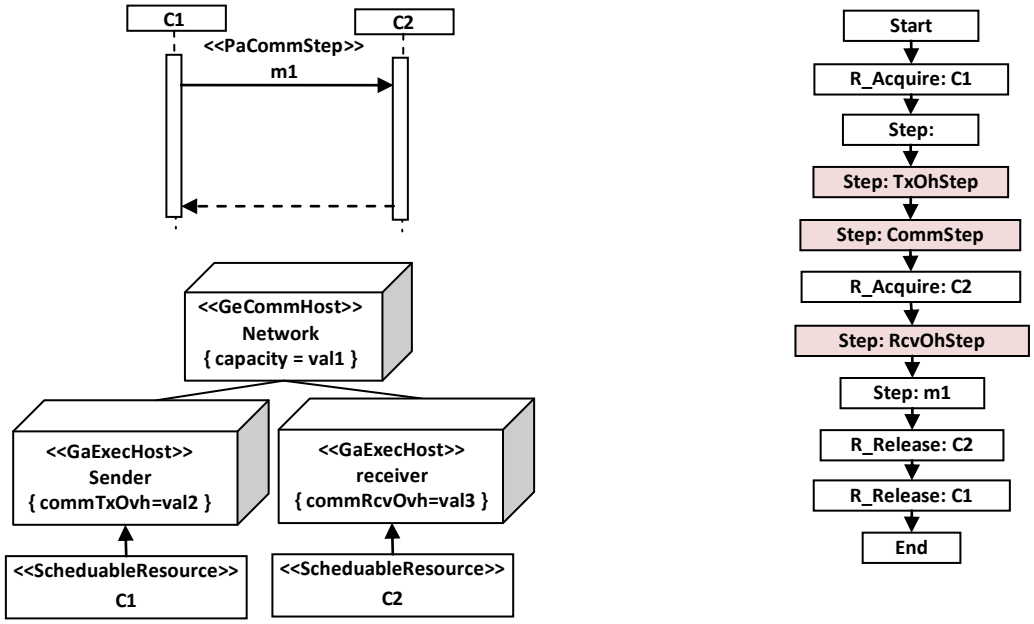


Figure 4.2: The mapping of UML elements annotated with *PaCommStep* to CSM Model

The *TxOhStep* step is defined at the sender side to represent the delay of the transmission overhead. The *hostDemand* property of the *TxOhStep* = *commTxOvh*, where the *commTxOvh* is a property of the Node *sender* in the deployment diagram. The step *RcvOhStep* is defined at the receiver side to represent the delay of the receiving overhead. The *hostDemand* property of the *RcvOhStep* = *commRcvOvh*, where the *commRcvOvh* is a property of the Node *receiver* in the deployment diagram. The step *CommStep* is defined at the receiver side to represent the network delay. The *hostDemand* property of the *CommStep* = *msgSize/capacity*, where the *msgSize* is a property of the Message *m1* in the sequence diagram which defines the size of the transmitted

message, and the *capacity* is a property of the communication Node *Network* in the deployment diagram.

The ‹‹*PaStep*›› stereotype and its specialized steps, such as ‹‹*PaCommStep*›› and ‹‹*PaRequestedService*››, define a *noSync* property to identify an operation which is forked but never joins to the main path. The property *noSync* has a Boolean value with a default equal to *false*. Figure 4.3 describes the model transformation of the *noSync* property to CSM. A parallel combined fragment defines two interaction fragments IT1 and IT2 and both are annotated with ‹‹*PaStep*››. The *noSync* property of IT1 has a default value (false), while for IT2 the *noSync* property is true. Mapping the parallel combined fragment to CSM shows a fork with two paths: a refined step IT1 which join to the original path and another refined step IT2 with an end path.

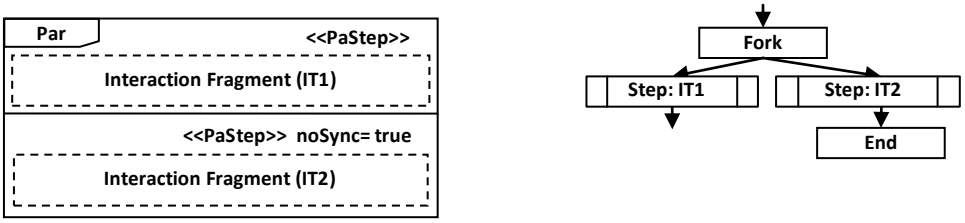


Figure 4.3: The mapping of UML elements annotated with *PaStep{ noSync }* to CSM Model

The ‹‹*PaStep*›› also defines six properties which describe the scenarios, services, and external operations that are called during an execution of an operation (such as a message or action). These properties are defined in three groups [OMG11a]:

1. *behavDemands*: defines the scenarios invoked while executing the operation (annotated by ‹‹*PaStep*››) and *behavCount*: defines the number of calls made to execute each scenario in the *behavDemands*.
2. *servDemand*: defines the other operations which have been called while executing the operation (annotated by ‹‹*PaStep*››) and *servCount*: defines the number of calls which are made to execute each operation in the *servDemand*.

3. *extOpDemands*: defines the interfaces of the external services which are called by the operation (annotated by ‹‹*PaStep*››) and *extOpCount*: defines the number of calls which are made to execute each interface in the *extOpDemands*.

The properties, *behavDemands*, *servDemands*, or *extOpDemands,* are defined as a list of values; and each element in the list is modeled as a *CSM Step*. Each number in the lists of *behavCount, servCount, extOpCount* represents the *repCount* value for its step in the previous demand lists.

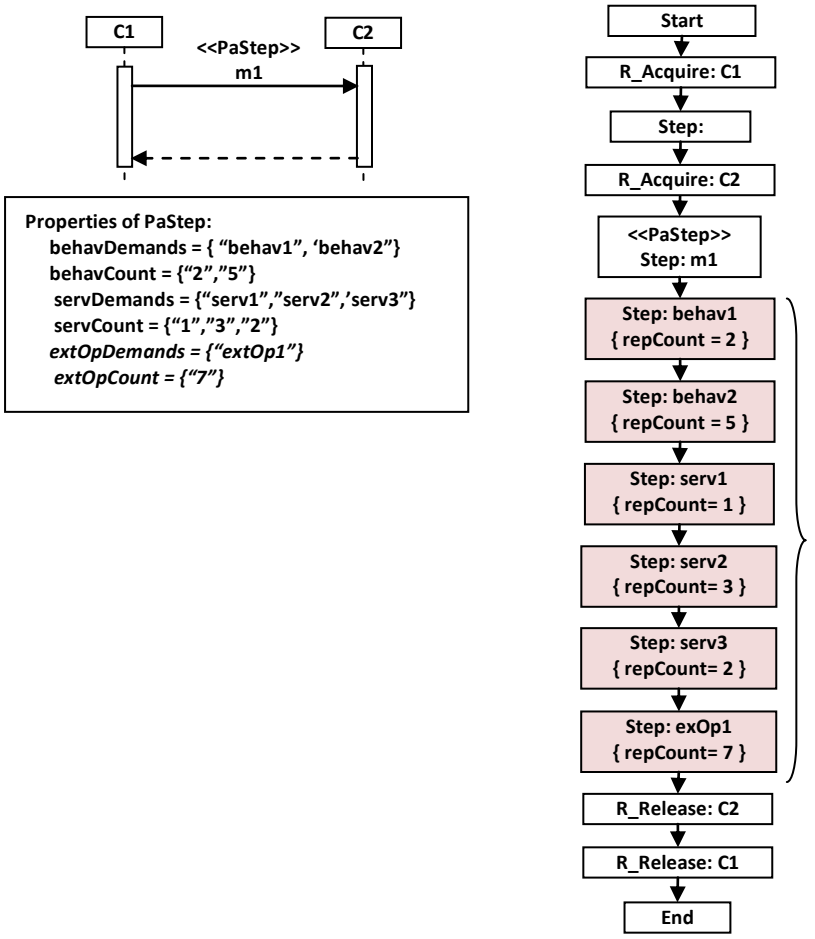


Figure 4.4: The mapping of UML elements annotated with *PaStep{ behavDemands, behavCount, ServDemands, ServCount, extOpDemands, extOpCount }* to a CSM Model

The demand groups are modeled in CSM randomly, and the default value of the *repCount* is one (if it is not defined by the modeler). Figure 4.4 describes how the three demand groups are modeled in the CSM.

The ‹‹*GaAcqStep*›› and ‹‹*GaRelStep*›› stereotypes are used to acquire and release logical resources, which are embodied as a software process, such as a mutex, exclusive lock. We updated the CSM metamodel with a new meta-class to represent the *LogicalResourceType* (see Figure 2.1 on page 23). The logical resource is defined in the deployment diagram and stereotyped with ‹‹*SchedulableResource*››. Figure 4.5 describes a sequence diagram where write and stop messages are stereotyped with ‹‹*GaAcqStep*›› and ‹‹*GaRelStep*›› respectively. The logical resource is *execLock* which is acquired on the write message and released on the stop message. In the CSM, new *ResourceAcquire* and *ResourceRelease* of execLock are created.

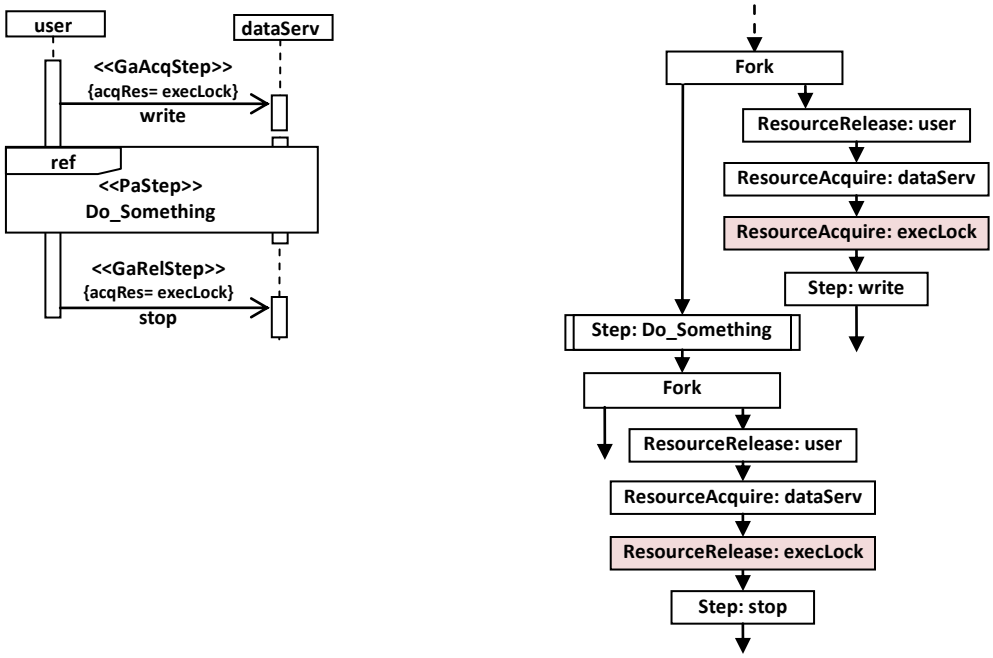


Figure 4.5: The mapping of UML elements annotated with *GaAcqStep* and *GaRelStep* to CSM Model

4.2.3 UML2CSM: summary of differences between PUMA4SOA and PUMA

PUMA4SOA is extended from the original PUMA to address new capabilities and features. These capabilities allow PUMA4SOA to apply model transformations on SOA systems. As we mentioned before that the UML design models of PUMA4SOA consist of different views: a workflow model using UML-AD, a service architecture model using SoaML, a service behavior diagram using UML-SD and a UML deployment diagram. In PUMA, the UML design model describes the software application using either UML-SD or UML-AD or both, and a UML deployment diagram for configuration.

During UML2CSM model transformation, PUMA4SOA separates the mapping of the business layer, represented by the workflow model, from the service layer, represented by the service architecture and behavior models, by assigning the CSM top scenario model to the workflow and the service to the subscenarios associated within the CSM top scenario. In PUMA, the mapping does not require the separation because the input model describes one layer, the software behavior models.

In PUMA4SOA, we also extended the CSM metamodel with the concept of *metadata*. *Metadata* is a light weight profiling mechanism like the one used in URN [MOS11] which extends a CSM metamodel by defining name-value pairs to capture stereotypes. The metadata mechanism can be used to manage the complexity of the software development, allowing for a dynamic data warehouse environment and reducing the cost of the modeling evolution. The main reason for adding metadata to the CSM schema is to provide a convenient way to define the point-cut rules at the CSM level. Metadata can also be used, as an extension mechanism, to tag the CSM elements with additional features and properties. More details are presented in Chapter 5.

Aspect Oriented Modeling (AOM) is also used in PUMA4SOA to allow software designers to address separately solutions for crosscutting concerns [ALH08]. In PUMA4SOA, we use AOM for platform models and for transforming platform independent model (PIM) to a platform specific model (PSM). The separation of platform models from the application model allows reusing the performance characteristics of different platforms, making the building of the performance model more efficient. Our proposed approach provides the ability to transform PIM to PSM based on the AOM approach at three modeling levels, the UML, CSM and LQN.

Using a PC feature model is another extension provided by PUMA4SOA. The PC feature model explicitly captures the variability in platform choices, execution environments, different types of communication realizations, and other external factors with an impact on performance [PET12]. The PC feature model represents a classification of the different SOA platform aspects and of their relationship in a hierarchical format [KAN90]. The modeler can select from the PC feature model multiple SOA platform aspects needed for the application according to the business requirements.

In PUMA4SOA, the traceability between the elements of the UML and the CSM is also maintained during the model transformation. This has been achieved by creating a traceability metamodel for PUMA4SOA where trace-links are applied between the elements of the source and the target model. Traceability between UML and CSM can be used to define the dependencies between their elements and to propagate changes of properties from the UML models to the CSM model. More details are presented in Chapter 6.

4.2.4 Transformation from CSM to LQN

The CSM model of PUMA4SOA defines two modeling layers: the business layer represented by the CSM top scenario model, and the service layer, represented by the CSM sub-

scenarios containing services. During PUMA4SOA model transformation, the CSM top scenario is transformed into an LQN *ActivityGraph*. The *ActivityGraph* is owned by a *Task* which represents the engine that runs the workflow, such as BPEL. This *Task* is deployed into the client processor. The CSM subscenarios describing service operations, contained in the service layer, are transformed into LQN entries owned by tasks, which in turn represent the components offering the services. Those tasks are deployed on different processors. The entries, tasks and processors are mapped from CSM steps, components and processing resources respectively.

Model transformations from CSM to LQN are presented in many researches. In [WOO13], the specifications of mapping CSM to LQN in PUMA, and the challenges which might cause the mapping inconsistent, are discussed. In our research, we do not specify a complete model transformation from CSM to LQN, since this part of the research is dealt with other research. We only specify the specifications that handle limited choices of CSM elements and handle the separation between the mapping of the business layer and the service layer. The model transformation specifications are based on a simplified template of an LQN model in XML format [FRA12] presented in Figure 4.6.

```
<lqn-model>
   <solver-params>
      <pragma/>
   </solver-params>
   <processor>
      <task>
         <entry>
            <entry-phase-activities>
               <activity>
                  <synch-call/>
                  <asynch-call/>
               </activity>
               <activity> ... </activity>
            </entry-phase-activities>
         </entry>
         <entry> ... </entry>
         <task-activities>
            <activity/>
            <precedence/>
         </task-activities>
      </task>
      <task> ... </task>
   </processor>
   <processor> ... </processor>
/lqn-model>
```

Figure 4.6: The XML file layout of the LQN Model

The mapping between the Top CSM and *ActivityGraph* is described in Specification 2a (*topCSMToActivityGraph*). The mapping begins by creating a reference task, where the processor, the task and the entry are named *"user"*. After creating the reference task, the CSM top scenario model is mapped into an *ActivityGraph* and the steps are mapped into activities connected by precedence elements. The CSM components are mapped into tasks and the processing resources are mapped into processors.

Specification 2a: Mapping CSM elements to LQN elements: Business layer

```
mapping CSMType::topCSMToActivityGraph(in csm:CSMType) :LQNModel
when{csm.getScenarios() <> null} {
   topScenario:= csm.getScenarios().get(0);
   constructor LQNModel::lqn(i:String, n:String){id:=I, name:=n};
   // Specification 2a(1): Creating the reference task
   refProcessor:= createReferenceTask();
   lqn.add(refProcessor);
```

Specification 2a: Mapping CSM elements to LQN elements: Business layer

```
//create the task activity graph
activityGraph:= map scenarioToActivityGraph();
topScenario.eContents() → forEach(csmElement){
    processor= csmElement.objectsOfType(ProcessingResource) → map
    processingResourceToProcessor(csmElement);
    task= csmElement.objectsOfType(ComponentType) → map
    componentTypeToTask(csmElement);
    activity:=  csmElement.objectsOfType(StepType) → map stepTypeToActivity(csmElement);
    //Specification 2a(6): Mapping pathconnection to precedence
    csmElement.objectsOfType(PathConnectionType) → map
    activity:= pathConnectionTypeToPrecedence(csmElement, activity);
    activityGraph.add(activity);
    task.add(activityGraph);
    processor.add(task);    }
   lqn.add(processor);
return lqn}
```

Specification 2a(1): Creating ReferenceTask of the LQN model

```
helper LQNModel::createReferenceTask() :Processor{
    //Creating new processor, task, entry, entry-phase-activities, activity and synch−call elements
    constructor Processor::refProcessor(s:float, sc: SchedulingType) {name="user",
   speed−factor:=s, scheduling:=sc, multiplicity:=1, replication:=1};
    constructor Task::refTask(sc: SchedulingType, t:float, p:unsigned)
    { name:="user", multiplicity:=1, replication:=1, scheduling:=sc, think−time:=t, priority:=p ,
    activity−graph:=false};
    constructor Entry::refEntry(o:float, p: integer) {name:="user", open−arrival−rate:=o, priority:=p};
    constructor Entry-Phase-Activities::refPhases(){phase:= 1};
    constructor Activity::refActivity(t:float, h:float, s:float) { think−time:=t, host−demand−mean:=h,
    max−service−time:=s};
    constructor Synch−call::refSynch(d:String, fi:unsigned, fo:unsigned, c:float){dest:=d, fanin:=fi,
```

Specification 2a(1): Creating ReferenceTask of the LQN model

fanout:=fo, calls−mean:=c };
refActivity.add(refSynch);
refPhases.add(refActivity);
refEntry.add(refphases);
refTask.add(refEntry);
refProcessor.add(refTask);
return refProcessor; }

The mapping between the CSM *PathconnectionType* to the LQN *Precedence* is described in specification 2a(6). Each Pathconnection type is mapped into a *Precedence* element with different association roles. The LQN metamodel defines five types of *Precedence* association roles: *SingleActivityList, ActivityList, AndJoinList, OrList* and *ActivityLoopList*. As an example, the CSM *SequenceType* is mapped to a *Precedence* with two associations *pre* and *post* of type *SingleActivityList*. The CSM *JoinType* is mapped to a *Precedence* with two associations *preAND* of type *AddjoinList* and *post* of type *SingleActivityList*. The CSM *StartType* and *EndType* are not mapped to the LQN model; they are used to check for the beginning and finishing of the CSM model.

Specification 2a(6): Mapping the CSM PathConnection to the LQN Precedence

mapping CSMType::pathConnectionToPrecedence(in csmElement:CSMElementType): Precedence{
precedence:= csmElement.objectsOfType(SequenceType)→map
SequenceTypeToPrecedenceType();
Specification 2a(6.2): Mapping the CSM ForkType to the LQN Precedence
precedence:= csmElement.objectsOfType(ForkType)→map ForkTypeToPrecedenceType();
precedence:= csmElement.objectsOfType(JoinType)→map JoinTypeToPrecedenceType();
precedence:= csmElement.objectsOfType(BranchType)→map BranchTypeToPrecedenceType();
precedence:= csmElement.objectsOfType(MergeType)→map MergeTypeToPrecedenceType();
return precedence; }

The mapping between the CSM *ForkType* to the LQN *Precedence* is described in specification 2a(6.2). The LQN *Precedence* of the fork type defines two association roles: a) the *pre* of type *SingleActivityType* which identifies the preceding activity is linked to, b) the *postAND* of type *ActivityList* which identifies the list of succeeding activities is linked to.

Specification 2a(6.2): mapping the CSM ForkType to PrecedenceType

```
mapping CSM:: ForkTypetoPrecedenceType(): Precedence {
    constructor SingleActivityListType::preSingle () {};
    constructor List(ActivityListType)::postActivityList() {};
    constructor Precedence::precedence() {pre:= preSingle, postAnd:= postActivityList};
return precedence; }
```

The mapping between the CSM Subscenario models to the LQN Underlying Tasks is described in specification 2b (*subScenariosToUnderlyingTasks*). The mapping begins with a processor followed by a task, then an entry. The entry might have a multiple phases of activities connected to different precedence types.

Specification 2b : Mapping CSM elements to LQN elements: Service layer

```
mapping CSMType::subScenariosToUnderlyingTasks(in csm:CSMType) :LQNModel
when{csm.getScenarios() <> null} {
    csm.getScenarios() → forEach(scenario){
        when{scenario.getIndex() <> 0} { //Scenrio of index 0 is the top level scenario
            scenario.eContents() → forEach(csmElement) {
                processor:= csmElement.objectsOfType(ProcessingResource) → map
                processingResourceToProcessor(csmElement);
                task:= csmElement.objectsOfType(ComponentType) → map
                componentTypeToTask(csmElement);
                entry:= csmElement.objectsOfType(StepType) → map
                stepTypeToEntry(csmElement);
                constructor Entry-Phase-Activities::phases(p:integer){phase:=p};
                while(--p <=0){
```

Specification 2b : Mapping CSM elements to LQN elements: Service layer

```
constructor Activity::activity(t:float, h:float, s:float) { think−time:=t,
host−demand−mean:=h, max−service−time:=s};
activity:= csmElement.objectsOfType(PathConnectionType) → map
pathConnectionTypeToMakingCallType(csmElement);
activity.add(makingCall);
phases.add(activity); }
entry.add(phases);
task.add(entry);
processor.add(entry); } } }
lqn.add(processor);
return processor; }
```

All specifications which describe the mappings between the elements of the CSM model and LQN model are presented in the technical report [ALH14]. Table 4.4 and 4.5 summarizes the mapping between the CSM to the LQN.

Table 4.3: The CSM2LQN model transformation: Business layer

CSM top Scenario (Business layer)	LQN model (Business layer)
Scenario	*ActivityGraph*
ActiveResource, such as ProcessingResource	*Processor*
PassiveResource, such as Component	*Task*
Step (refined)	*Activity*
Pathconnection { Sequence, Fork, Join, Branch, Merge)	*Precedence*

Table 4.4: The CSM2LQN model transformation: Service layer

<table>
<tr><th colspan="2">CSM subscenarios (Service layer)</th><th colspan="2">LQN model (Service layer)</th></tr>
<tr><td colspan="2">ActiveResource, such as ProcessingResure</td><td colspan="2">Processor</td></tr>
<tr><td colspan="2">PassiveResource, such as Component</td><td colspan="2">Task</td></tr>
<tr><td colspan="2">Step</td><td colspan="2">Entry with an option of multiphases activities</td></tr>
<tr><td colspan="2">Step (refined)</td><td colspan="2">Repeat mapping for the refined elements</td></tr>
<tr><td rowspan="4">Pathconnection::Classifier::Message::MessageKind</td><td>Sync</td><td rowspan="4">MakingCallType</td><td>Synch-call</td></tr>
<tr><td>Async</td><td>Asynch-call</td></tr>
<tr><td>Reply</td><td></td></tr>
<tr><td></td><td>Forwarding</td></tr>
</table>

4.2.5 CSM2LQN: summary of differences between PUMA4SOA and PUMA

During CSM2LQN model transformation, PUMA4SOA separates the mapping of the business layer from the service layer. In PUMA, the mapping does not require the separation because the input model represents one layer. The LQN top level *ActivityGraph* is mapped from CSM top level to represent the business layer, and each element in the CSM top scenario is mapped to its equivalent as in Table 4.4.

Each subscenario in the CSM top scenario is mapped into an activity element. CSM pathconnectors are mapped to its equivalent LQN precedence. We also can have more than one *ActivityGraph* based on number of *external* subscenarios. The *PassiveResources* which are defined within the subscenarios are mapped into tasks and their owned entries are mapped from the acquired steps. The service platform is mapped into entries, which represent the service middleware, owned by tasks. The CSM communication step which represents the network delay is mapped into a delay entry owned by a task representing the network.

In PUMA4SOA, the traceability between the elements of the CSM2LQN and LQN2UML is also maintained during the model transformation chain. We have created a traceability metamodel for PUMA4SOA where trace-links are defined between the elements of the source and the target models. Traceability between CSM2LQN and LQN2UML can be used to define the dependencies between related elements in different models, to propagate changes of properties from one model to another and to analyze the impact of these changes. Traceability between LQN2UML is also used to feed back the performance results from the LQN model to the UML model which is not available in PUMA [ALH13]. More details are presented in Chapter 6.

Chapter 5: Generating PSM using the AOM approach

Chapter 5 presents the algorithms and techniques used in PUMA4SOA to generate the platform specific model (PSM) from the platform independent model (PIM) based on AOM approach. Aspect Oriented Modeling (AOM) techniques are used to produce a Platform Specific Model by weaving platform aspect models of platform operations into the Platform Independent Model. The AOM architecture consists of a primary model, which describes the core design decision, and a set of aspect models, each describing a concern that crosscuts the primary model [PET09a]. The primary model in our approach is the PIM. Platform aspect models are developed to describe the solution proposed in a general way, unrelated to the primary model [FRA04].

Aspect composition in AOM is performed in a number of steps:

1. Defining the point-cut rules (a set of conditions applied to the primary model to identify the join-points) [ALH12a]. Each rule in the point-cut is a constraint related to an operation in the primary model and defined by either automatically by the system design or by the modeler. This means that if a rule is applied on an operation, the return will be either true or false.
2. Identifying the join-points (the locations in the primary model where the point-cut condition is true and the aspect compositions should occur).
3. Instantiating a context specific aspect model, by binding the parametric values of the generic aspect to concrete values related to the context of the join-point in the primary model [ALH08].
4. Performing the aspect composition at the join-points. The composition involves inserting the context specific aspect model at the defined join-points. Three types of insertion are defined: Insert context specific aspect model before the join-point, Insert the context specific aspect model after the join-point, or Replace the join-point by the context specific aspect model.

The chapter begins with Section 5.1 which presents the generating of the platform specific model using AOM approach at three modeling levels: UML, CSM and LQN. Section 5.2 presents the model transformation from the platform independent model (PIM) to the platform specific model (PSM) at the CSM level based on AOM approach.

5.1 Aspect composition at three levels: UML, CSM and LQN

PUMA4SOA provides the ability to transform PIM to PSM based on the AOM approach at three modeling levels: the input design models (UML level), the intermediate model (CSM level) and the performance model (LQN level) [ALH12b]. Although platform aspect models are originally defined in UML, they can be transformed separately from the UML level and composed into the primary model at the other levels: CSM, and LQN [ALH12a]. In the next section, we use the Purchase Order system case study to perform aspect composition at the three modeling levels.

5.1.1 Aspect composition at the UML level

PUMA4SOA provides the flexibility to perform the aspect composition at the UML level, as in Figure 5.1. The design models described in Figure 3.3 to Figure 3.6 (on pages 43 to 47) represent the platform independent model (PIM) and deployment diagram of a purchase order system in UML extended with MARTE stereotypes. The primary model is the platform independent model (PIM). The platform aspect models define the middleware structure and behavior of the selected aspects from the PC feature model [PET12], as shown in Figure 3.7 (on page 49). The feature groups: Operation, Message Protocol, and Realization are mandatory for defining the platform aspect. The operation and message protocol are selected to represent the behavior of the platform aspect, while the realization is selected to represent the configuration of the platform aspect. In our example, we selected a service invocation aspect realized as a

webservice with the message protocol SOAP. Figures 5.2 and 5.3 describe the deployment and behavior views of the service invocation aspect annotated with MARTE stereotypes.

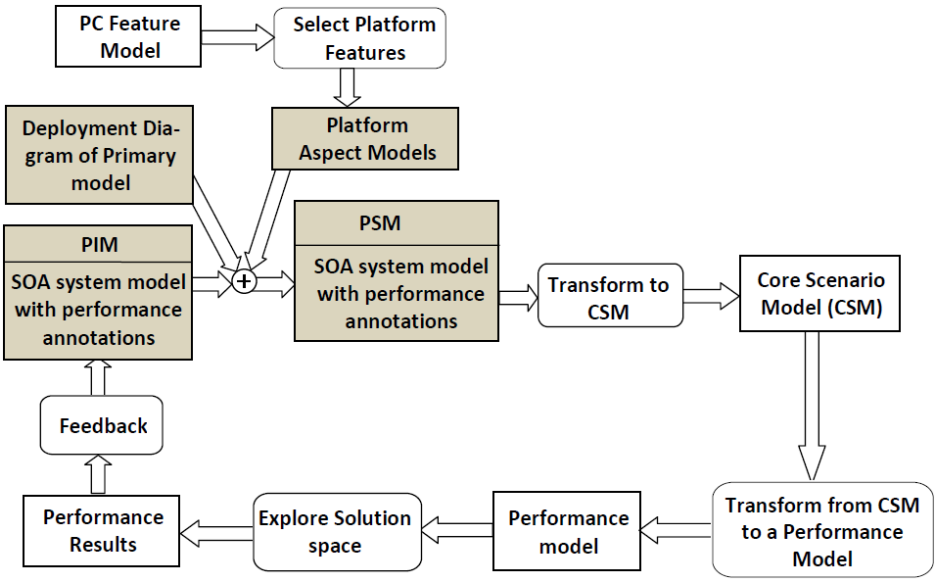


Figure 5.1: PUMA4SOA Approach: Aspect Composition at the UML Level

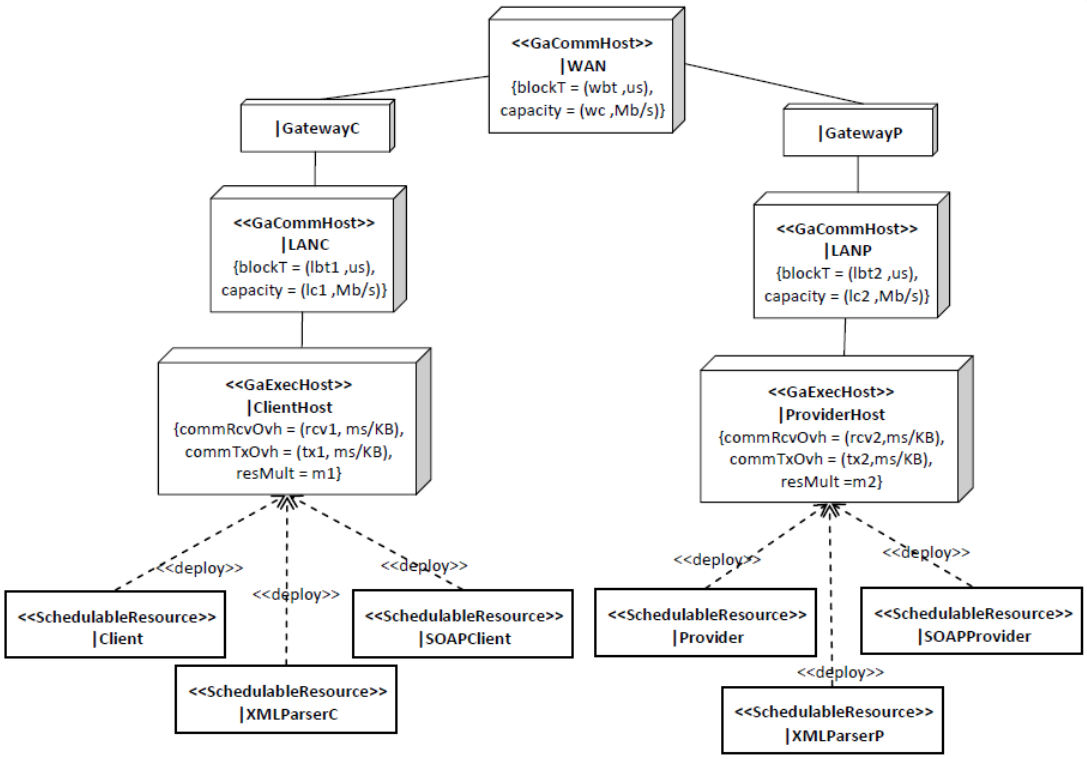


Figure 5.2: The Service Invocation Aspect: Deployment View annotated with MARTE

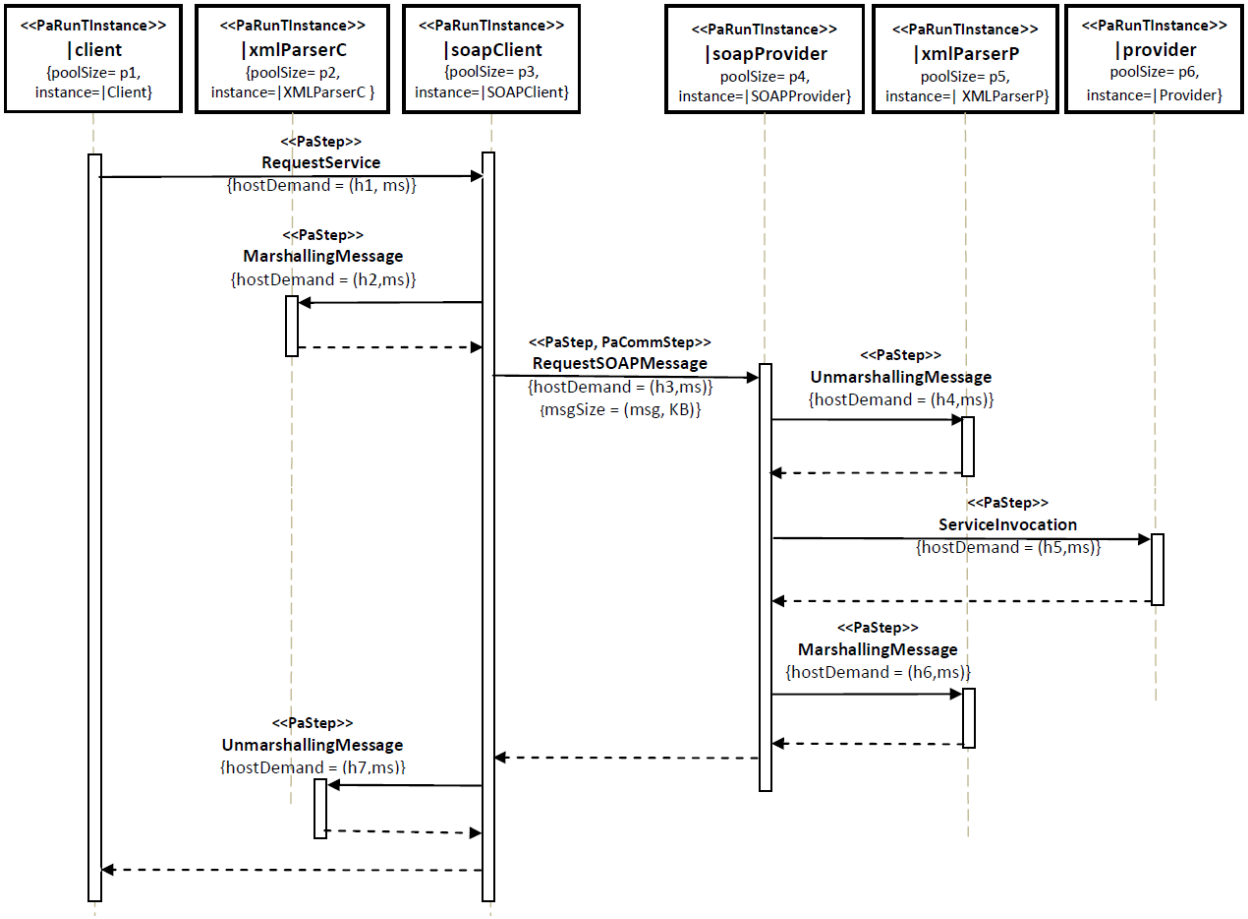


Figure 5.3: The Service Invocation Aspect: Behaviour View annotated with MARTE

The first step in the AOM approach is defining the point-cut rules. Since our proposed aspect (service invocation) applies only to services, the point-cut rules must be defined to locate the service calls in the primary model. Figure 5.4 defines the point-cut rules for Service Invocation. The result of applying the point-cut rules on UML elements of PIM can be either true (if the UML element is of type *Message* and if the *Message* is defined in the service architecture model as an operation of a provided interface owned by a port stereotyped as *«Service»*), or false (otherwise) [ALH12a].

Any UML *Message* element existing in the service behavior models of PIM, such that:

1. *Message* element is expressed in the service architecture model as an operation of a provided interface (of symbol o—).
2. This provided interface is owned by a port stereotyped with *«Service»*.

Figure 5.4: Defining the Point-cut rules for Service Invocation at the UML Level

The next step identifies the join-points by applying the point-cut rules to the PIM. Applying the point-cut rules to the Purchase Order system locates four join-points (UML elements of *Message* type) defined in Table 5.1. The highlighted message (*ScheduleManufacturingPeriod*) of the *ProcessSchedule* in Figure 3.5 (on page 46) is one of the join-points.

Table 5.1: Identifying the Join-points at the UML level

Join-point	Sender's lifeline	Receiver's lifeline	Name of Service Behavior model
InitialInvoiceRequest	*sales*	*invoicing*	*Process Invoice*
GetShippingCost	*invoicing*	*shipping*	*ShipCost*
ScheduleManufacturingPeriod	*sales*	*scheduling*	*ProcessSchedule*
SetShippingDate	*sales*	*shipping*	*ShipProduct*

Once the join-points are identified, a context specific aspect model is generated by instantiating the generic aspect model for each join-point. The instantiating requires binding the generic elements; i.e., roles and messages, to concrete ones and assigning their properties values with fixed ones. First, we bind the generic *Hosts* and *Artifacts* elements in the deployment diagram of the service invocation aspect (Figure 5.2) with the deployment diagram of the PIM (figure 3.6, on page 47). The generic element is bound to newly created element when there is not an equivalent concrete one in the deployment diagram of the PIM.

Table 5.2 represents the binding results, which apply to the join-point *ScheduleManufacturingPeriod*. The join-point *ScheduleManufacturingPeriod* is a service operation which is requested by the *Sales* participant and provided by the *Scheduling* participant; both participants are within the same organization, and they are communicating through the same local area network (*LAN1*) and not through the *WAN* network.

Table 5.2: Binding Generic to Concrete elements (Deployment Diagram)

Generic Aspect			Context Specific Aspect		
Generic elements	**Property**	**value**	**Concrete elements**	**Property**	**value**
|ClientHost	*commRcvOvh*	rcv1	*SalesHost*	*commRcvOvh*	0.15
	commTxOvh	tx1		*commTxOvh*	0.15
	resMult	m1		*resMult*	1
|ProviderHost	*commRcvOvh*	rcv2	*SchedulingHost*	*commRcvOvh*	0.15
	commTxOvh	tx2		*commTxOvh*	0.10
	resMult	m2		*resMult*	1
|LANC	*blockT*	lbt1	*LAN1*	*blockT*	20
	capacity	lc1		*capacity*	100
|LANP	*blockT*	lbt2	*LAN1*	*blockT*	20
	capacity	lc2		*capacity*	100
|WAN	*blockT*	wbt	Not applicable		
	capacity	wc			
|GatewayC			Not applicable		
|GatewayP			Not applicable		
|Client			*Sales*		
|XMLParserC			*XMLParserSales (new)*		
|SOAPClient			*SOAPSales (new)*		
|Provider			*Scheduling*		
|XMLParserP			*XMLParserScheduling (new)*		
|SOAPProvider			*SOAPScheduling (new)*		

We apply the same binding process to the other join-points (*InitialInvoiceRequest*, *GetShippingCost* and *SetShippingDate*) to produce another three tables like Table 5.2. Notice that the generic elements (|*WAN*, |*GatewayC* and |*GatewayP*) are bound when we apply the binding process to the join-point (*GetShippingCost* and *SetShippingDate*), since the provided

participant is *Shipping* which operates externally in another organization (the choreography are defined in the workflow model in Figure 3.3, on page 43).

Next, we bind the generic elements (*messages*, *lifelines*…etc) in the behavior diagram of the service invocation aspect (Figure 5.2) with the service behavior model of PIM (Figure 3.5, on page 46) based on the join-point *ScheduleManufacturingPeriod*. Table 3.3 illustrates the binding of generic elements; some to existing PIM elements and some to newly created elements. Notice that the values of the properties of the new roles, such as *poolSize*, *hostDemand* and *msg*, are undefined. These values can be either estimated or evaluated based on previous simulation experiments or benchmark.

Table 5.3: Binding Generic to Concrete elements (Behaviour Diagram)

Generic Aspect			Context Specific Aspect		
Generic elements	**Property**	**value**	**Concrete elements**	**Property**	**value**
/client	*poolSize*	p1	*sales*	*poolSize*	1
	instance	*/Client*		*instance*	*Sales*
	host	*/ClientHost*		*host*	*SalesHost*
/xmlParserC	*poolSize*	p2	*xmlParserSales* (new)	*poolSize*	?
	instance	*/ XMLParserC*		*instance*	*XMLParserSales*
	host	*/ClientHost*		*host*	*SalesHost*
/soapClient	*poolSize*	p3	*soapSales* (new)	*poolSize*	?
	instance	*/ SOAPClient*		*instance*	*SOAPSales*
	host	*/ClientHost*		*host*	*SalesHost*
/soapProvider	*poolSize*	p4	*soapScheduling* (new)	*poolSize*	?
	instance	*/SOAPProvider*		*instance*	*SOAPScheduling*
	host	*/ProviderHost*		*host*	*SchedulingHost*
/xmlParserP	*poolSize*	p5	*xmlParserScheduling* (new)	*poolSize*	?
	instance	*/ XMLParserP*		*instance*	*XMLParserScheduling*
	host	*/ProviderHost*		*host*	*SchedulingHost*
/provider	*poolSize*	p6	*scheduling*	*poolSize*	1
	instance	*/ Provider*		*instance*	*Scheduling*
	host	*/ProviderHost*		*host*	*SchedulingHost*
RequestService	*hostDemand*	h1	*RequestScheduleManufacturingPeriod* (new)	*hostDemand*	?
MarshallingMessage	*hostDemand*	h2	*MarshallingMessage* (new)	*hostDemand*	?
RequestSOAPMessage	*hostDemand*	h3	*RequestSOAPMessage* (new)	*hostDemand*	?
	msgSize	msg		*msg*	?
UnmarshallingMessage	*hostDemand*	h4	*UnmarshallingMessage* (new)	*hostDemand*	?
ServiceInvocation	*hostDemand*	h5	*ScheduleManufacturingPeriod*	*hostDemand*	0.15

Generic Aspect			Context Specific Aspect		
Generic elements	**Property**	**value**	**Concrete elements**	**Property**	**value**
MarshallingMessage	*hostDemand*	h6	*MarshallingMessage* (new)	*hostDemand*	?
UnmarshallingMessage	*hostDemand*	h7	*UnmarshallingMessage* (new)	*hostDemand*	?

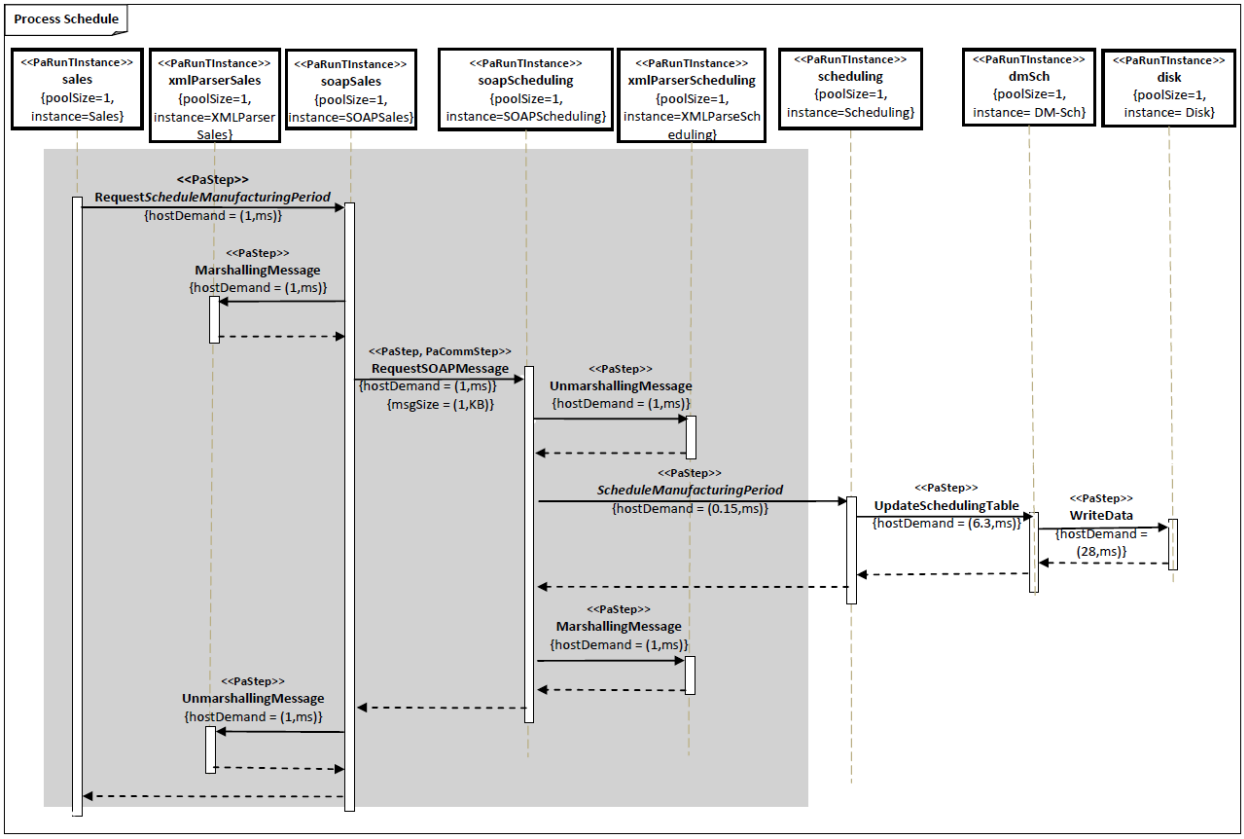


Figure 5.5: The Composed Model for *ProcessSchedule*

After performing the binding process, the context specific aspect model which represents the behavior model can be instantiated. The context specific aspect model is not shown here; however, it appears (the gray region) within the composed model in Figure 5.5.

The aspect composition occurs by replacing the *ScheduleManufacturingPeriod* join-point with the context specific aspect model (gray region). The deployment diagram of the composed model is updated with the new hosts and artifacts introduced in the binding tables. Figure 5.6 shows the updated deployment diagram.

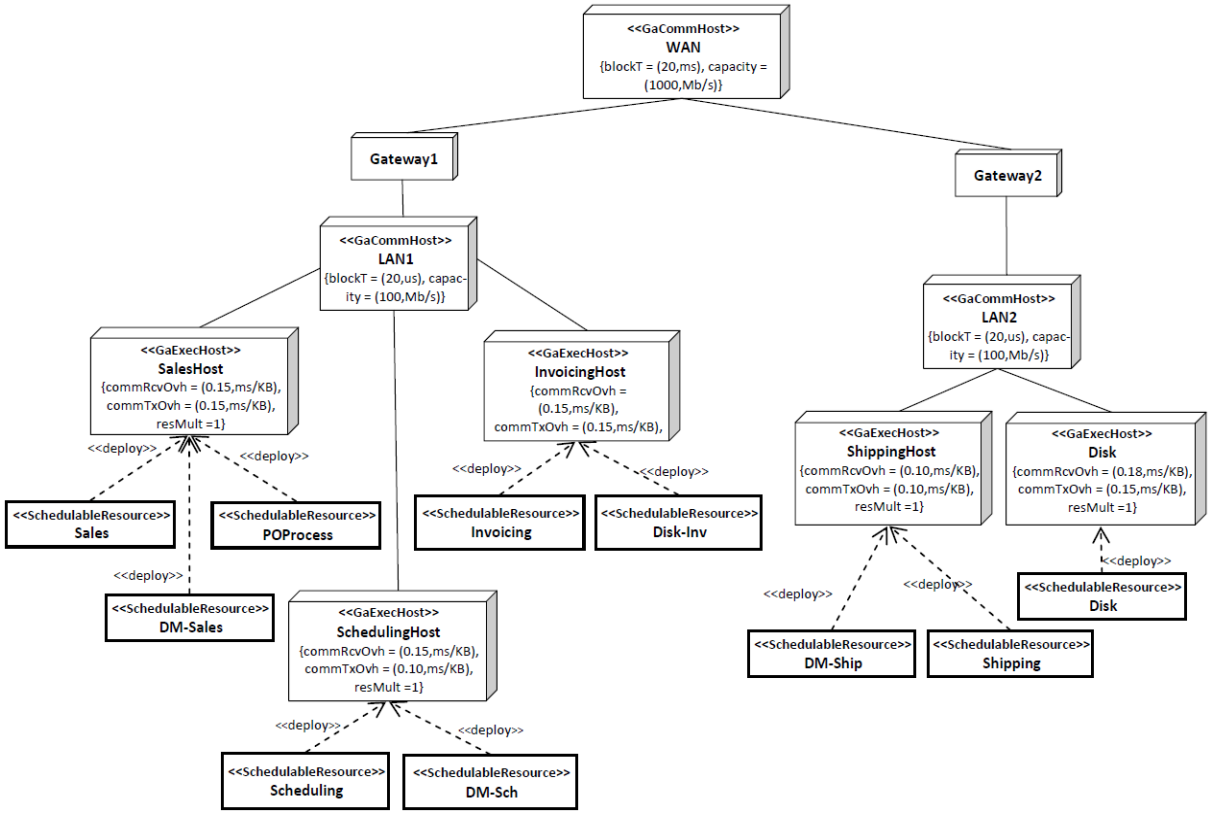
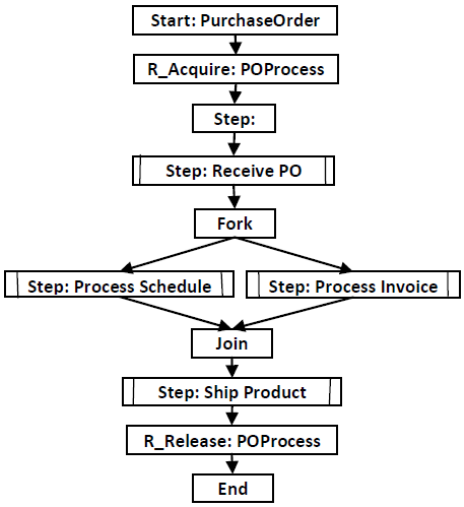


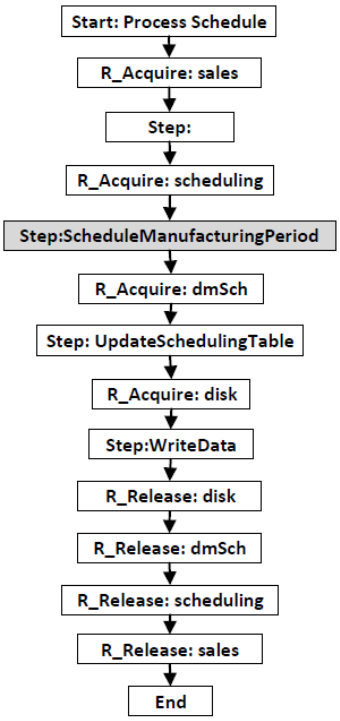
Figure 5.6: The Deployment Diagram for the Composed Model of PO System

5.1.2 Aspect composition at the CSM level

Our proposed PUMA4SOA in Figure 3.2 (on page 39) shows the aspect composition taking place at the CSM level. Figures 5.7 (a, b) describe the CSM model of the platform independent model of our PO system. The business layer, which represents the workflow model at the UML level, is transformed into the CSM top level. The service layer, which represents the service behavior models at the UML level, is transformed into a set of CSM subscenario models that contain the services. The platform aspect model of service invocation described in Figures 5.2 and 5.3 are transformed into the CSM model as in Figure 5.8.



a) The CSM top level describes the workflow



b) The CSM sub-scenario for *ProcessSchedule*

Figure 5.7: The CSM Models for PO System

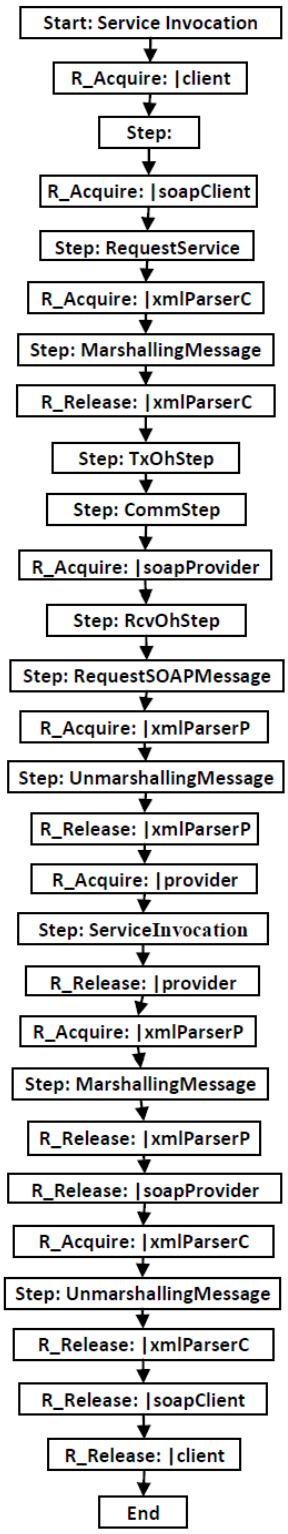


Figure 5.8: The CSM model for Service Invocation Aspect

The aspect composition at the CSM level is performed using the same AOM steps that are being used at the UML level. It is just a different modeling context, where the point-cut rules, the join-points, the binding process and the aspect composition are defined based on the CSM metamodel specification.

The point-cuts conditions cannot be defined at the CSM level, because the details of service architecture model at the UML level is not transformed to CSM model. The CSM model is scenario-based, and its metamodel mainly captures the details of the behavior models, while many of the details of the structure models are lost during the transformation. This is one of the major drawbacks of performing aspect composition at the CSM level [ALH12b].

There are different solutions for this issue, one of them is that we need to use the service architecture model defined at the UML level as in Figure 3.4, (to allocate the service operations of the PO system) in conjunction with the CSM platform independent model (PIM) to define the point-cut rules as in Figure 5.9. The way of defining the point-cut described before is not suitable, since it does not maintain the separation between the models at different modeling levels of PUMA4SOA. In the next section, we present another solution which maintains the separation between the modeling layers, by extending the CSM metamodel with new metadata to capture the missing details of the service architecture model at the UML level.

Any CSM *StepType* element exists in the PIM, such that:

1. *StepType* element is expressed in the service architecture model as an operation of a provided interface (of symbol o—).
2. This provided interface is owned by a port stereotyped with *«Service»*.

Figure 5.9: Defining the Point-cut rules for Service Invocation at the CSM Level

The join-points are identified based on the point-cut rules. Any *StepType* in the CSM model of PIM complies with the point-cut conditions is a join-point as in Table 5.4. The join-point for *ProcessSchedule* subscenario is identified in Figure 5.7(b) as a gray region.

Table 5.4: Identifying the Join-points at theCSM level

Join-point	CSM scenario of PIM model
InitialInvoiceRequest	*ProcessInvoice*
GetShippingCost	*ShipCost*
ScheduleManufacturingPeriod	*ProcessSchedule*
SetSchedulingDate	*ShipProduct*

Instantiating the context specific aspect model is the next step in the AOM approach after identifying the join-points. The instantiation process requires binding the CSM generic elements (such as *Scenarios*, *Steps*, *Components*, *ProcessingResources*) to concrete ones and assigning their property values with fixed ones. Table 5.5 describes the binding of the generic CSM elements to the concrete ones. The values of the properties of the new roles (such as *multiplicity*, *hostDemand*) are initially undefined. These values can be either estimated or evaluated based on previous simulation experiments or benchmarks. The context specific aspect model produced by the binding process is not shown here; however, it appears (the gray region) within the composed model in Figure 5.10.

Aspect composition is the last step in the AOM approach by weaving the context specific aspect model in the CSM PIM. Figure 5.10 describes the CSM composite aspect.

Table 5.5: Binding Generic to Concrete roles

CSM type	Generic Aspect			Context Specific Aspect		
	Generic elements	**Property**	**value**	**Concrete elements**	**Property**	**value**
Component	*|client*	*multiplicity*	p1	*sales*	*multiplicity*	1
		host	*|ClientHost*		*host*	*SalesHost*
Component	*|xmlParserC*	*multiplicity*	p2	*xmlParserSales* (new)	*multiplicity*	?
		host	*|ClientHost*		*host*	*SalesHost*
Component	*|soapClient*	*multiplicity*	p3	*soapSales* (new)	*multiplicity*	?
		host	*|ClientHost*		*host*	*SalesHost*
Component	*|soapProvider*	*multiplicity*	p4	*soapScheduling* (new)	*multiplicity*	?
		host	*|ProviderHost*		*host*	*SchedulingHost*
Component	*|xmlParserP*	*multiplicity*	p5	*xmlParserScheduling* (new)	*multiplicity*	?
		host	|ProviderHost		*host*	*SchedulingHost*
Component	*|provider*	*multiplicity*	p6	*scheduling*	*multiplicity*	1
		host	*|ProviderHost*		*host*	*SchedulingHost*
Step	*RequestService*	*hostDemand*	h1	*RequestScheduleManufacturingPeriod* (new)	*hostDemand*	?
Step	*MarshallingMessage*	*hostDemand*	h2	*MarshallingMessage* (new)	*hostDemand*	?
Step	*RequestSOAPMessage*	*hostDemand*	h3	*RequestSOAPMessage* (new)	*hostDemand*	?
Step	*UnmarshallingMessage*	*hostDemand*	h4	*UnmarshallingMessage* (new)	*hostDemand*	?
Step	*ServiceInvocation*	*hostDemand*	h5	*ScheduleManufacturingPeriod*	*hostDemand*	0.15
Step	*MarshallingMessage*	*hostDemand*	h6	*MarshallingMessage* (new)	*hostDemand*	?
Step	*TxOhStep*	*hostDemand*	*commTxOvh*	*TxOhStep*	*hostDemand*	0.15
Step	*RcvOhStep*	*hostDemand*	*commRcvOvh*	*RcvOhStep*	*hostDemand*	0.15
Step	*CommStep*	*hostDemand*	msg/lc1	*CommStep*	*hostDemand*	1/100
Step	*UnmarshallingMessage*	*hostDemand*	h7	*UnmarshallingMessage* (new)	*hostDemand*	?
ProcessingResource	*|ClientHost*	*multiplicity*	m1	*SalesHost*	*multiplicity*	1
ProcessingResource	*|ProviderHost*	*multiplicity*	m2	*SchedulingHost*	*multiplicity*	1

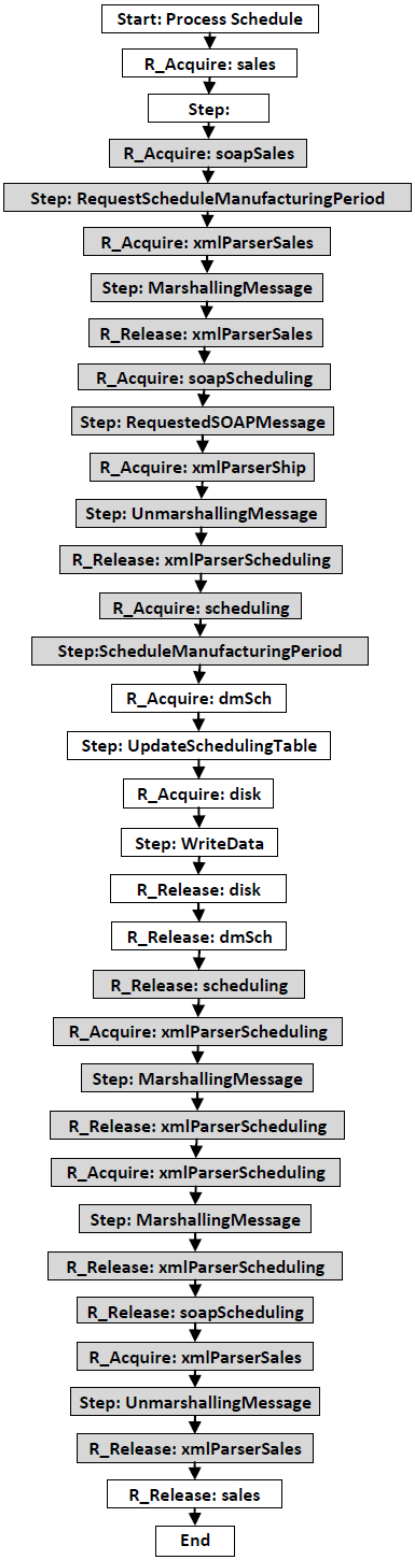


Figure 5.10: The Composed model for *ProcessSchedule*

5.1.3 Aspect composition at the LQN level

PUMA4SOA allows the AOM aspect composition to be performed at the LQN level. The CSM of PIM and the generic CSM of platform aspect model are first transformed into LQN models separately [ALH12a], as in Figure 5.11.

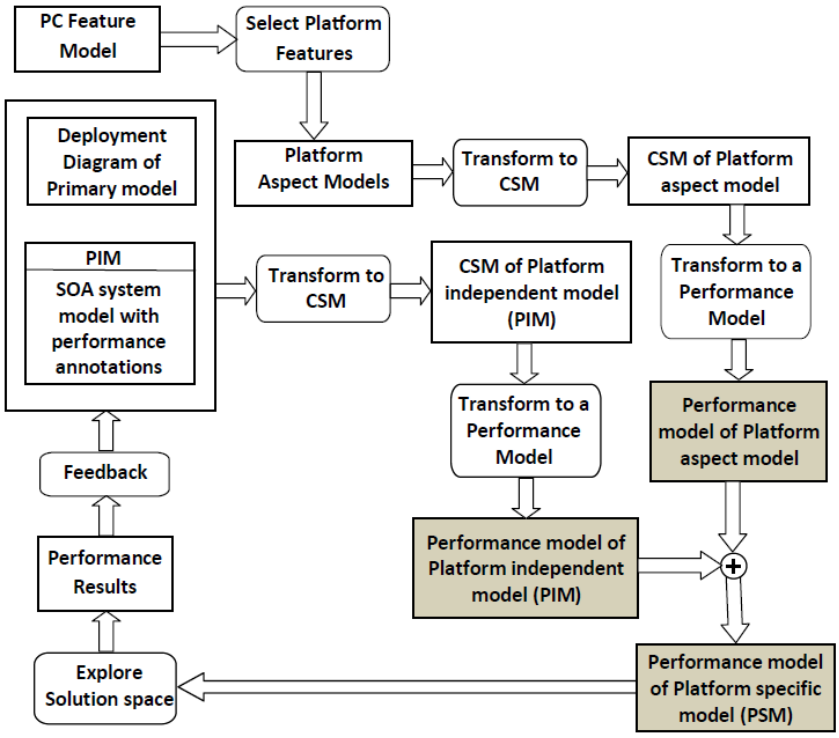


Figure 5.11: PUMA4SOA approach: Aspect Composition at the LQN Level

Figure 5.12 describes the LQN model of platform independent model of PO system. The Reference task (*User*) is used to generate traffic for the underlying LQN model, which can be either an open workload or a closed workload. The business layer, which represents the CSM top scenario model, is transformed into a top level LQN *ActivityGraph* associated with a task called *POProcess* allocated on *SalesHost* [ALH12b]. The CSM subscenario models (representing the service layer) are transformed into a set of tasks associated with entries which describe the service operations. Figure 5.13 describes the LQN model of the generic platform aspect model. Notice that the task |*Network* is used to represent the network delays (the *TxOhStep*, *RcvOhStep*

and *CommStep* in CSM model). The |*Network* task can be either a |*LAN* task or a |*WAN* task based on whether the service provider is an internal or external participant.

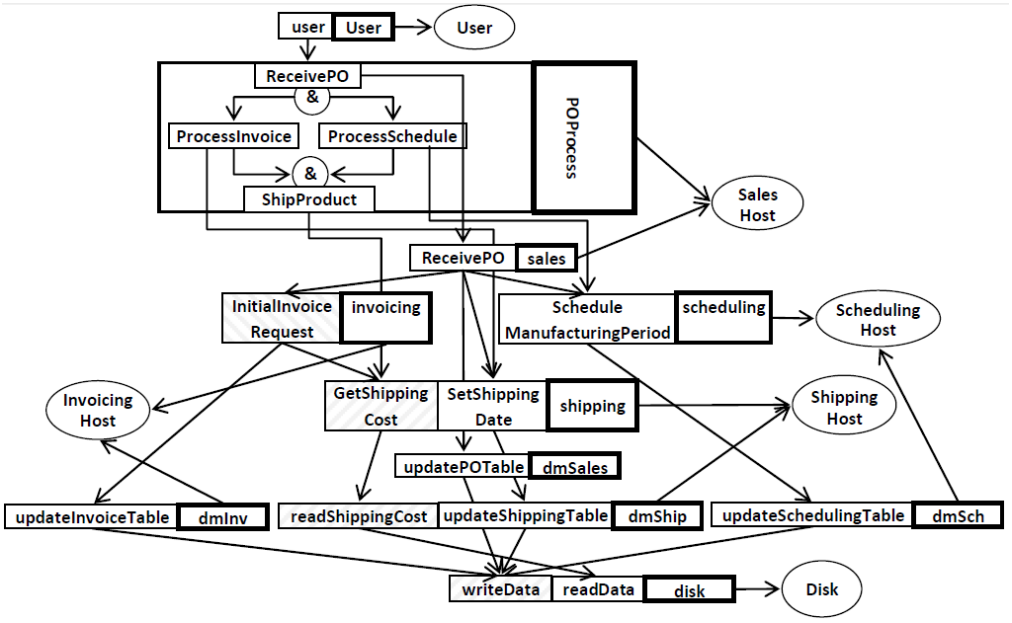


Figure 5.12: The LQN Model for the PO System

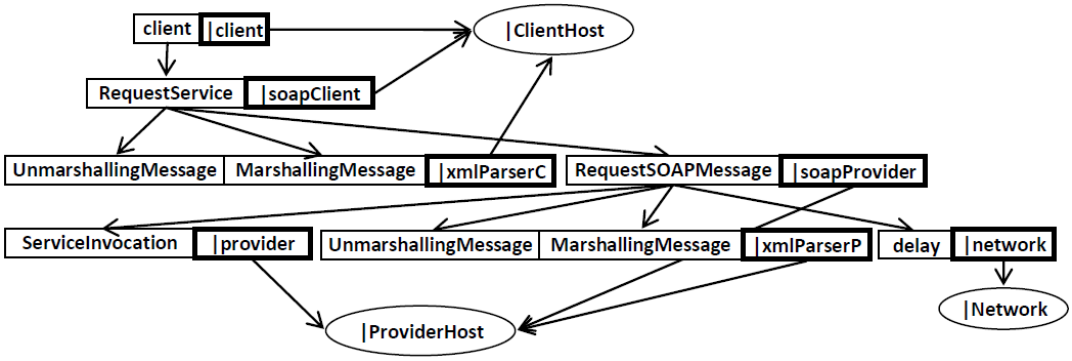


Figure 5.13: The LQN Model for the Service Invocation Aspect

The point-cut conditions cannot be defined at the LQN level, because the details of service architecture model at the UML level is not transformed to CSM model. The drawback of mapping UML to CSM regarding losing the service structure details has been also propagated to the LQN level. The solution is to use the service architecture model defined at the UML level as in Figure 3.4, (to allocate the service operations of the PO system) in conjunction with the LQN

platform independent model (PIM). The LQN point-cut rules for Service Invocation are defined as in Figure 5.14.

Any LQN *Entry* element exits in the LQN platform independent model, such that:

1. The *Entry* element is expressed in the service architecture model as an operation of a provided interface (of symbol o—).
2. This provided interface is owned by a port stereotyped with *«Service»*.

Figure 5.14: Defining the Point-cut rules for Service Invocation at the LQN level

The join-points are identified based on the point-cut conditions. Any *Entry* in the LQN model of PIM complies with the point-cut conditions is a join-point as in Table 5.6.

Table 5.6: Identifying the Join-points at the LQN level

Join-point	Workflow activity
InitialInvoiceRequest	*Process Invoice*
GetShippingCost	*Ship Cost*
ScheduleManufacturingPeriod	*Process Schedule*
SetSchedulingDate	*Ship Product*

Instantiating the context-specific aspect model is the next step. This step is similar to the one performed at the UML and CSM levels, where the generic elements defined in the LQN model in Figure 5.13 are bound with concrete ones, and assigning their property values with fixed ones. Table 5.7 describes the binding of the generic LQN elements to the concrete ones. The context specific aspect model produced by the binding process is not shown here; however, it appears (the gray region) within the composed model in Figure 5.15.

Table 5.7: Binding Generic to Concrete roles

CSM type	Generic Aspect			Context Specific Aspect		
	Generic elements	**Property**	**value**	**Concrete elements**	**Property**	**value**
Task	|*client*	*multiplicity*	p1	*sales*	*multiplicity*	1
		processor	|*ClientHost*		*processor*	*SalesHost*
Task	|*xmlParserC*	*multiplicity*	p2	*xmlParserSales* (new)	*multiplicity*	?
		processor	|*ClientHost*		*processor*	*SalesHost*
Task	|*soapClient*	*multiplicity*	p3	*soapSales* (new)	*multiplicity*	?
		processor	|*ClientHost*		*processor*	*SalesHost*
Task	|*soapProvider*	*multiplicity*	p4	*soapScheduling* (new)	*multiplicity*	?
		processor	|*ProviderHost*		*processor*	*SchedulingHost*
Task	|*xmlParserP*	*multiplicity*	p5	*xmlParserScheduling (new)*	*multiplicity*	?
		processor	|*ProviderHost*		*processor*	*SchedulingHost*
Task	|*provider*	*multiplicity*	p6	*scheduling*	*multiplicity*	1
		processor	|*ProviderHost*		*processor*	*SchedulingHost*
Entry	*RequestService*	*demand*	h1	*RequestScheduleManufacturingPeriod* (new)	*demand*	?
Entry	*MarshallingMessage*	*demand*	h2	*MarshallingMessage* (new)	*demand*	?
Entry	*RequestSOAPMessage*	*demand*	h3	*RequestSOAPMessage* (new)	*demand*	?
Entry	*UnmarshallingMessage*	*demand*	h4	*UnmarshallingMessage* (new)	*demand*	?
Entry	*ServiceInvocation*	*demand*	h5	*ScheduleManufacturingPeriod*	*demand*	0.15
Entry	*MarshallingMessage*	*demand*	h6	*MarshallingMessage* (new)	*demand*	?
Entry	*delay*	*demand*	*commTxOvh + commRcvOvh + msg/lc1*	*delay*	*demand*	0.15 + 1/100 +0.15
Entry	*UnmarshallingMessage*	*demand*	h7	*UnmarshallingMessage* (new)	*demand*	?
Processor	|*ClientHost*	*multiplicity*	m1	*SalesHost*	*multiplicity*	1
Processor	|*ProviderHost*	*multiplicity*	m2	*SchedulingHost*	*multiplicity*	1

The final step is performing the aspect compositions at the join-points. In the PO example, we composed the context specific aspect model of *ScheduleManufacturingPeriod* by adding new tasks with entries to the LQN model which express the service platform of the *sales* (client), *scheduling* (provider), and *LAN* (network delay). Figure 5.15 shows the result of aspect composition in LQN.

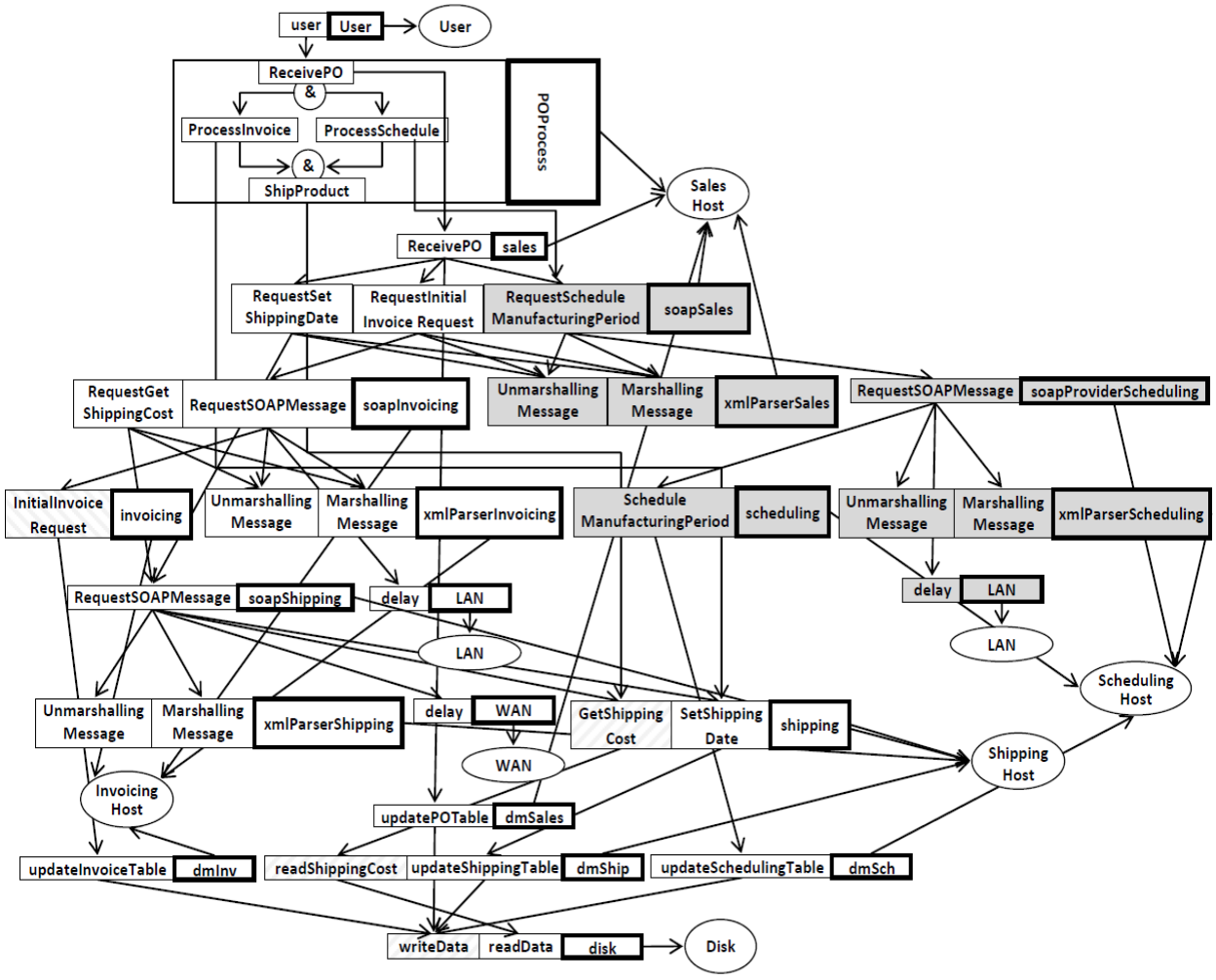


Figure 5.15: The LQN Composed Model

5.1.4 Comparisons between Aspect composition at UML, CSM and LQN levels

PUMA4SOA provides the flexibility of transforming platform independent model to platform specific model based on AOM aspect composition at three levels (UML, CSM and LQN).

The main advantage of performing aspect composition in UML stems from the language features. UML is an OMG standard modeling language. It is popular and supported by several model driven tools. We also noticed that we were able to define easily the point-cut rules in UML, which is more complicated in CSM and LQN (we have to go back to the service architecture model). Another advantage of performing aspect composition in UML is the ability to modify models with changing requirements, which we cannot do in CSM or LQN.

On the other hand, CSM and LQN both have a lightweight metamodel, compared with UML, whose metamodel is considerably more complex. This advantage makes the implementation of the aspect composition easier in CSM and LQN; it is significantly easier to ensure composition consistency and to reduce the execution time required for the model transformation in PUMA4SOA [ALH12a]. CSM and LQN are also indepenedent from the UML source model, so aspect composition can be used for software modeling languages other than UML, such as UCM. CSM is also defined between the UML source model and the LQN performance model; this allows for using CSM aspect composition to generate different performance models, such as LQN and Petri net. Table 5.8 summarizes the points of comparison between the three levels.

Table 5.8: Comparison between aspect composition at UML, CSM and LQN level

Points of Comparison	UML level	CSM level	LQN level
Language features (standard, popularity)	✓	×	×
Modeling langauge abstraction level	×	✓	✓
Ability to modify model with changing requirements	✓	×	×
Ability to define the point-cut rules	✓	×	×
Metamodel complexity	×	✓	✓
How easy is to insure composition consistency	×	✓	✓
Implementation complexity of aspect composition	×	✓	✓
Execution time of model transformations in PUMA4SOA	×	✓	✓
Independent from source model	×	✓	✓
Independent from target model	✓	✓	×

5.2 AOM aspect composition at the CSM level (best solution)

The platform specific aspect model (PSM) is generated from the platform independent model (PIM) using AOM aspect composition. We decided to perform the aspect composition in PUMA4SOA at the CSM level due to the advantages presented before. In the following sections, we present the algorithms and techniques used in defining AOM step: Defining the point-cut rules, identifying the join-point, Instantiating a context specific aspect model and performing the aspect composition.

5.2.1 Defining the point-cut rules

Defining the point-cut conditions is used to identify the join-point at the CSM model. The point-cut specifies the conditions which differentiate between the CSM steps that behave as a service operation or a regular call. Unfortunately, all CSM metamodel versions do not explicitly differentiate between the two types. In fact, CSM supports a light weight metamodel which is mainly focused on the behaviour of the scenarios. So, during the model transformation to CSM model, most of the service details described at the UML service architecture model are not mapped.

There are different approaches which have been proposed to solve this problem (not being able to define the point-cut conditions at the CSM level). We find that the ideal approach is by introducing the concept of *Metadata* to the CSM specification. *Metadata* is a light weight profiling mechanism which extends a CSM metamodel by defining name-value pairs to capture stereotypes. Figure 5.16 shows the extended CSM metamodel, where a *Metadata* element is aggregated with a *Scenario* element.

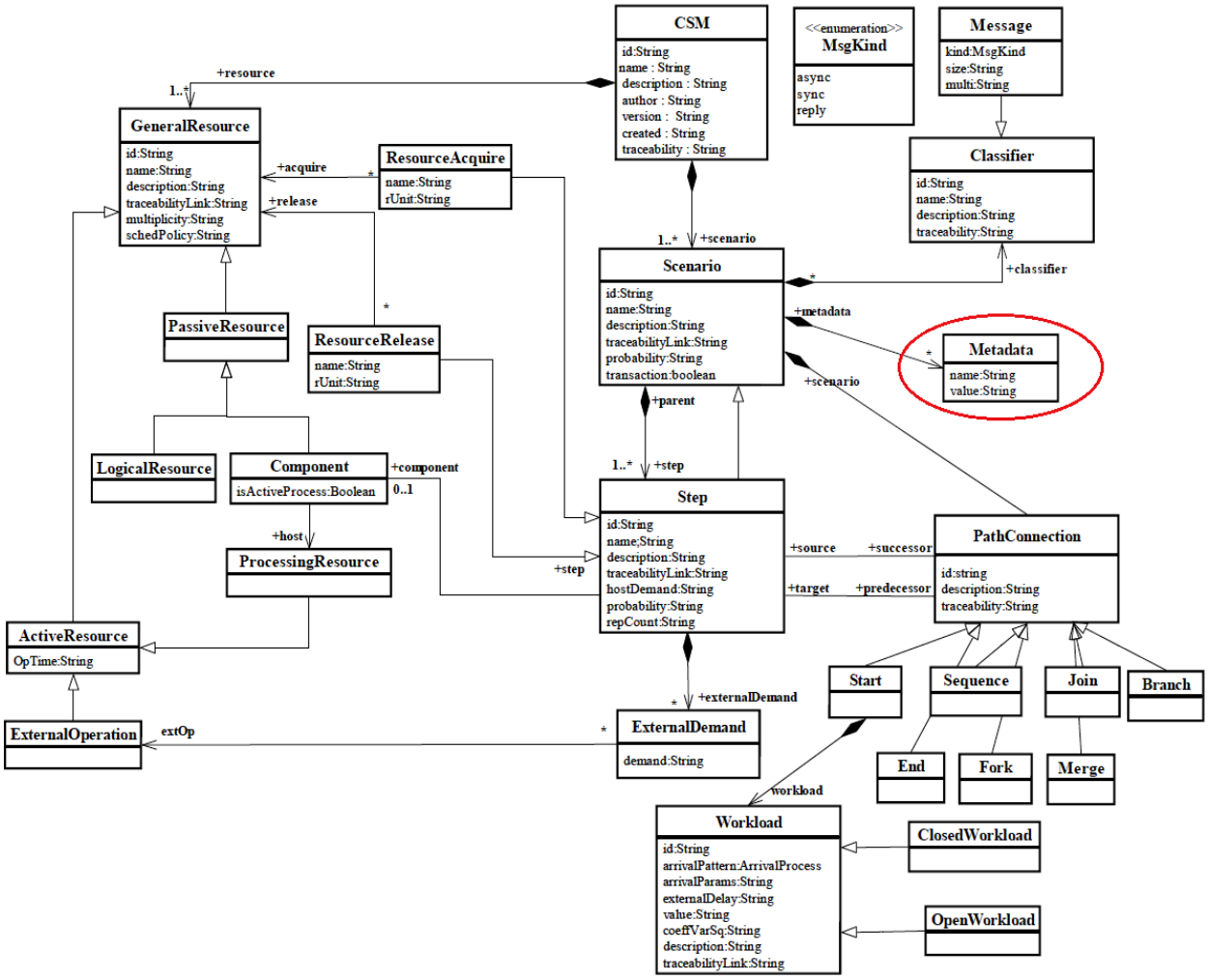


Figure 5.16: Extended CSM metamodel

The *Metadata* element defines the *name* and the *value* properties which are used to tag the *Scenario* and *Step* elements. During the mapping from the UML to CSM, all the service

operations defined in the service architecture service model are mapped to CSM steps and those steps are tagged with metadata, where the name = "_JOIN_POINT" and the value= "ServiceOp". The tagged value is used to differentiate between a service operation step and a regular call step when defining point-cut conditions. Figure 5.17 shows the point-cut rules at the CSM level.

Condition: Defining the point-cut rules of CSM aspect composition

Any CSM *StepType* element exists in CSM PIM, such that:

StepType element is stereotyped with ‹‹*ServiceOp*››

Figure 5.17: Defining the Point-cut rules at the CSM Level

5.2.2 Identifying the join-points

Join-points are the locations on the primary model where the aspect model will be inserted. In a CSM model, join-points are steps which are only found at the subscenarios of the CSM top scenario model. In order to allocate the join-points, we need to check the point-cut conditions on each step defined in the subscenarios. Algorithm 1 describes the step of indentifying the join-points at the CSM level.

Algorithm 1: Identifying the join-points at the CSM model

```
helper CSM::identifyJoinPoint(in csm:CSMType, out csmJoinPoints:List) : List {
    csm.getSteps() → forEach(step) {
        if (step.getRefinement() <> null) {
            Var refinement= step.getRefinement();
            Var scenario= model.getScenario(refinement.getSub());    }
        scenario.getSteps() → forEach(stp){
        //Check the point-cut condition
            if(stp.getMetadata().getValue()= "ServiceOp")
                csmJoinPoints.add(stp);    }    }
return csmJoinPoints;    }
```

Algorithm 1 navigates through the CSM sub-scenarios to identify the steps that satisfy the point-cut conditions. The output is a list of all join-points (Steps) in the CSM sub-scenarios.

5.2.3 Binding the formal parameters of the generic aspect model

Instantiating the generic aspect model to generate the context specific aspect model is the next step after identifying the join-points. The instantiation requires binding the generic elements to concrete ones and assigning their property values with fixed ones. We presented three generic aspect models in the previous chapter: Service Invocation (Figures 3.9, 3.10, on page 51), Service Publishing (Figures 3.11, 3.12, on page 52) and Service Discovery (Figures 3.13, 3.14, on page 54).

Initially, the aspect model has a generic form that is independent from a specific platform. During the binding process, an automatic binding between the generic elements and their matching occurs at each join-point identified in the PIM. Some of the generic elements, which do not have matching, are bound to newly created ones.

At each join-point in the PIM, there are many concrete elements used for automatic binding: a) the *Step* element that represents the join-point, b) the *ResourceAcquire* and *ResourceRelease* elements of the service consumer and c) the *ResourceAcquire* and *ResourceRelease* elements of the service provider. Figure 5.18 shows an example of a CSM PIM, where the component *c1* (represents service consumer) is acquired to execute the step *Op1*. Then the component *p1* (represents the service provider) is acquired by *c1* to execute the step *JP* (represents the join-point). Both *c1* and *p1* are then released. The concrete elements, *c1* and *p1*, of the PIM can be automatically navigated and bound with |*client*, |*provider* respectively (the generic elements defined in the generic aspect model). The concrete element *JP* can also be automatically navigated and bound with multiple generic elements: *ServiceInvocation* step (at

service invocation aspect model), *PublishService* step (at service publishing aspect model) and *QueryService* (at service discovery aspect model).

Algorithm 2 describes the process of automatic navigation and binding of the generic elements. It is divided into three parts:

1. In algorithm 2a, the binding occurs between the generic step *"Step: ServiceInvocation"* and its equivalent the concrete join-point step *"Step: JP"*. The binding has been also applied between the properties of the generic and concrete elements, such as *id*, *hostDemand* and *repCount*. The same binding applies when the generic aspect is the service publishing: *"Step:*

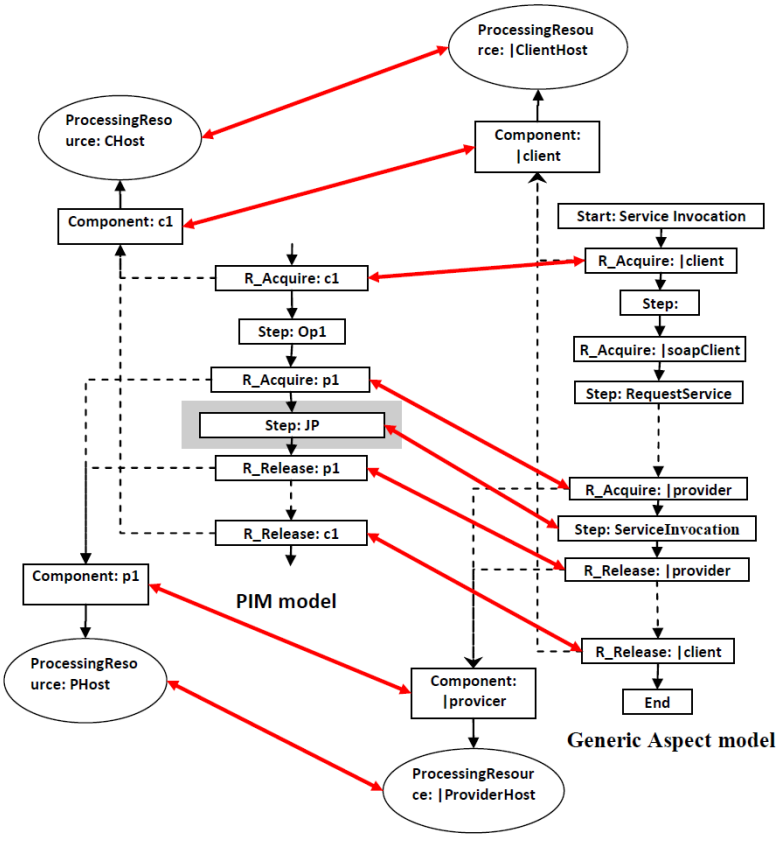


Figure 5.18: Binding the elements of the Generic Aspect Model to the PIM context

PublishService" binds with the join-point step *"Step: JP"* and when the generic aspect is the service discovery: *"Step: QueryService"* binds with the join-point step *"Step: JP"*.

For simplicity and in order to avoid redundancy, algorithm 2a describes the binding for only the *ServiceInvocation* step and a sample of its properties.

Algorithm 2: Automatic binding of generic elements with concrete ones

```
helper CSM::bindingGenericElements(in joinPoint:StepType, in pim:CSMType, in
genericAmodel:CSMType) : CSMType{
    //Initially cloning the concrete aspect model from the generic aspect model.
    constructor  CSMType::concreteAmodel(genericAmodel:CSMType) {
        scenarioList:= genericAmodel.getScenarios();     }
        pim.getSteps() → forEach(step) {
            if (step = joinPoint){
            //Algorithm 2a: binding the generic service operation with the join-point.
            concreteAmodel := map genericToConcreteService(joinPoint);
            // Algorithm 2b: binding the generic client element with the concrete one at PIM.
           concreteAmodel := map genericToConcreteClient(joinPoint, concreteAmodel, pim)     }
            //Algorithm 2c: binding the generic provider element with the concrete one at PIM.
            concreteAmodel := map genericToConcreteProvider(joinPoint,concreteAmodel);
           //Algorithm 2d: binding the generic Tx and Rcv overheads and Communication delay
            //elements with the concrete ones at PIM.
           concreteAmodel := map genericToConcreteDelay(joinPoint,  concreteAmodel, pim);
return concreteAmodel;     }
```

Algorithm 2(a): binding the generic service operation with the join-point.

```
mapping CSMType::genericToConcreteService(in joinPoint:StepType):CSMType
    when { concreteAmodel.getSteps().get("ServiceInvocation") <> null}
    {
        Var serviceStep= concreteAmodel.getSteps().get("ServiceInvocation");
        object serviceStep:StepType{
        name= joinpoint.getName();
        id = joinPoint.getID();
        hostDemand= joinPoint.getHostDemand();
        repCount = joinPoint.getRepCount();
```

Algorithm 2(a): binding the generic service operation with the join-point.

```
    return concreteAmodel;    }
// The same procedures applies to "Publish Service" and "Query Service" aspects
```

2. In algorithm 2b, the generic elements (|*client* and |*ClientHost*) are bound to the concrete client elements (the component and the host) in the PIM. The generic *ResourceAcquire* and *ResourceRelease* elements of the client are also bound to their concrete equivalents in the PIM. In order to define the concrete elements of the client (the component and the host), we use the functions getComponent and getHost of the step which precedes the *Join-Point* step to get the concrete values of the client.

Algorithm 2b: binding the generic client element with the concrete one at PIM.

```
mapping CSMType::genericToConcreteClient(in joinPoint:StepType, in concreteAmodel:CSMType, in
pim:CSMType):CSMType{
    Var prevStep= pim.getSteps().get(joinPoint.getIndex()-1);
    Var genericHost=  concreteAmodel.getSteps().getHost("|ClientHost");
    //Binding the Generic Host of the Client
    object genericHost:ProcessingResource{
        name= prevStep.getHost().getName();
        id = prevStep.getHost().getID();
        multiplicity= prevStep.getHost().getMultiplicity();    }
    Var genericComponent=  concreteAmodel.getSteps().getComponent("|client");
    //Binding the Generic Component of the Client
    genericComponent:ComponentType{
        name= prevStep.getComponent().getName();
        id = prevStep.getComponent().getID();
        multiplicity= prevStep.getComponent().getMultiplicity();
        host= prevStep.getComponent().getHost();    }
return concreteAmodel; }
```

3. In algorithm 2c, the generic elements *(|provider* and *|ProviderHost*) are bound to the concrete provider elements (the component and the host) in the PIM. The generic *ResourceAcquire* and *ResourceRelease* elements of the provider are also bound to their concrete equivalents in the PIM. We use the functions *getComponent* and *getHost* of the *join-Point* step to get the concrete values of the provider.

Algorithm 2c: binding the generic provider element with the concrete one at PIM.

```
mapping CSMType::genericToConcreteProvider(in joinPoint:StepType, in
concreteAmodel:CSMType ):CSMType {
    Var genericHost=  concreteAmodel.getSteps().getHost("|ProviderHost");
    //Binding the Generic Host of the Provider
    object genericHost:ProcessingResource{
        name= joinPoint.getHost().getName();
        id = joinPoint.getHost().getID();
        multiplicity= joinPoint.getHost().getMultiplicity(); }
    Var genericProvider= concreteAmodel.getSteps().get("|provider");
    //Binding the Generic Component of the Provider
    object genericProvider:ComponentType{
        name= joinPoint.getComponent().getName();
        id = joinPoint.getComponent().getID();
        multiplicity= joinPoint.getComponent().getMultiplicity();
        host= joinPoint.getComponent().getHost();    }
return concreteAmodel; }
```

4. In algorithm 2d, the generic elements (*TxOhStep*, *RcvOhStep* and *CommStep*) are also bound to their equivalents in PIM when there are message overheads and communication delay.

Algorithm 2d: binding the generic Tx and Rcv overheads and Communication delay elements with the concrete ones at PIM.

```
mapping CSMType::genericToConcreteDelay(in joinPoint:StepType, in concreteAmodel:CSMType,  in
pim:CSMType):CSMType {
```

Algorithm 2d: binding the generic Tx and Rcv overheads and Communication delay elements with the concrete ones at PIM.

```
Var concreteTxOh=  PIM.getSteps().getSteps("TxOhStep");
Var genericTxOh= concreteAmodel. getSteps("TxOhStep");
//Binding the Generic Tx overhead
object genericTxOh:StepType{
    name= concreteTxOh.getName();
    id = concreteTxOh.getID();
    hostDemand= concreteTxOh.getHostDemand(); }
Var concreteRcvOh=  PIM.getSteps().getSteps("RcvOhStep");
Var genericRcvOh= concreteAmodel. getSteps("RcvOhStep");
// Binding the Generic Rcv overhead
object genericRcvOh:StepType{
    name= concreteRcvOh.getName();
    id = concreteRcvOh.getID();
    hostDemand= concreteRcvOh.getHostDemand(); }
Var concreteComm=  PIM.getSteps().getSteps("CommStep");
Var genericComm= concreteAmodel. getSteps("CommStep");
// Binding the Generic communication delay
object genericComm:StepType{
    name= concreteComm.getName();
    id = concreteComm.getID();
    hostDemand= concreteComm.getHostDemand(); }
return concreteAmodel; }
```

Table 5.9 presents the list of generic elements which do not have equivalent concrete elements in the PIM. These elements are bound manually to newly created ones. Algorithm 3 describes the steps of the manual binding of the generic elements. The properties of the generic elements, of type number, are bound with estimated or calculated concrete values based on

previous simulations or benchmarks. If the concrete values are not available, we can initialize those properties (in our case we use 1).

Table 5.9: List of the generic elements which are bound manually

CSM type	Generic Aspect	
	Generic elements	**Property**
Component	*|xmlParserC, |soapClient, |soapProvider, |xmlParserP*	*name, multiplicity*
Step	*RequestService, MarshallingMessage, RequestSOAPMessage, UnmarshallingMessage*	*name, hostDemand, repCount*

Algorithm 3: Manual binding generic elements with concrete ones

```
helper CSM::bindingGenericElements(in concreteName:String, in genericElement:CSMType, out
concreteAmodel:CSMType) : CSMType{
    if(concreteAmodel.contains(genericElement)){
        object  genericElement:ComponentType{
            name= concreteName;
            multiplicity= '1";   }
        object genericElement:StepType{
            name= concreteName;
            hostDemand= "1";
            repCount= "1";  }
 return concreteAmodel;  }
```

5.2.4 Weaving context-specific aspect models into the CSM primary model

The final step in CSM aspect composition is weaving the context specific aspect models at each join-point of the primary model to generate the composed model. There are three functions performing aspect composition: Insert before the join-point, Insert after the join-point, and replace the join-point. The process of weaving the context specific aspect model is performed automatically except for the selection of the aspect composition function, which is done manually.

The complexity of aspect composition depends on the scenario of PIM and whether the join-point is executing nested steps or not. In general, the aspect composition is split into two parts: a) the top part (represents the service invocation request) is composed before the step that acquires the service provider and b) the bottom part (represents the service invocation reply) is composed after the step that releases the service provider. Figure 5.19 shows the *Process Schedule* subscenario of the PO system example where the top part is composed before the element *R_Acquire:scheduling*, and the bottom part is composed after the element *R_Release:scheduling*.

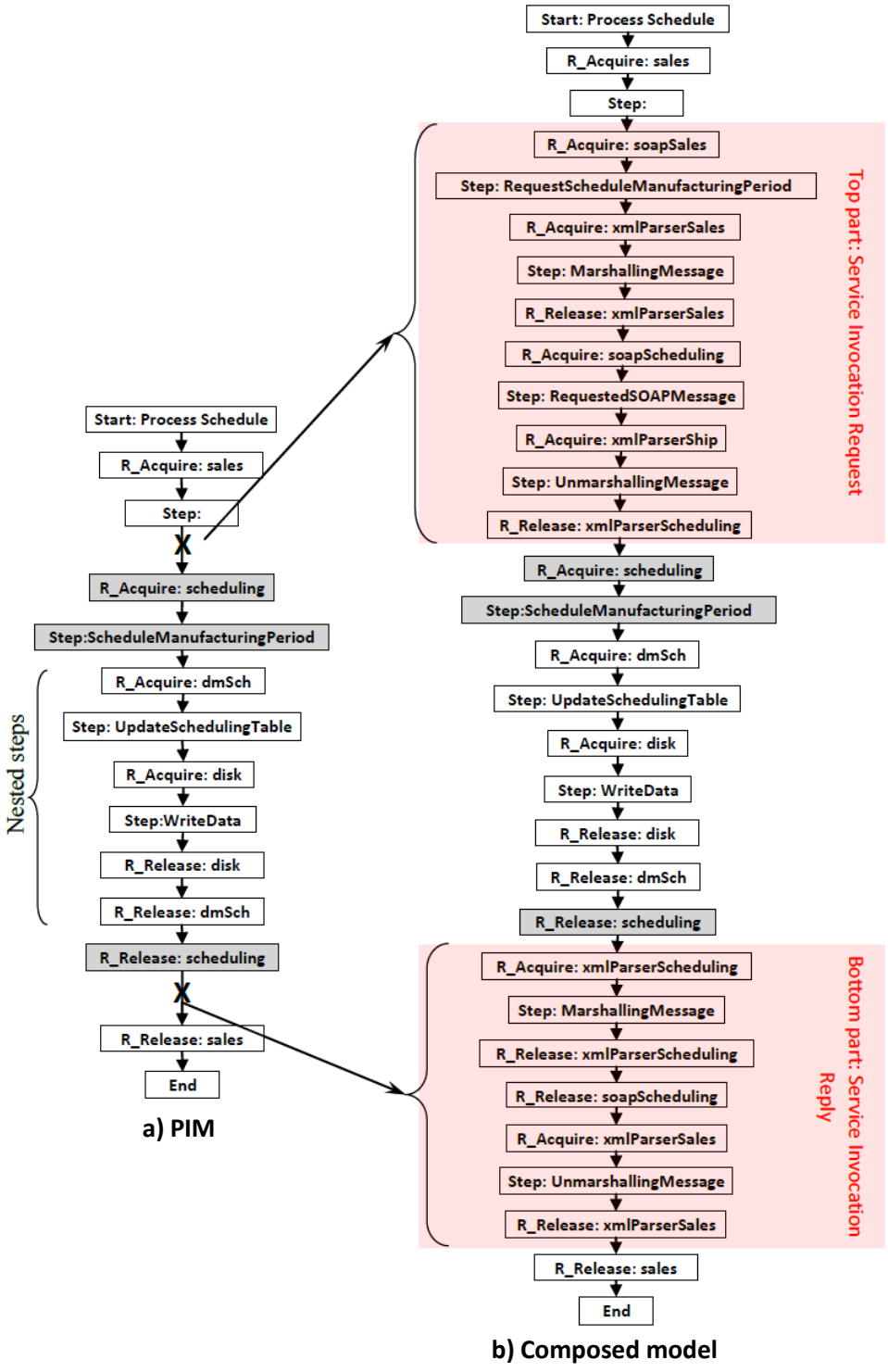


Figure 5.19: The Aspect Composition of *Process Schedule* example

Algorithm 4 describes the steps of composing a context specific aspect model into a PIM to generate the composed model following the same way described before. The elements of the context aspect model are added to the composed model. The algorithm is valid for multiple aspect compositions.

Algorithm 4: Automatic weaving of a set of context specific aspect models into CSM primary model

```
helper CSM::weavingContextSpecificAspectModel(in pim:CSMType, in joinPoint:StepType,  in
contextAmodel:CSMType) : CSMType{
    //Initially cloning the composed model from PIM.
    constructor  CSMType::composedModel(pim:CSMType) {
        scenarioList:= pim.getScenarios();    }
    Var JPComponent= joinPoint.getcomponent();
    //For top part composition
    contextAmodel.getSteps()→forEach(step){
         if(not pim.contains(step) and step.getName() <> JPComponent.getName()){
             composedModel.add(step);  }    }
    //For bottom part composition
    contextAmodel.getSteps()→forEach(step){
        if(step ofType ResourceRelease and step.getName() = JPComponent.getName()){
            step.getNext();
            while(step.hasNext()){
                composedModel.add(step);
                step.getNext(); }  }   }
return composedModel;    }
```

Chapter 6: Traceability Modeling

Traceability is used to maintain relationships between different software artifacts during the software life cycle. It is also used to define the dependencies between related elements in different models, to propagate changes of properties from one model to another and to analyze the impact of these changes. In PUMA4SOA, a traceability model is used to define the trace-links between the elements of the source and the target models during the model transformation chain. In this chapter, we present the approach used to generate the traceability model, which begins by creating the metamodel of the trace-links between the elements of the UML, CSM and LQN models. The metamodel is then used to define the trace-links' instances between the elements during the modeling transformation chain.

Section 6.1 presents the traceability metamodel of PUMA4SOA. Section 6.2 presents the trace-links related to aspect models. Section 6.3 presents the specifications for applying trace-links in PUMA4SOA. Section 6.4 presents a traceability model for the purchase order system case study.

6.1 Traceability metamodel of PUMA4SOA

PUMA4SOA is a modeling framework which generates an LQN model from the UML design models of a SOA system, through a chain of model transformations. Each model transformation maps the elements of the source model into their equivalent in the target model, and it does not provide any sort of traces between them. In order to be capable of tracing, and analyzing the impact of changes between different models, there is a need to apply trace-links between the elements of the source and the target models.

The traceability metamodel of PUMA4SOA is built using an approach similar to [PAI11] where typed trace-links are defined between the source and the target models in a model

transformation chain. The traceability metamodel of PUMA4SOA defines three groups of trace-links, as shown in Figure 6.1: a) *UML2CSMTraceLink*, which defines trace-links between the elements of the UML (the source) and CSM (the target) models, b) *CSM2LQNTraceLink*, which defines trace-links between the elements of CSM (the source) and the LQN (the target) models and c) *LQN2UMLTraceLink*, which defines the trace-links between the elements of the LQN (the

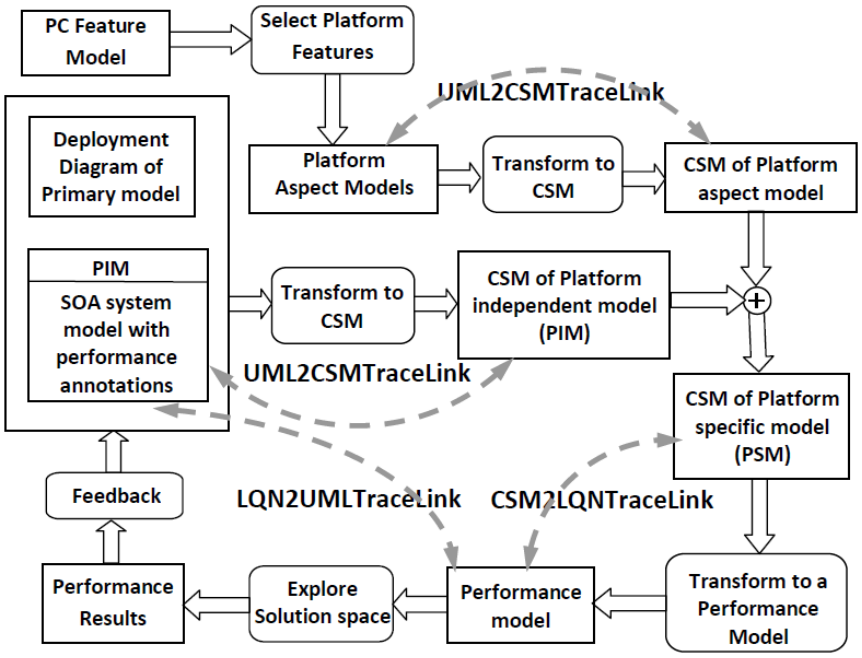


Figure 6.1: Traceability in PUMA4SOA

source) and the UML (the target) models. The first two groups correspond to the model transformations, as in figure 6.1, while the third one can be derived from the first two and it is used to complete tracing back the LQN output results to the UML models. The three trace-links groups are aggregated into *TraceModel* as shown in Figure 6.2.

In our approach, trace-links are applied only to the elements that hold performance properties, which are mainly the elements that are annotated with MARTE. Other elements, like *ControlNodes* and *ControlFlow*, and *PathConnections,* are not traced because they do not hold performance properties.

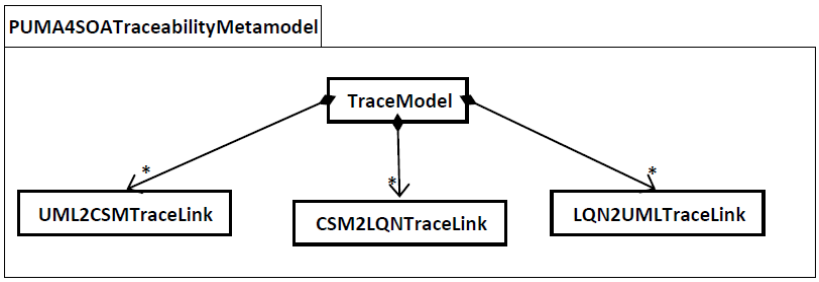


Figure 6.2: The PUMA4SOATraceabilityMetamodel top level

6.1.1 UML to CSM Traceability

The first group of trace-links is defined between the elements of the UML input design models and the CSM model. To define the *UML2CSMTraceLink*, we import the UML and CSM metamodels, since our purpose is to capture trace-links between the elements of the models which conform to those two metamodels. A typed trace-link is used to define the relationship between an element of the source model and an element of the target model. The type of the trace-link is defined as:

Type of trace-link := the source element type + the target element type +"TL"
where *TL* is an abbreviation of Trace-Link

The typed trace-link also defines two association roles: a) the *source*, which refers to the source element in the UML metamodel and b) the *target*, which refers to the target element in the CSM metamodel. As an example, the type of trace-link which defines the relationship between the UML *LifeLine* element and the CSM *Component* element is *LifeLineComponentTL*.

The relationships between the elements of the source and the target models can be a one-to-one (such as the one between the UML *Node* element and the CSM *ProcessingResource* element) and one-to-many (such as the one between the UML *Message* element and the CSM *Step* element). All trace-link metaclasses inherit from the UML2CSMTraceLink. Figure 6.3 presents the traceability metamodel between the UML (at the top) and CSM (at the bottom).

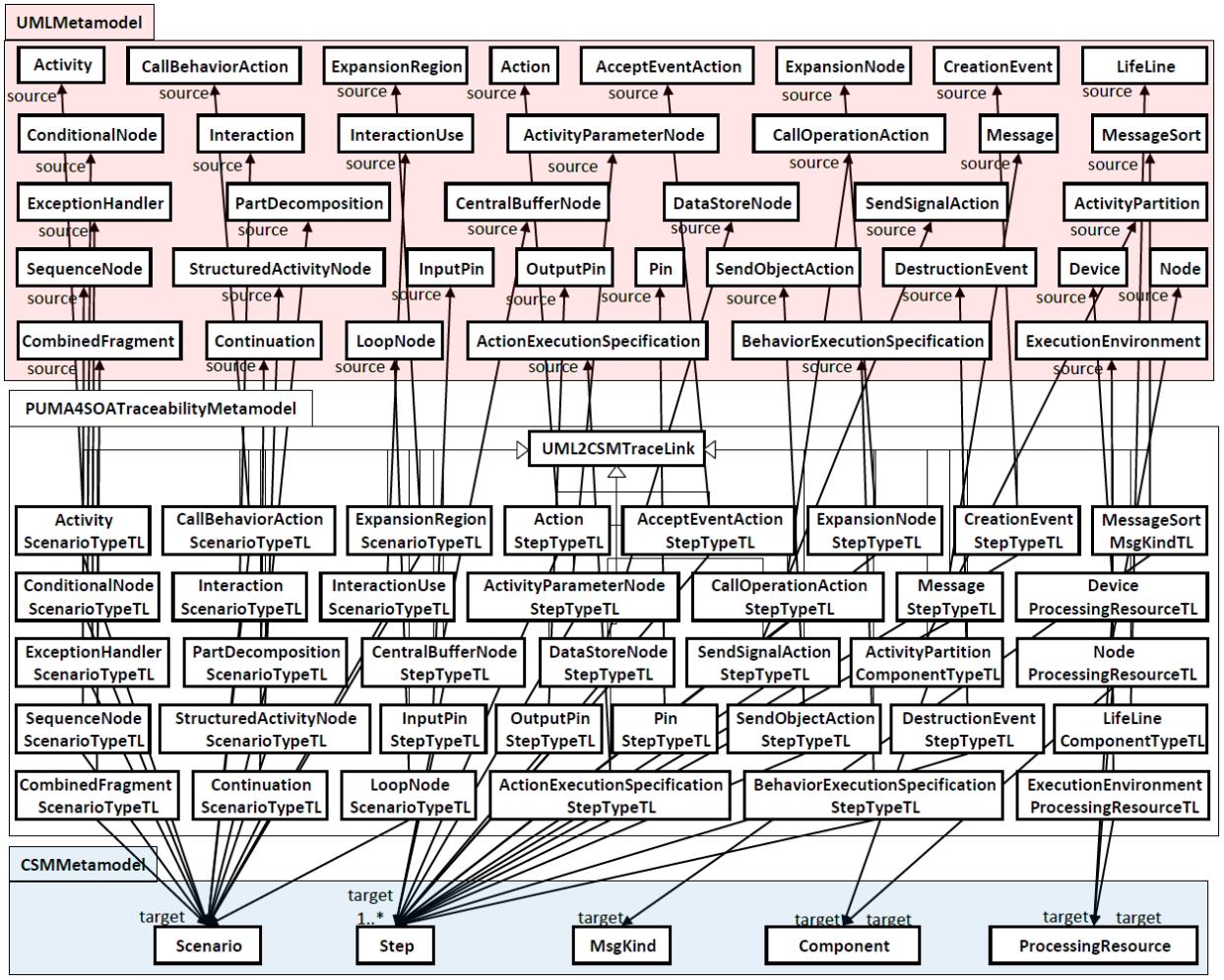


Figure 6.3: Trace-Links metamodel between the elements of UML and CSM

6.1.2 CSM to LQN Traceability

The next group of trace-links in PUMA4SOA is defined between the elements of the CSM and the LQN models. We use the same technique as in the previous section. Our purpose is to capture the traceability between the elements of the models which conform to the CSM metamodel and the LQN metamodel. A typed trace-link defines the relationship in a similar way as the one before between the elements of the CSM and LQN models. Each typed trace-link also has two association roles: a) the *source* which refers to the source element in the CSM metamodel and b) the *target* which refers to the target element in the LQN metamodel. As an

example the type of trace-link which defines the relationship between the CSM *Step* element and the LQN *Entry* element is *StepEntryTL*. Figure 6.4 presents the traceability metamodel between the CSM (at the top) and LQN (at the bottom).

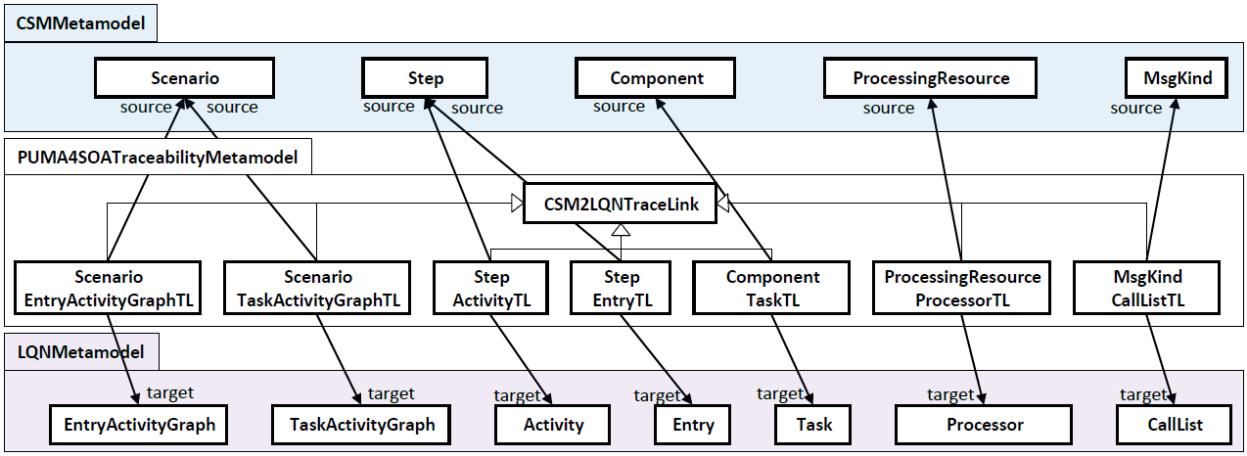


Figure 6.4: Trace-Links metamodel between the elements of CSM and LQN

6.1.3 LQN to UML Traceability

The third group of trace-links in PUMA4SOA is defined between the elements of the LQN (the source) and CSM (the target) models. Our purpose is to capture the traceability between the elements of the models which conform to the LQN metamodel and the UML metamodel. A typed trace-link defines the relationship, in a similar way as the one before, between the elements of the LQN and UML models. Each typed trace-link also has two association roles: a) the *source* which refers to the source element in the LQN metamodel and b) the *target* which refers to the target element in the UML metamodel. As an example, the type of trace-link which defines the relationship between the LQN *Activity* element and the UML *Action* element is *ActivityActionTL*.

Since there is no direct transformation from the LQN model to the UML input design models in PUMA4SOA, the only way to apply the trace-links between their elements is by using

the combined effect of the two model transformations; i.e., from UML to CSM and from CSM to LQN. Figure 6.5 presents the traceability metamodel between the LQN (at the top) and UML (at the bottom).

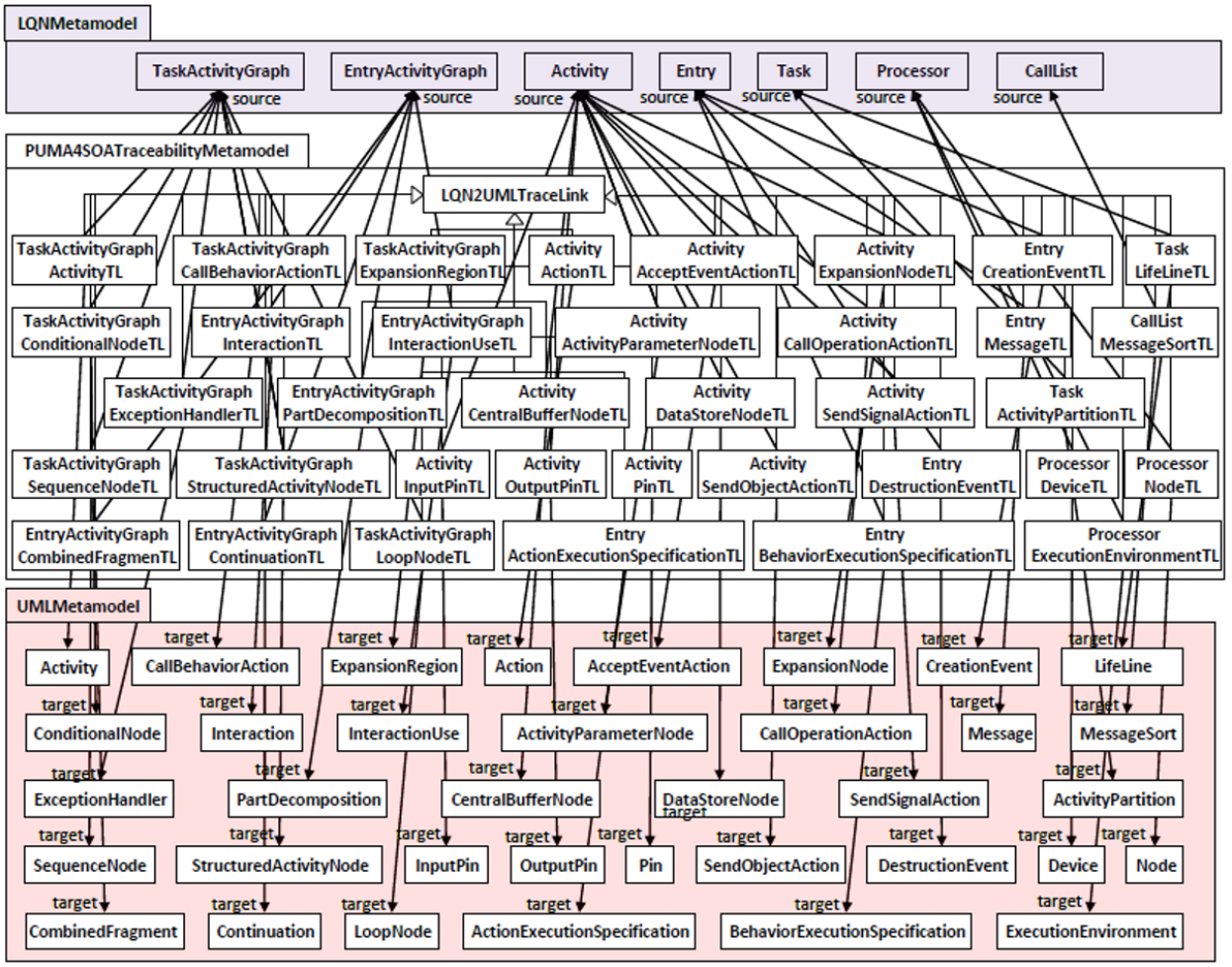


Figure 6.5: Trace-Links metamodel between the elements of LQN and UML

During the performance analysis, the performance output is produced by executing the LQN model using the LQN solver. The *OutputResultType* is an LQN metaclass which creates the elements that store the output results. The LQN metamodel defines six subtypes of *OutputResultType*: *result-processor*, *result-group*, *result-task*, *result-entry*, *result-activity* and *result-call*. Each subtype element is aggregated into its related LQN element; as an example, the *result-processor* is aggregated to a *ProcessorType* instance, the *result-task* is aggregated to a

TaskType instance, etc. The subtype elements own a set of attributes for storing the output results produced by their related LQN elements. The set of the attributes contains: *proc-utilization*, *service-time*, *throughput*, *utilization*, *waiting*, etc. [FRA12]. After executing the LQN model using the LQN solver, the attributes will store the output produced by its related elements and the trace-links between the LQN and UML model elements will be used to update the results to the UML model elements.

6.1.4 Implementing the PUMA4SOA traceability metamodel

As part of the implementation, we have created PUMA4SOA traceability metamodel. The metamodel defines three groups of trace-links which are used to define the relationships between the elements of the UML, the CSM and the LQN. The metamodel is created using an Eclipse project called Eclipse Modeling Framework (EMF). EMF is a modeling framework and code generation which is used to build an application based on a structured data model [EMF13]. It uses a metamodel (Ecore) to describe structured models. The Ecore tool provides a graphical Ecore editor which edits, creates and maintains Ecore models.

The traceability metamodel is created separately from the metamodels of PUMA4SOA modeling languages (UML, CSM and LQN). We begin implementing the traceability metamodel by first importing the three metamodels of the model languages to the traceability metamodel. Then, the traceability top level metaclasses (*TraceModel*, *UML2CSMTraceLink*, *CSM2LQNTraceLink* and *LQN2UMLTraceLink*) and their relationships are defined as in Figure 6.2. Next, for every trace-link, a metaclass is created and two associations are defined: one for the source element and one for its corresponding target element. Figure 6.6 describes the creation of the trace-links between the UML and CSM using Eclipse Ecore.

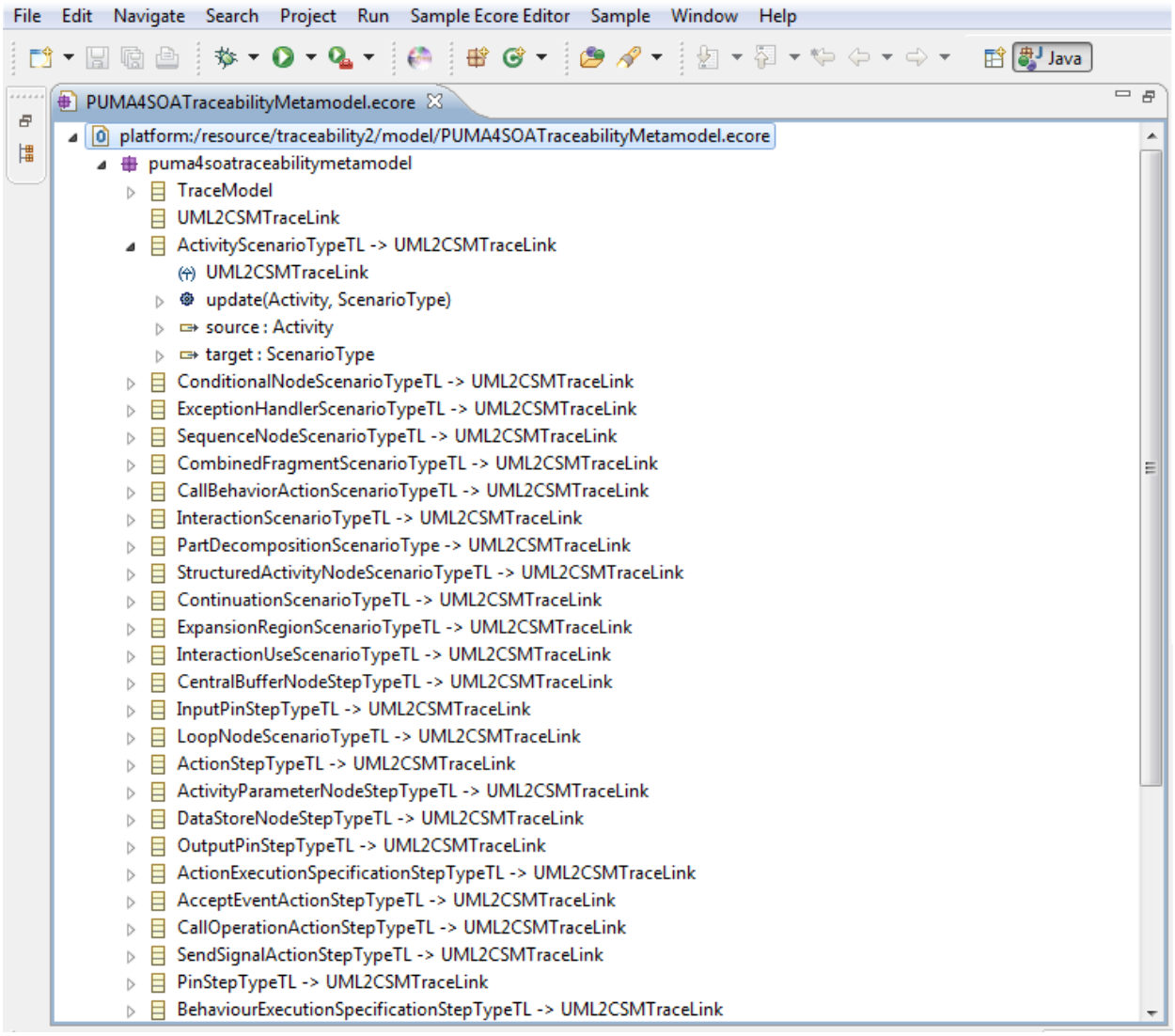


Figure 6.6: Trace-Links metamodel between the elements of UML and CSM using Ecore

6.2 Trace-Links Related to Aspect Models

The trace-links between the elements of the LQN and UML models have not been properly defined. This is because the LQN model is a platform specific model (PSM), which is generated by composing the platform independent model (PIM) with a set of context specific aspect models. The UML input design models, however, consist of a platform independent model (PIM) and a set of generic aspect models. In general, we can describe the models at the LQN and UML levels as follows:

At the LQN: PSM= PIM + $\sum_{i=1}^{n} CA_i$

At the UML: {*PIM, GA1, ... GAn*}

where *CAi* is a context-specific aspect model and GA_i a generic aspect model.

The trace-links can be applied between the model elements of LQN-PIM and UML-PIM. However, we cannot apply trace-links between the model elements of the LQN context specific aspect model and the UML generic aspect model because they are not equivalent.

In order to perform a complete tracing between the model elements of the LQN and the UML, we use the binding table which was created during the aspect composition at the CSM level. The binding table defines the elements of generic aspect model and their concrete values in the context specific aspect model (see Table 5.3). In the binding table, there is a sort of trace between the generic and concrete elements of the aspect model. We can use those traces to complete the trace-links between the LQN and the UML models.

We first define trace-links between the model elements of the LQN-PIM and their equivalent in the UML-PIM; then, we apply trace-links between the rest of the model elements (which do not have equivalent in the UML-PIM) and the concrete model elements defined in the binding table. Figure 6.7 describe the complete definition of trace-links between the UML, CSM and LQN models.

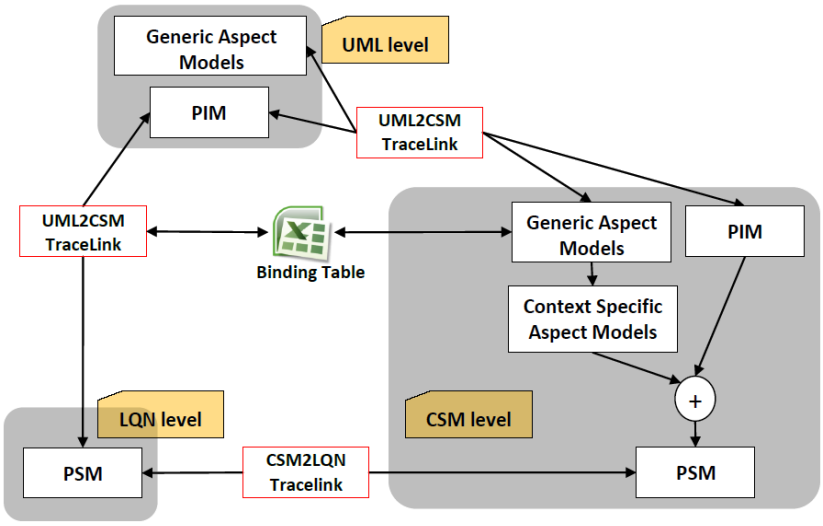


Figure 6.7: The complete traceability between the elements of PUMA4SOA

6.3 Specifications of applying Trace-links in PUMA4SOA

In this section, we describe the specifications used to create and apply the traceability model during a PUMA4SOA model transformation chain based on the traceability metamodel presented before. The specifications used to perform model transformations and to apply traceability between models are very similar. Figure 6.8 describes the specification used to perform the chain of model transformation and to model traceability. Specification 3 describes the chain of performing model transformations and applying trace-links in PUMA4SOA.

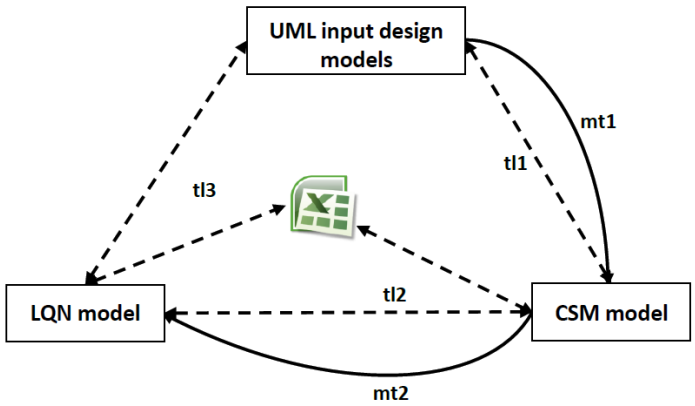


Figure 6.8: The Model Transformation and the Model Traceability relationship

Specification 3: Steps of applying model transformation and trace-links in PUMA4SOA

mt1: Model transformation which maps the elements of the UML input design models and to the CSM model.

tl1: Trace-links are created and applied between the elements of the UML input design model and their equivalent in the CSM model (based on mt1).

mt2: Model transformation which maps the elements of the CSM model and to the LQN model.

tl2: Trace-links are applied between the elements of the CSM model and their equivalent in the LQN model (based on mt2).

tl3: Trace-links are applied between the elements of LQN model and their equivalent in the UML input design models. Any LQN element which does not have an equivalent is linked to binding table (discussed in the previous section). Since the model transformation from the LQN model to the UML is not defined, this step (tl3) is derived from the model transformations mt1 and mt2

When a trace-link is applied between the source and target models, it links the elements and their properties together based on the mapping between the source and target models. Channing the values of the element's properties is not affecting the generated models and trace-links, so it does not require performing the model transformation chain and model traceability again.

We present a generic specification which is used to apply trace-links between the elements of UML, CSM and LQN models. Although the original implementation of the traceability specifications is based on an Eclipse plugin using Java (J2EE), in this research we are using OMG QVT to present a formal description of these specifications.

Let X be an element in the Source model which owns a set of properties: *X{a1, a2, a3 ...}, Y* be an element in the Target model which owns a set of properties: *Y{b1, b2, b3 ...}*

X maps to Y and *a1 maps to b1, a2 maps to b2, a3 maps to b3 ….*

XYTL is the type of the trace-link between element *X* and element *Y*.

Specification 4:Trace-Link between the element x and the element Y


```
Relation XtoY {
    enforce  domain xytl  xy:XLTL {
        source= s:X{ a1=var1, a2= var2, a3= var3, ...};
        target= t:Y{ b1=var1, b2= var2, b3= var3,... };    }
    When {
        domain x s:X{} ;
        domain y t:Y{} ; }
    Where { s.isStereotyped(stereotype); }    }
```

Where stereotype is a variable which defines the owned stereotype of element *X*.

Specification 4 presents the *Relation XtoY* which defines the relationships between the source element *X* and the target element *Y*. The *domain* specifies the related elements of the candidate models. The *Relation* begins with the *when* clause by checking if the elements *X* and *Y* are instantiated. Then, a matching template is enforced by assigning the same variable to the equivalent properties (*a1* is mapped to *b1*, *a2* is mapped to *b2*…). The properties which are defined in the matching template correspond to a certain stereotype. The matching is validated by checking if the source element is annotated with the stereotype that owns the properties defined in the matching template.

As an example, we use specification 4 to define trace-link of type *MessageStepTL* between UML *Message* and CSM *StepType*. A *Message* can be annotated with four different stereotypes:

1. When the message is stereotyped with ‹‹*PaStep*››, the traceability relationship between the *Message* and *StepType* can be one-to-many. This is because the ‹‹*PaStep*›› stereotype has six properties which describe the scenarios, services, and external operations that are called during an execution of the *Message* operation. When those properties are defined, each value in their list represents a step *in the* CSM model. Section 4.2.2 provides more details about the mapping of ‹‹*PaStep*››. The matching template in *MessageToStepType* Relation shows that the message properties: *name*, *hostDemand*, *rep* and *prop* are linked to the *Step* properties: *name*, *hostdemand*, *repCount* and *probability*. The UML *Message* can be also linked to Multiple *Steps* based on the values of the properties: *behavDemands*, *servDemands*, *and extOpDemands*.
2. When the message is stereotyped with ‹‹*PaCommStep*››, the trace-links are defined between the message and additional three steps: *TxOhStep*, *RcvOhStep* and *CommStep*. The properties of those steps are linked to the properties of the *client*, *provider* and *network* nodes.
3. When the message is stereotyped with ‹‹*GaAcqStep*››, an additional trace-link is applied between the message and a resource acquire step.
4. When the message is stereotyped with ‹‹*GaRelStep*››, an additional trace-link is applied between the message and a resource release step.

Trace-links between SD Message and CSM StepType: Service layer

```
Relation MessagetoStepType {
    enforce  domain traceabilityMetamodel  ms: MessageStepTypeTL {
        source= s: Message{name=n, hostDemand= h, rep=r, prob=p,
```

```
                    behavDemands= bd:List(String), behavCount= bc:List(String),
                    servDemands= sd:List(String), servCount= sc:List(String),
                    extOpDemands= ed:List(String), extOpCount= ec:List(String) },
        target= t:List(StepType) { { name=n, hostdemand= h, repCount=r, probability=p },
                bdStep:List{name= bd, repCount= bc},
                sdStep:List{name= sd, repCount= sc},
                edStep:List{name= ed, repCount= ec}  } }
        When { domain uml  s: Message{};
               domain csm t:StepType{};    }
        Where {  s.isStereotyped("Pastep") };    }
```

Trace-links between SD Message and CSM communication StepType: Service layer

```
  Relation MessagetoStepType {
     enforce  domain traceabilityMetamodel  ms1: MessageStepTypeTL {
          source= s: Message{name:= a };
          target= t: StepType{name:= a };    }
     enforce  domain traceabilityMetamodel  ms2: MessageStepTypeTL {
          source= clientNode: Node{CommTxOvh:= tx};
          target= txOvhStep: StepType{hostDemand:= tx};  }
     enforce  domain traceabilityMetamodel  ms3: MessageStepTypeTL {
          source= providerNode: Node{CommRcvOvh:= rcv};
          target= rcvOvhStep: StepType{hostDemand:= rcv};  }
     enforce  domain traceabilityMetamodel  ms4: MessageStepTypeTL {
          source= network: Node{capacity:= c};
          target= commStep: StepType{hostDemand:= msgSize/c};  }
     enforce  domain traceabilityMetamodel  ms5: MessageStepTypeTL {
          source= s: Message{ msgSize:= m};
          target= commStep: StepType{hostDemand:= m/capacity};  }
        When {  domain uml  s: Message{};
                domain uml clientNode: Node{};
                domain uml providerNode: Node{};
                domain uml network: Node{};
               domain csm t:StepType{};
```

```
                domain csm txOvhStep:StepType{};
                domain csm rcvOvhStep:StepType{};
                domain csm commStep:StepType{};    }
        Where { s.isStereotyped("PaCommStep"); }    }
```

Trace-links between SD Message and CSM acquired logical resource AcquireResourceType: Service layer

```
   Relation MessagetoAcquireResourceType {
      enforce  domain traceabilityMetamodel  ms: MessageAcquireResourceTypeTL {
            source= s: Message{name:= a, acqRes:= ac, resUnits:= r };
            target1= t: StepType{name:= a };    }
            target2= rAcquire: ResourceAcquireType{name:= ac, repCount:=r};   }
         When {  domain uml  s: Message{};
                domain csm t:StepType{};
                domain csm rAcquire: ResourceAcquireType{};  }
         Where { s.isStereotyped("GaAcqStep"); }    }
```

Trace-links between SD Message and CSM released logical resource ReleaseResourceType: Service layer

```
   Relation MessagetoReleaseResourceType {
      enforce  domain traceabilityMetamodel  ms: MessageReleaseResourceTypeTL {
            source= s: Message{name:= a, relRes:= ac, resUnits:= r };
            targe1t= t: StepType{name:= a };    }
            target2= rRelease: ResourceReleaseType{name:= ac, repCount:=r};   }
         When {  domain uml  s: Message{};
                domain csm t:StepType{};
                domain csm rRelease: ResourceReleaseType{};  }
         Where { s.isStereotyped("GaRelStep"); }    }
```

A complete set of the trace-links specifications are defined in the technical report at [ALH14].

6.4 Example: Traceability Model of Purchase Order System

We use the Purchase Order case study to apply the trace-links between the elements at the UML, CSM and LQN models. The same case study is used in Chapters 3 and 5 to generate an LQN model from a UML input design model. The same example is also used in [ALH13].

At the UML level, the platform independent model of the PO system defines three UML views: a) the workflow model which describes the processes of receiving, invoicing, scheduling and shipping an order, as in Figure 3.3, on page 43. b) the service architecture model which describe the service structure in the PO system, as in Figure 3.4, on page 45. c) the service behavior models which describe the details about the service invocation. Figure 3.5 (on page 46) shows the behavior details of *ProcessSchedule*. The deployment diagram describes the system configuration by defining the allocation of the artifacts into the host nodes, as in Figure 3.6 (on page 47). The platform aspect model of the *service invocation* describes the structure and behavior of the service platform in a generic format, as Figure 5.2 and 5.3 (on page 91).

At the CSM level, the PIM and the platform aspect model are separately transformed into two CSM model. Figure 5.7 (on page 98) describes the PIM model, which contains the workflow at the CSM top scenario, and the service behavior models as subscenario models. Figure 5.8 (page 99) describes the CSM generic platform aspect model of the *ServiceInvocation*. The AOM approach is applied to generate the composed model which represents the platform specific model (PSM), as in Figure 5.10 (page 103).

At the LQN level, the CSM composed model of the PO system is transformed into the performance model (LQN model), as in Figure 5.15 (page 108). After generating the LQN model, it will be executed using an LQN solver to produce the statistical performance measures of the PO system such as the mean of response time, delay, throughput and service time. The

performance results are used to identify the performance hotspots in the system. The performance results can be fed back to the UML input design model, using the trace-links, to analyze the performance characteristics of the system for further improvements.

The workflow model, in Figure 3.3, is stereotyped with ‹‹*GaAnalysisContext*›› which defines the global variables in the model context and ‹‹*GaScenario*›› which captures the allocation and resource usage at the system-level behavior. The *contextParams* property of ‹‹*GaAnalysisContext*›› defines two variables: *$Users* to indicate the number of users and *$R* to indicate the response time. The *respTime* of the ‹‹*GaScenario*›› defines two values: a required value (*req*) which must not exceed 3 seconds and the calculated value (*cal*) which will be assigned by *$R*. The *throughput* of the ‹‹*GaScenario*›› defines a calculated value assigned by *$T*.

PUMA4SOA can be used for different types of performance analysis, such as the pass/fail, sensitivity analysis and finding the optimal values, which may require multiple iterations to meet the performance requirements. The performance requirement of the PO system specifies that the average response time must not exceed 3 seconds.

In our case, several changes in the model configuration are required to meet the performance requirements. A traceability model is shown in Figure 6.9 which manages these changes during the performance analysis of PO system. The example is simplified by applying trace-links for three UML elements: *POSystem:Activity*, *sales:LifeLine* and *InvoicingHost:Node*.

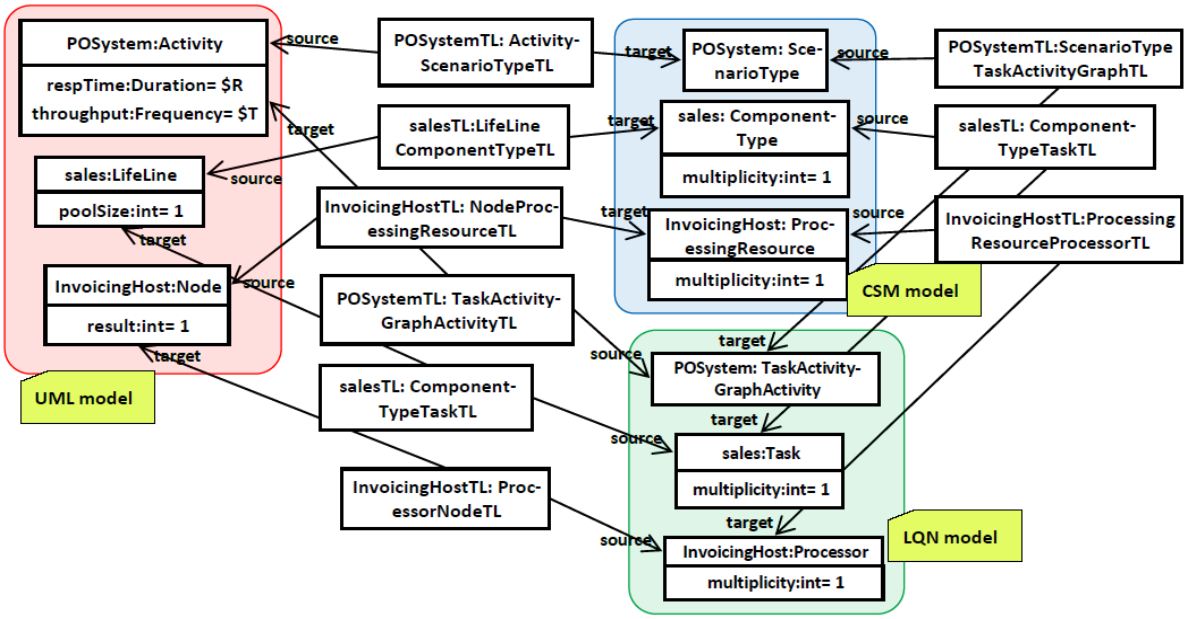


Figure 6.9: Sample of the traceability model of PO System

In order to achieve the performance requirements for the PO system, which is a maximum 3s for the mean response time, three iterations of model transformation chain have been manually executed using *$Nusers*= 100, as shown in Table 6.1.

1. In the first iteration (base iteration), the number of threads for each server and number of cores for each host is initialized to one (*multiplicity* = 1). The results of executing the LQN model show that the mean *response time* of the reference task (*user*) is equal to 7014ms, which does not meet the performance requirements (<=3000ms). The results also show that the hotspot in the PO system is caused by the software bottleneck *sales:Task*. This can be solved by increasing the concurrency level by having more threads in the pool of the *sales* server. So, the *multiplicity* of the *sales* server is recommended to be changed from 1 to 15. The trace-links between the LQN, UML and CSM propagate the changes caused by the first execution; i.e., *response time*, *throughput* and the *multiplicity* of the *sales* server.
2. In the second iteration, the results of executing the LQN model again show that the mean *response time* is equal to 3442ms. The improvement of the response time caused by adding

15 threads to the *sales* task but still does not meet the performance requirements (<=3000ms). The results also show that the hotspot in the PO system is caused by the hardware bottleneck *InvoicingHost:Node*. This can be solved by increasing the number of cores of the *InvoicingHost* processor. So, the *multiplicity* of the *InvoicingHost* processor is recommended to be changed from 1 to 5. The trace-links between the LQN, UML and CSM propagate again the changes caused by the second execution.

3. In the third iteration, the results of executing the LQN model show that the mean response time is equal to 1607ms. The improvement of the response time caused by adding 5 cores to the *InvoicingHost* processor which meets the performance requirements (<=3000ms). So, further iteration is not required. The trace-link between the LQN and UML propagates again the changes caused by the third execution.

Table 6.1: Performance analysis of PUMA4SOA

$Nuser= 100, respTime = {((3,s,mean),req),(($R,s,mean),calc)}				
	UML	**CSM**	**LQN**	**Performance output**
1	POSystem:Activity {respTime=$R}	POSystem: ScenarioType	POSystem: ActivityGraph	user: Task {serviceTime = 7014ms} * Recommendation: sales:Task is bottleneck * Solve software bottleneck → multiplicity=15
	sales: LifeLine {poolSize=1}	sales: ComponentType {multiplicity=1}	sales: Task {multiplicity= 1}	
	InvoicingHost: Node { resMult=1}	InvoicingHost: ProcessingResource {multiplicity=1}	InvoicingHost: Processor {multiplicity=1}	
2	POSystem: Activity {respTime= 7014ms}	POSystem: ScenarioType	POSystem: ActivityGraph	user:Task {serviceTime = 3442ms} * Recommendation: InvoicingHost: Processor is bottleneck. * Solve hardware bottleneck → multiplicity=5
	sales:LifeLine { poolSize= 15}	sales: ComponentType {multiplicity= 15}	sales:Task { multiplicity= 15}	
	InvoicingHost: Node { resMult= 1}	InvoicingHost: ProcessingResource {multiplicity= 1}	InvoicingHost: Processor { multiplicity= 1 }	
3	POSystem: Activity {respTime=3442,ms}	POSystem: ScenarioType	POSystem: ActivityGraph	user: Task {serviceTime = 1607, ms} * Satisfy performance requirements
	sales: LifeLine {poolSize= 15}	sales:ComponentType {multiplicity= 15}	sales: Task {multiplicity= 15}	
	InvoicingHost:Node { resMult= 5}	InvoicingHost: ProcessingResource {multiplicity= 5}	InvoicingHost: Processor { multiplicity= 5}	

Chapter 7: Case studies for Testing and Validation

In this Chapter, we present the testing and validation of the model transformation chain of PUMA4SOA. The validation process is used to measure how close the model is representing the real world with respect to the intended usage of the model [THA04].

In Section 7.1 we present the test cases used for the modeling approach PUMA4SOA. We also present the scope and the limitations of this testing. In Section 7.2, we use our developed tool to present the model transformation steps using an electronic prescription system case study. The tool is developed based on an Eclipse plugin using Java language (J2EE). In Section 7.3, we validate the results produced during the performance analysis of our approach using the Insurance Broker Service System case study which is presented in [LI13].

7.1 Testing of PUMA4SOA model transformation

The model transformation chain of PUMA4SOA begins by modeling the software design of a SOA system using UML. Practically, we use a graphical modeling tool, mainly IBM Rational Software Architecture (RSAv7.0), to model the software design models of SOA systems. We have developed a model transformation tool using Eclipse plugins. The user can either install the tool within RSAv7.0 or within an Eclipse environment (using Eclipse Modeling project) if he does not own an RSA license. If the tool is installed within the eclipse environment, it is easier to model the software design model first using any graphical modeling tool, such as RSA or MagicDraw, and then export it in the form of XML. The exported model is then loaded into our model transformation tool, and then transformed automatically into CSM model in the form of platform independent model.

The tool is also used to perform an automatic aspect composition using AOM approach to generate the CSM platform specific model. The final step in PUMA4SOA is performed manually

where the CSM platform specific model is transformed to an LQN model and then executed to produce the performance results of the software design models. During the model transformation chain, the trace-links are defined between the different models in PUMA4SOA. Until now, only the trace-links between the UML and CSM models are automatically applied.

We use our modeling transformation tool to test the modeling transformation chain of PUMA4SOA. The transformation tool recognizes most of the meta-elements of the UML-SD and UML-AD. Some UML meta-elements that are not recognized do not have definite equivalent elements in the CSM metamodel. These elements are: *Pin*, *InputPin*, *OutputPin*, *ActionInputPin*, *BehavioralFeature*, *Parameter*, *ParameterEffectKind* and *ParameterSet* and *StateInvariant*.

Several test cases have been used to test the model transformations of PUMA4SOA. There are five groups of test cases scenarios described in Table 7.1 and Table 7.2:

1. The business layer scenarios: covers all possible behaviors in the workflow model
    a. Activity with a single action (Orchestration workflow)
    b. Activity with multiple actions (Choreography workflow)
    c. Activity with a *StructuredActivityNode*
    d. Activity with an *ObjectNode*
    e. Activity with a sequence structure
    f. Activity with a decision structure
    g. Activity with a parallel structure
    h. Activity with a structured loop
    i. Activity with an unstructured activity (second order nesting).
2. The service layer scenarios: covers all possible behaviors in the service behavior model

   a. Interaction with a single synchronous message
   b. Interaction with a single asynchronous message
   c. Interaction with multiple synchronous and asynchronous messages
   d. Interaction with combined fragments: op, alt, par, and loop
   e. Interaction with a destruction event
3. Scenarios of adding stereotypes: cover the changes in the behavior caused by defining stereotypes to the model elements
   a. Using ‹‹*PaStep*›› with behavDemands, servDemands, extOpDemands
   b. Using ‹‹*PaStep*›› with noSync=true
   c. Using ‹‹*PaCommStep*››, ‹‹*PaRunTInstance*››, ‹‹*GaExecHost*›› and ‹‹*GaCommHost*››
   d. Using ‹‹*GaAcqStep*›› and ‹‹*GaRelStep*››
4. Scenario of traceability modeling: covers the scenario of applying trace-links between the UML and CSM models
5. Scenarios of aspect composition modeling: covers the scenarios of applying the AOM approach on single and multiple aspects.

At the business layer, several workflow scenarios can be created based on the type of the work flow: orchestration, choreography, structured or unstructured; and the kind of the UML elements used (such as *StructuredActivityNode* and *ObjectNode*). The workflow scenario can be defined in single or multiple lanes to differentiate between the *internal* components of an organization and the other *external* organizations. The workflow scenario also may contain four types of structured workflow: sequence structure, decision structure, parallel structure and structured loop. In structured workflows, each split in the control nodes, whether it is parallel or choice must have the corresponding join element [LIU05]. The workflow scenario is also

described as unstructured when improper flow nestings and mismatched split-join pairs exist [LIU05].

Table 7.1: TestCases for PUMA4SOA Model Transformation

	Business layer									Service layer					Stereotype				
	Activity with a single action (orchestration)	Activity with multiple actions (choreography)	Activity with a sequence structure	Activity with a structured activity node	Activity with an object node	Activity with a decision structure	Activity with a parallel structure	Activity with a structured loop	Activity with an unstructured activity (second order nesting)	Interaction with a single synchronous message	Interaction with single asynchronous message	Interaction with multiple synchronous and asynchronous messages	Interaction with combined fragments: op, alt, par, and loop	Interaction with destruction event	Using *PaStep* with *behavDemands*, *servDemands*, *extOpDemands*	Using *PaStep* with *noSync=true*	Using PaCommStep, PaRunTInstance, GaExecHost and GaCommHost	Using GaAcqStep and GaRelStep	Trace-links between the UML and the CSM models
TC#1	X									X									X
TC#2	X										X								X
TC#3		X		X								X			X		X		X
TC#4		X	X		X							X	X						X
TC#5		X	X			X						X	X						X
TC#6			X				X					X	X			X			X
TC#7			X					X				X	X						X
TC#8			X						X			X		X					X
TC#9		X								X		X			X			X	X
TC#10			X				X	X					X		X		X		X
TC#11		X										X	X		X		X	X	X

At the service layer, several interaction scenarios can be defined based on the type of the message (synchronous and asynchronous), combined fragment type (opt, par, alt and loop) and the type of UML elements added to the scenario.

In the third group, the scenario of the workflow and the interaction are annotated with different stereotypes (such as ‹‹*PaStep*››, ‹‹*PaCommStep*››, ‹‹*GaExecHost*››, ‹‹*GaCommHost*››, ‹‹*GaAcqStep*›› and ‹‹*GaRelStep*››).

Based on the test scenarios described before, we have created 14 test cases as shown in Table 7.1 and 7.2. All test cases passed the testing.

Table 7.2: TestCases for PUMA4SOA Aspect Composition

	Composing a single platform aspect model	**Composing a single platform with nested behaviour**	**Composing multiple aspect model: Service invocation and security**
TC#12	X		
TC#13		X	
TC#14			X

7.2 Case Study: Electronic Prescription System

The electronic prescription system is introduced in [ALH11, PAR10] to handle the process of prescribing a medicine based on the patient medical history and medication factors, then notifying the pharmacy with the created prescription. The business process is handled by two organizations: the clinic and the insurance company. The case study uses the *ServiceInvocation* as a platform aspect model selected from the PC feature model.

7.2.1 At the UML level

The platform independent model of the electronic prescription system describes three UML views: a) the workflow model which describes the actions of requesting the prescription, evaluating the medical history of the patient, the medication factors and the insurance coverage. Finally, the process decides whether to prescribe or not (Figure 7.1); b) the service architecture model which describes the service structure of the electronic prescription system (Figure 7.2); c)

the service behavior models which describe the details about the service invocation. Figure 7.3 describes the behavior details of *Check Payor Coverage*.

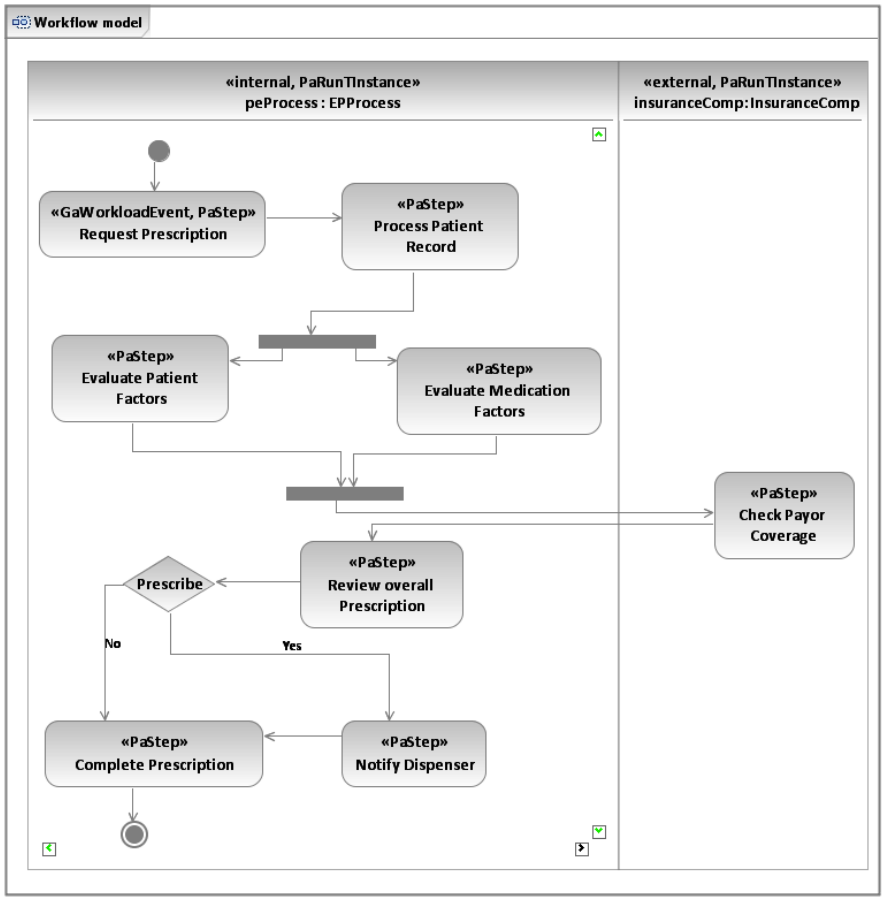


Figure 7.1: The Workflow Model of Electronic Prescription System

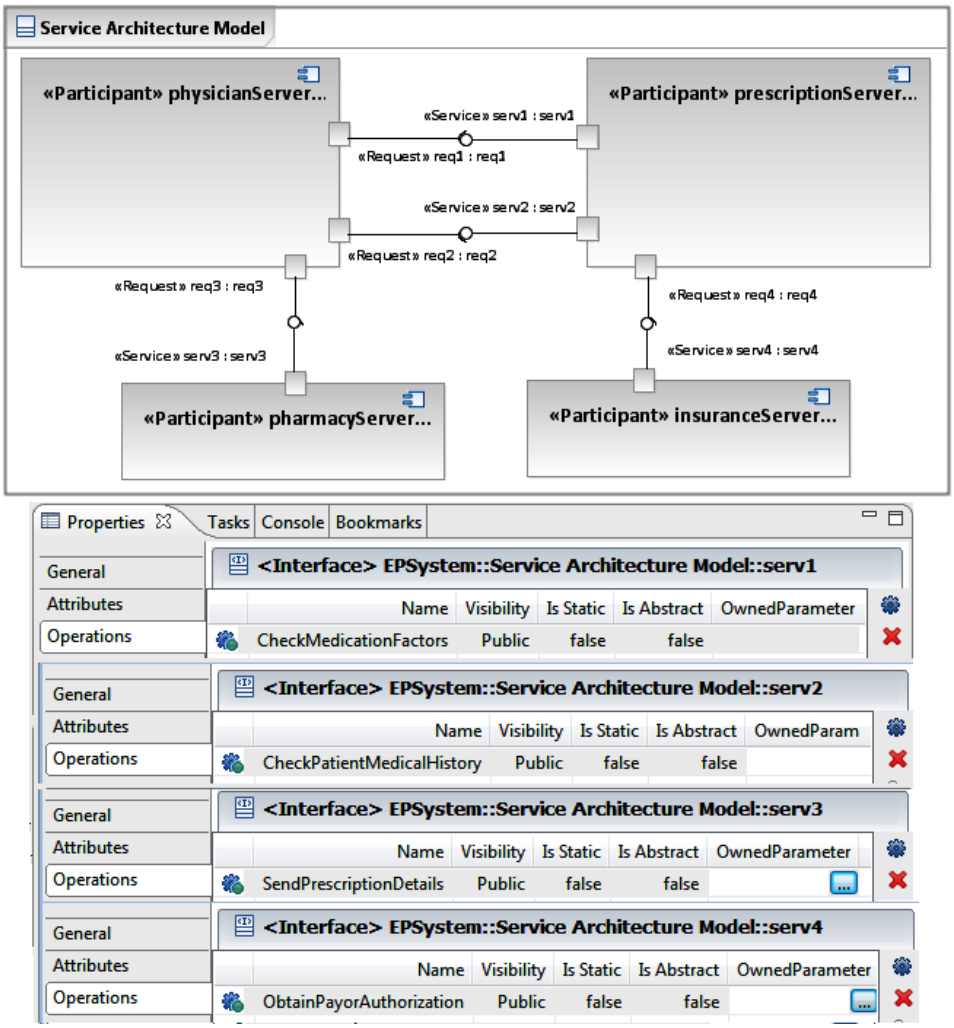


Figure 7.2: The Service Architecture Model of Electronic Prescription System

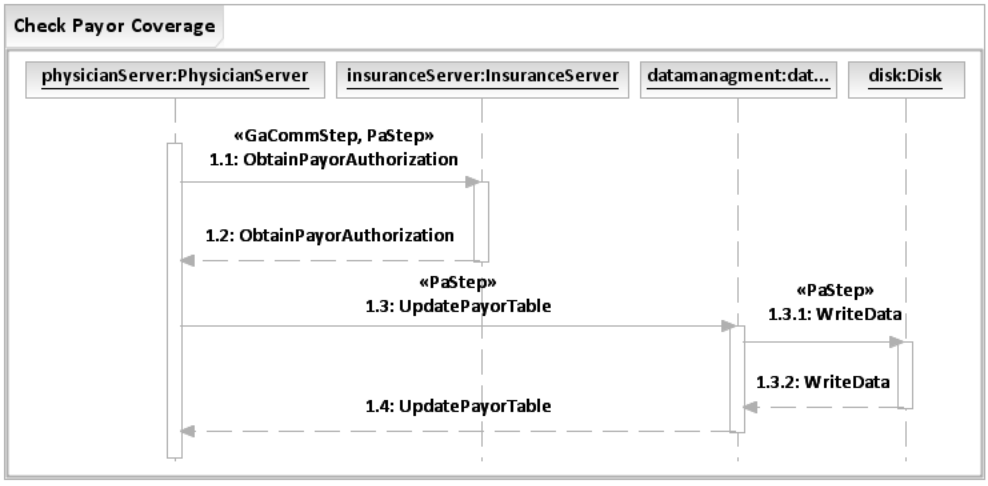


Figure 7.3: The Sequence Diagram of *Check payer coverage*

The deployment diagram describes the system configuration by defining the allocation of the artifacts into the host nodes, as in Figure 7.4.

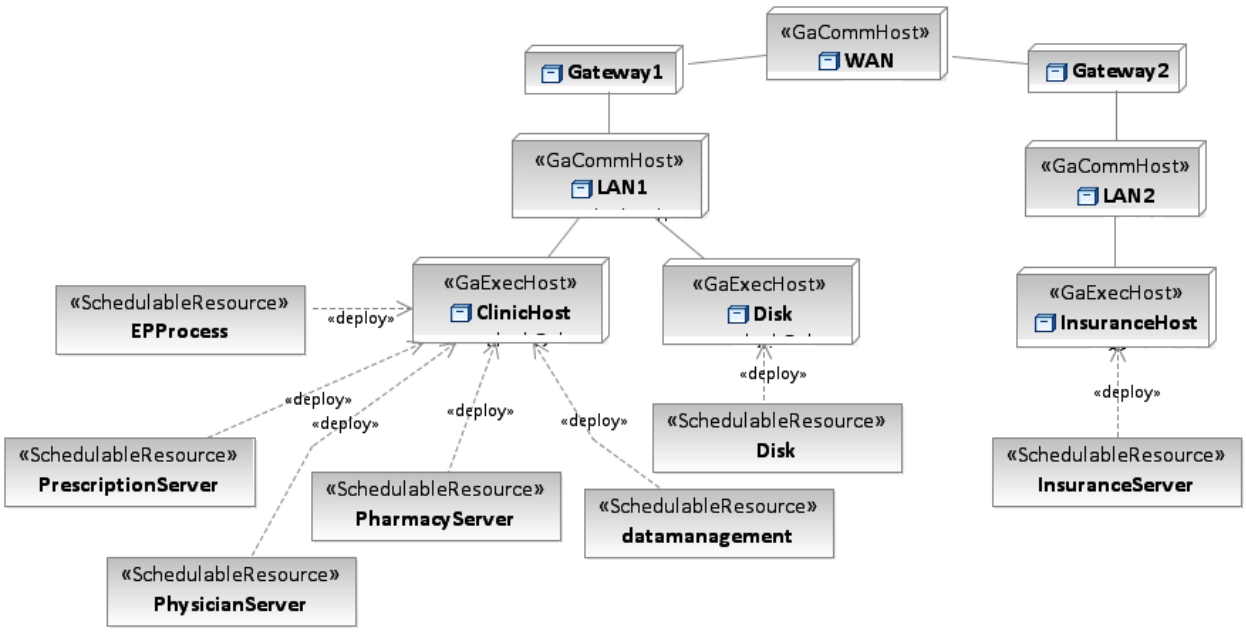


Figure 7.4: The Deployment Diagram of Electronic Prescription System

The platform aspect model of the *service invocation* describes the structure and behavior of the service platform in a generic format (Figures 3.9 and 3.10 on page 51).The *ServiceInvocation* aspect model is selected from the PC feature model. We use IBM RSA 7.0 to model the UML design models.

7.2.2 At the CSM level

The graphical design models of the electronic prescription system are then exported into .uml format and loaded into the PUM4SOA model transformation tool, as in Figure 7.5. The generated CSM models represent the workflow model (Figure 7.6) and the service behavior models (Figure 7.7 describes the *Check Payor Coverage*). Notice that the join-point step

ObtainPayerAuthorization is tagged with ‹‹*ServiceOp*››. The generic aspect models are also loaded and generated the same way (not shown).

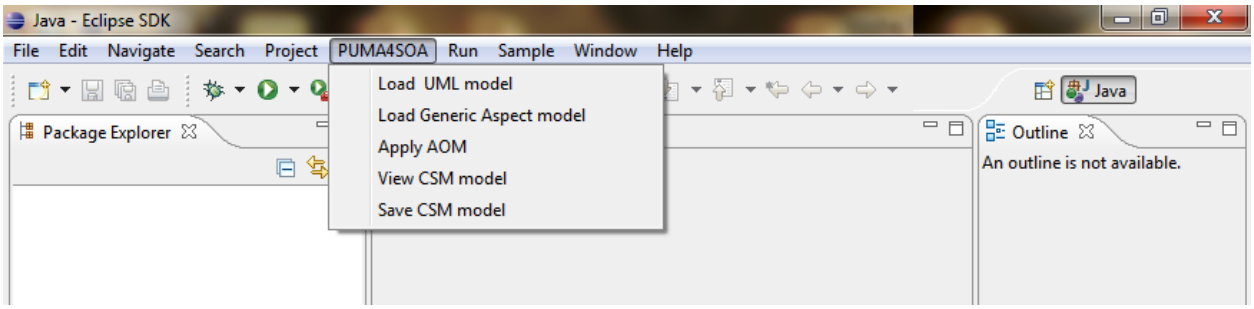


Figure 7.5: The Menu of the PUMA4SOA Tool

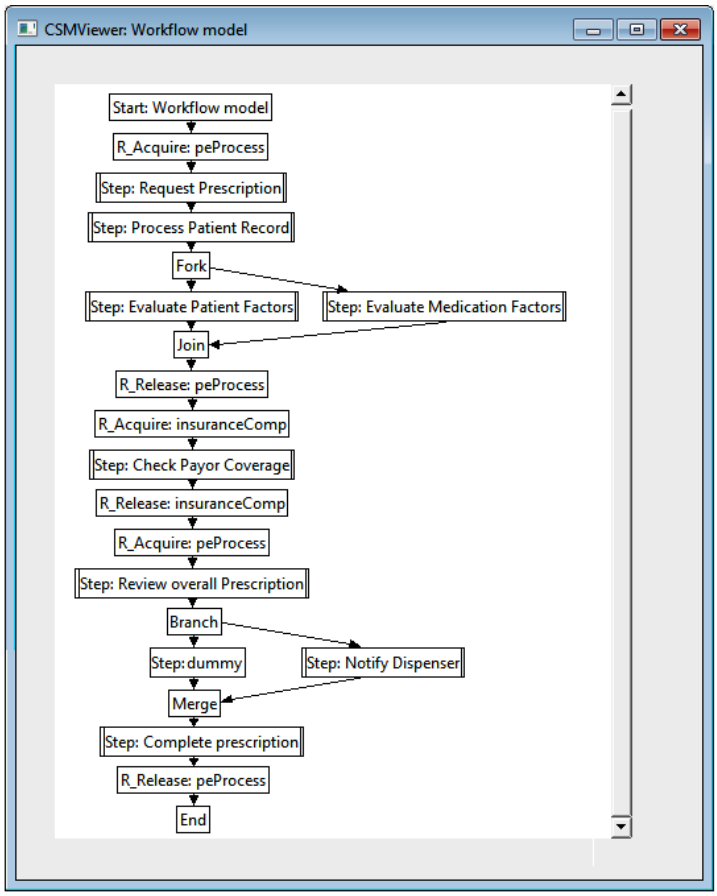


Figure 7.6: The CSM top level describes the Workflow

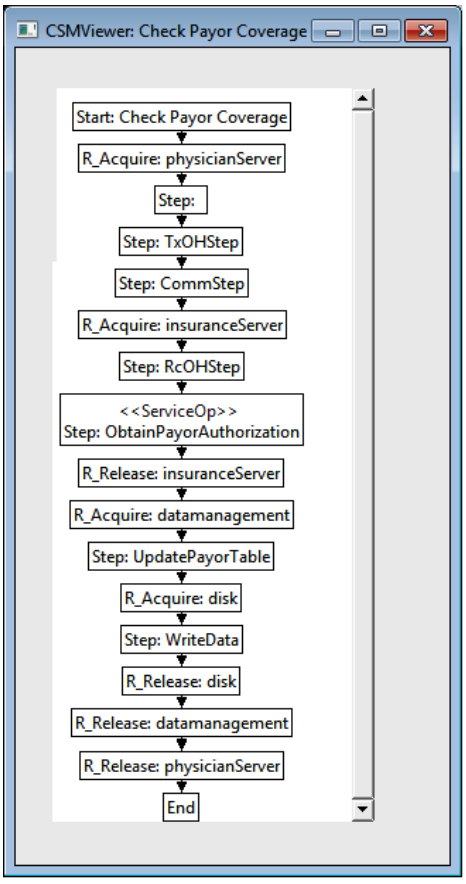


Figure 7.7: The CSM sub-scenario for *Check payor coverage*

After generating the CSM input design models, the AOM is applied to derive the CSM platform specific model of the electronic prescription system. Clicking on "Apply AOM", opens a window which represents the AOM process as in Figure 7.8.

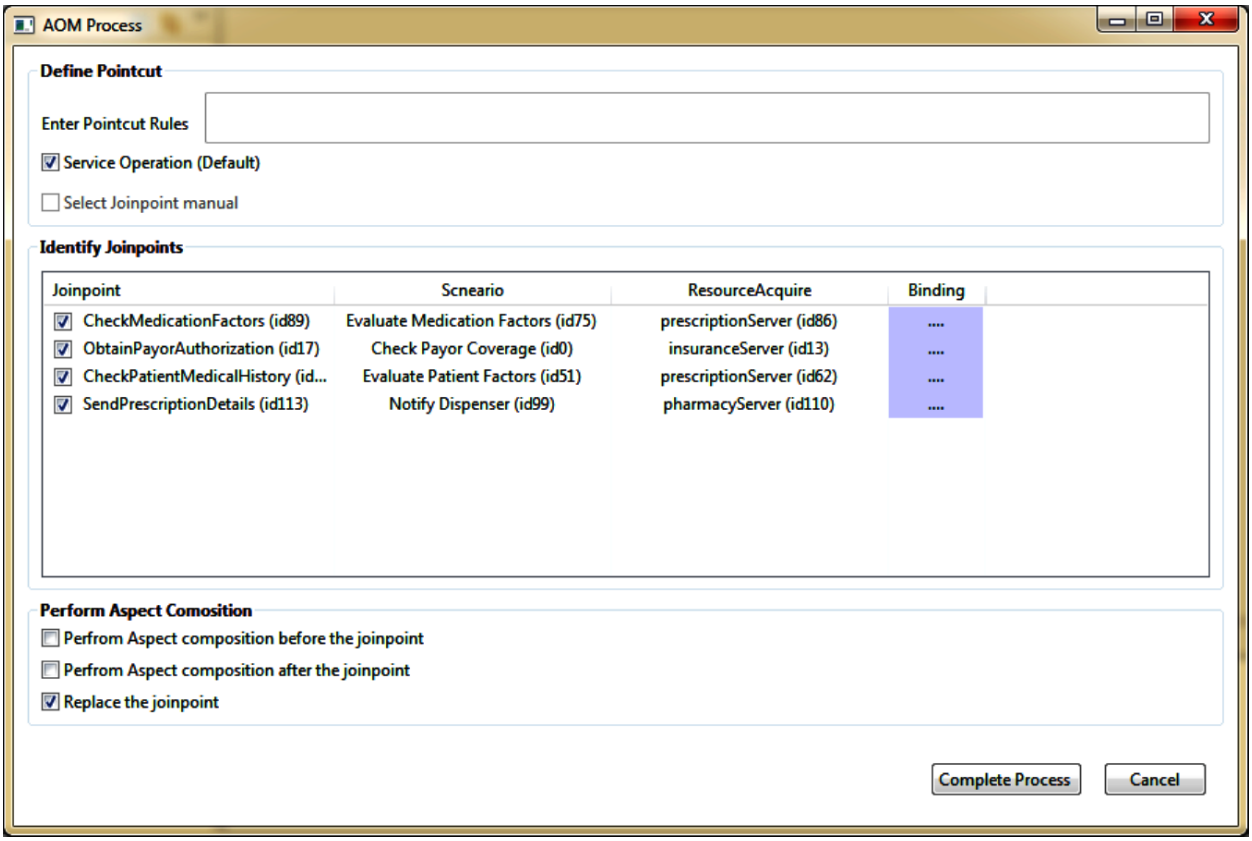


Figure 7.8: The AOM process window in the PUMA4SOA Tool

The AOM process contains three sections: the first section to define the point-cut conditions, the second section to identify the join-point and the third section perform aspect composition. We select the default point-cut conditions which identify the service operation. The table of the join-points identifies four join-points for the electronic prescription system. One of them is *ObtainPayorCoverage* step which is defined within the *Check Payor Coverage* scenario.

Now on every join-point, we need to bind the generic elements of the aspect model with concrete one. Clicking on the binding cell of *ObtainPayorCoverage* join-point, we open the binding window as in Figure 7.9. The binding is performed automatically for some of the

elements, such as |*client* and |*provider*. For the rest of the elements, the binding is performed manually by creating new elements such as |*xmlParserC*. The tool also allows the user to drag and drop concrete elements from the PIM concrete element list. Binding the generic parameters is performed manually by the modeler.

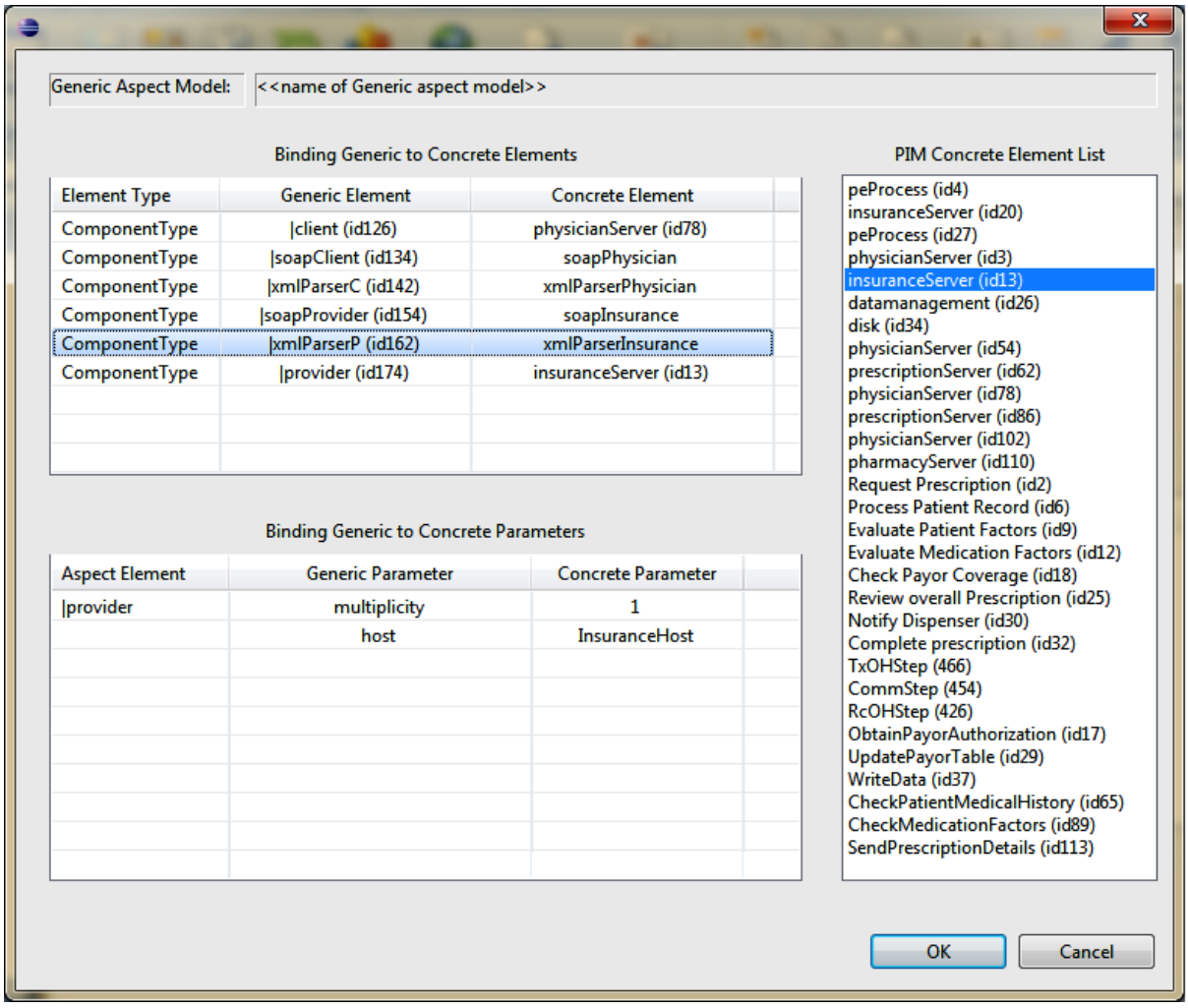


Figure 7.9: The Binding window in the PUMA4SOA Tool

Once the binding is done, we get back to the previous window "AOM process" and clicking on complete process generates the CSM composed model which represents the CSM platform specific model. Figure 7.10 describes the composed model of the *Check Payor Coverage* scenario.

Figure 7.10: The Composed Model for *Check payor coverage*

7.3 Validation of PUMA4SOA model transformation

We use the case study which was introduced in [LI13] to validate our proposed approach. The insurance broker service system is a service-based application that allows the clients to send requests dynamically to different insurance services and pick the most suitable one. The purpose of the system is to allow the construction of different request formats to different service insurance providers and handle different responses. The system is controlled by a BPEL orchestration which orchestrates the execution of different operations on web services involved in the system process.

The central workflow engine (BPEL engine) of the system is implemented using a BizTalk server to compose two insurance web services into a new broker service. The two insurance web services, implemented using ASP .Net, accept quotes from clients and then call the database to reply back to the client. The database is implemented to hold different insurance rates based on certain risk factors such as DOB, Gender, and health conditions. The researchers also injected custom logging code into the two web services and the Biztalk server to get the performance measurements. The execution of the workflow process, which includes the two insurance services, the Biztalk BPEL engine and the database, introduces several overhead like XML request messaging, send/reply request using a SOAP protocol, parsing of SOAP message, execution of business logic, execution of SQL database and disk read/write access.

In [LI13], the insurance broker service system is designed using three models: a) the usecase diagram, which describes the functional requirements from the user's point view, b) the activity diagram, which describes the business logic and c) the BPEL orchestration model to describe the implementation of the business workflow. The performance characteristics are described using MARTE stereotypes.

In order to be able to use PUMA4SOA, we need to describe the insurance broker service system using its UML input design models where the three models are defined: the platform independent model, the deployment diagram and the platform aspect model. In PUMS4SOA, the input design model describes the business layers, the service layer, the deployment specifications and the service platform using separate model abstractions. The design models of the insurance broker service system in [LI13] describe the elements representing the business view, the service view and the configuration view in the same model. In this kind of situations, there is a need to re-model the insurance broker service system in order to extract the details of each model abstraction in a separate view. The re-modeling may require extracting some elements from other elements or aggregating groups of elements into one element. The performance details may also require additional manipulation in order to match with re-modeled design. The key point is that the new design models must maintain the structure, the behaviour and the performance details of the system.

After performing the re-modeling on the insurance broker service system, we obtain the following models: the platform independent model of the insurance broker service system is described using the workflow model (Figure 7.11), the service architecture model (Figure 7.12) and the service behaviour models (Figure 7.13 and Figure 7.14).

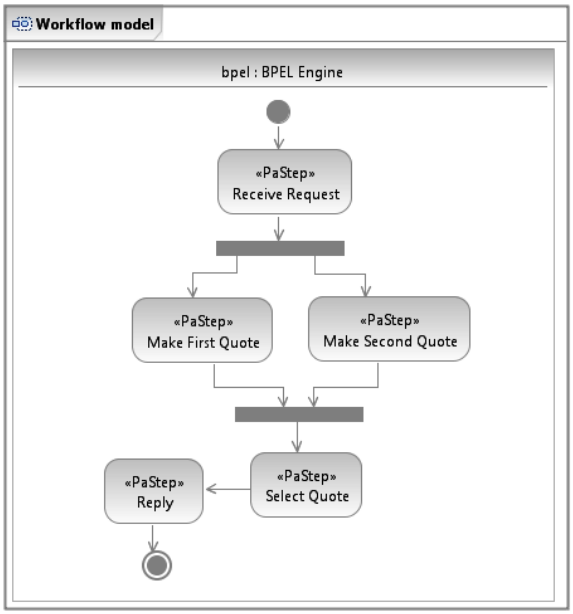


Figure 7.11: The Workflow Model of Insurance Broker Service System

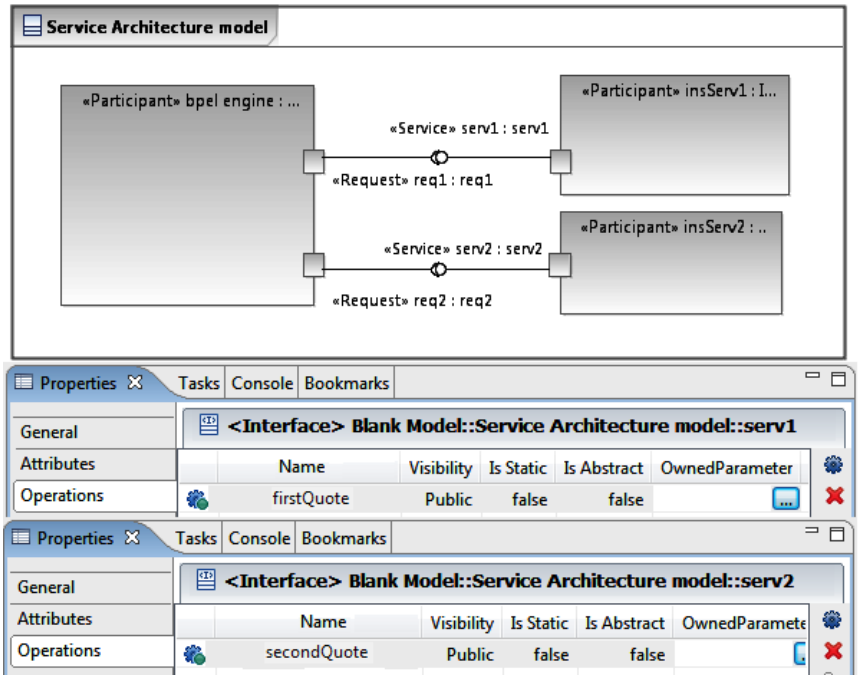


Figure 7.12: The Service Architecture Model of Insurance Broker Service System

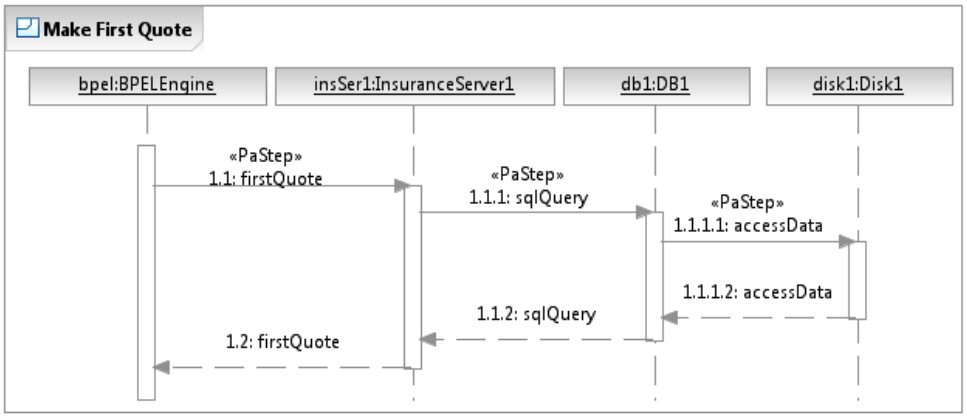


Figure 7.13: The Sequence Diagram of *Make first quote*

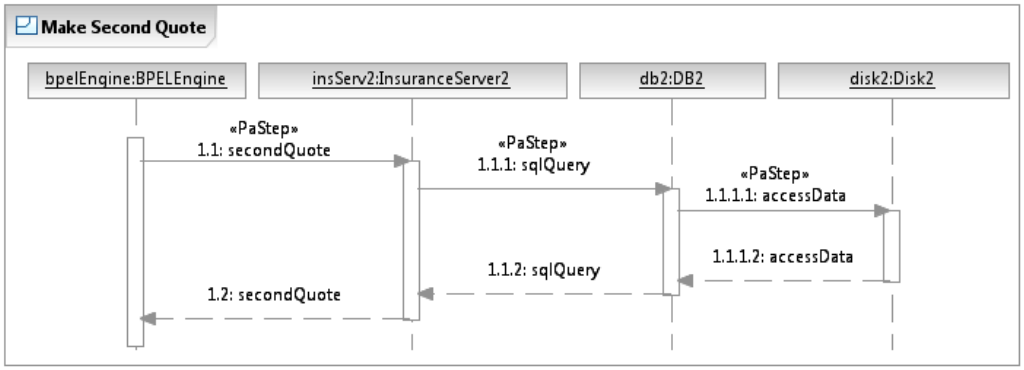


Figure 7.14: The Sequence Diagram of *Make second quote*

The deployment diagram is described in Figure 7.15. The platform aspect model is the service invocation, which is described in Figure 3.9 and Figure 3.10 on page 51.

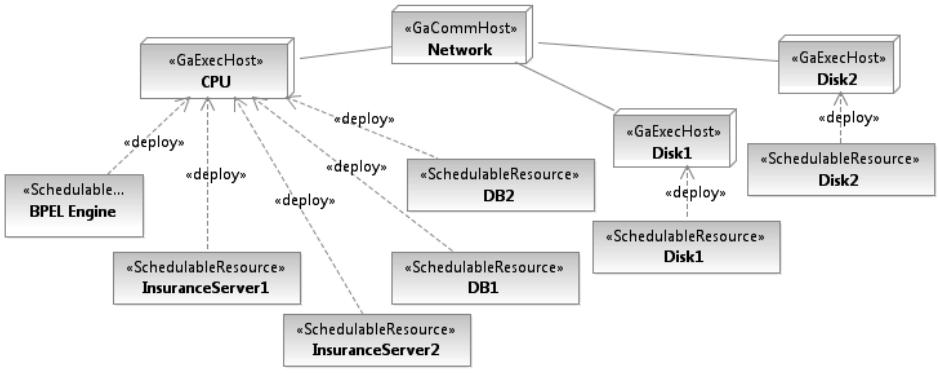


Figure 7.15: The Deployment Diagram of Insurance Broker Service System

7.3.1 Performance analysis

The design models described before are used to generate the LQN model of the insurance broker service system. The LQN model is executed using the LQN solver and then we study the performance results produced by our approach and measured results introduced in [LI13]

The performance model has been generated using the execution demands of system activities obtained in [LI13]. Those demands are used to compute the service time, throughput and utilization of the LQN model, at the system-level and at every element in the model. The execution demands have been validated with measured values obtained by executing the implemented insurance broker service system.

Table 7.3 describes the execution demands of the activities of the original LQN and their equivalent in PUMA4SOA. Some of the elements in the LQN model generated by PUMA4SOA do not have equivalent in the original LQN model introduced in [LI13]. Those elements have been assigned to a small fraction of execution demands to avoid the warnings during the execution of the model.

Table 7.3: The execution demands of the LQN activities

Activities of the original LQN	Activities of LQN generated by PUMA4SOA	Service time (ms), msgSize= 1Kb
receiveRequest	receiveRequest	331.4618512
callSL	requestFirstQuote	0.0001
callML	requestSecondQuote	0.0001
buildXML	marshallingBpel	6
ConstructResponseXml	unmarshalBpel	4.360225141
sendReply	reply	22.93370857
slreceive	requestSoapMessageIns1	432.5059412
slparse	unmarshallingIns1	2.823639775
slwscode	firstQuote	283.095372139

Activities of the original LQN	Activities of LQN generated by PUMA4SOA	Service time (ms), msgSize= 1Kb
slwsresponse		
slquery	sqlQuery1	1872.651657
sldiskaccess	dataAccess1	162.4534084
mlreceive	requestSoapMessageIns2	408.362414
mlparse	unmarshallingIns2	2.563789869
mlwscode	secondQuote	352.337085631
mlwsresponse		
mlquery	sqlQuery2	940.7667292
mldiskaccess	dataAccess2	81.40806754
	MakeFirstQuote	0.0001
	MakeSecondQuote	0.0001
	SelectQuote	0.0001
	marshallingIns1	0.0001
	marshallingins2	0.0001

Table 7.3 is used to generate the LQN model presented in Figure 7.16. The LQN model represents the system in the configuration of single-threaded tasks; all running on a 2-core processor, number of users is ranges from 1 to 40 and message size is 1 Kb. Figure 7.17 shows the difference in the system response time between the measured results and the ones produced by the PUMA4SOA approach. Table 7.4 shows the response times of the performance measurement and the results of the LQN produced by PUMA4SOA approach. The results show that the performance measurement and the LQN model of PUMA4SOA are very close. The average difference in response time is about 7%.

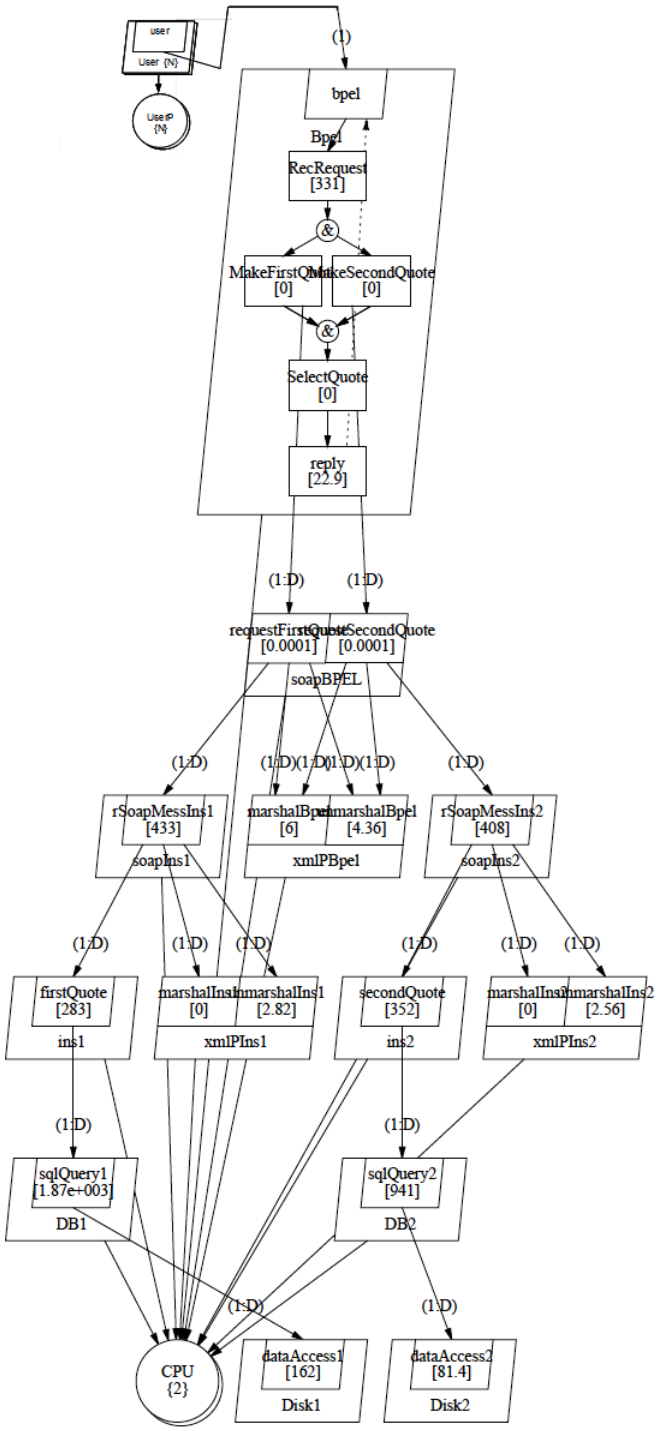


Figure 7.16: The LQN Model generated by PUMA4SOA

Table 7.4: Comparison between the response time of the Performance Measurement and the LQN results

N	Measured RT	LQN Response Time
1	2750	3400.41
5	12390	15095.7
10	30156.2	30941.3
15	45012.6	46833.9
20	57852.3	62738.5
25	72636.8	78645.6
30	91583.6	94554.7

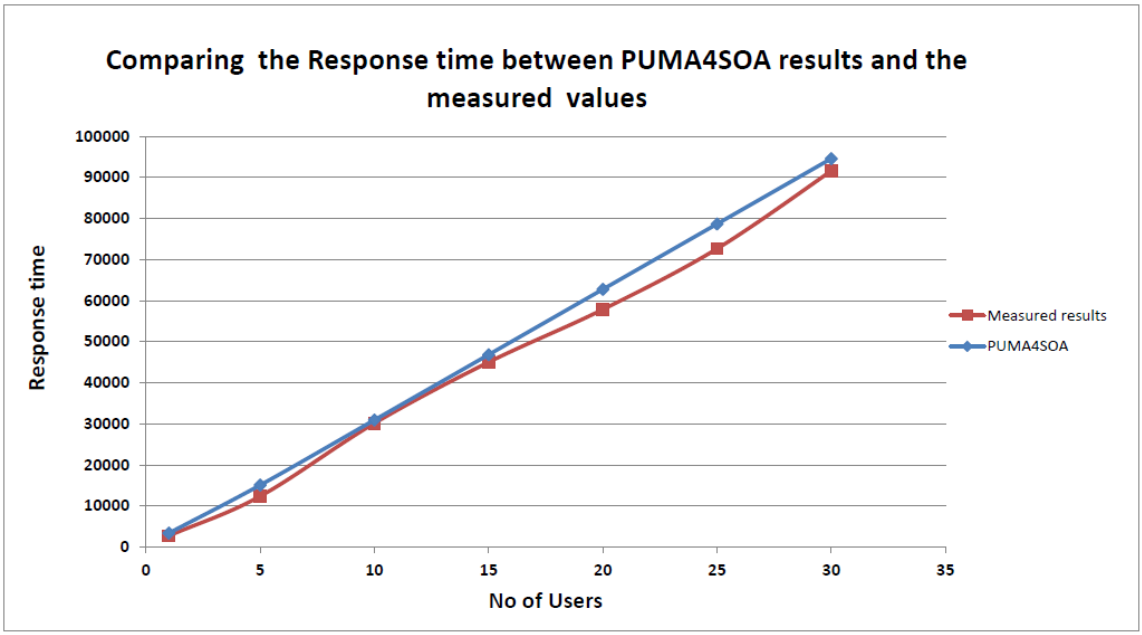


Figure 7.17: The response time of the performance measurement and the LQN model of PUMA4SOA

Chapter 8: Conclusions

The thesis proposes a software approach called Performance from Unified Modeling Analysis for SOA systems (PUMA4SOA) for generating a performance model from the software design models of a SOA system. The main goal of the approach is to study the performance properties of software systems in the early phases of software development.

The proposed approach is an extension of an ongoing project called PUMA. PUMA handles the complexity of making the performance analysis from different UML modeling views of software systems by providing an intermediate modeling language called Core Scenario Modeling (CSM). Several additions to PUMA have been introduced in PUMA4SOA, such as: a) providing additional modeling views to the software design models, b) separating between the platform independent model and the platform specific model to describe the effect of the service middleware behaviour, c) using the platform aspect model to describe the structure and behaviour of the service platform, d) introducing a PC feature model to describe the variability in the service platform; and e) introducing a traceability model for PUMA4SOA, where trace-links are maintained between the model elements of UML, CSM and LQN.

Some of the approaches added to PUMA4SOA, such as the aspect-oriented modeling of platforms, aspect composition, PC feature model, and model traceability are not specific to service-based system and can be used for other classes of software, not just for SOA systems.

The proposed approach is implemented to perform two model transformations. The first implementation transforms a UML design model extended with SoaML and MARTE profiles to a CSM model. The transformation covers structural diagrams (class and deployment) and behavioural diagrams (most of the UML activity and sequence diagrams). The mapping also covers profile stereotypes and their properties. The model transformations are implemented in

Java as an Eclipse plugin. There are more than 50 specifications, presented in Sections 4.2.1, 4.2.2 and the technical report [ALH14], describing the UML to CSM transformation in QVT.

The second implementation is used to transform the CSM platform independent model into CSM platform specific model using the AOM approach. The implementation covers the AOM steps, which are used to perform the aspect composition. There are 7 algorithms, presented in Section 5.2, describing the pseudo-code of this transformation in QVT.

In order to capture certain SOA properties, we had to extend the existing CSM metamodel. First, we used the concept of *Metadata* to extend the CSM metamodel. *Metadata* is a light weight profiling mechanism, able to extend a given metamodel by defining name-value pairs to capture stereotypes. We also introduced a meta-element in the CSM schema called *LogicalResource*, which can be acquired or released through the operations on resources represented by the stereotypes ‹‹*GaAcqStep*›› and ‹‹*GaRelStep*››. The transformation from CSM PIM to PSM based on aspect composition is implemented in Java as an Eclipse plugin. While executing the AOM steps, the modeler must define some of the inputs. We implemented several user interfaces which allow the modeler to easily interact with the modelling tool.

From the beginning, the focus of this thesis was on the transformation from an annotated UML software model to CSM, while the transformation from CSM to LQN was the concern of a different thesis. However, we realized that in some cases it was necessary to specify some aspects of the transformation from CSM to LQN and its traceability model, in order to explain how specific SOA characteristics are handled. The transformation between CSM and LQN has been specified using QVT specifications, but has not been implemented. There are 12 specifications, in Section 4.2.4 and the technical report [ALH14], which describe the CSM to LQN transformation for SOA systems.

As part of the implementation of the transformation from UML to CSM, we have created the PUMA4SOA traceability metamodel. The metamodel defines three groups of trace-links that define the relationships between the elements of UML, CSM and LQN. The metamodel is created using an eclipse project called Eclipse Modeling Framework (EMF). The traceability metamodel is created separately from the metamodels of PUMA4SOA modeling languages (UML, CSM and LQN). There are 80 trace-link types, presented in Section 6.1.1, 6.1.2 and 6.1.3. The traceability metamodel is used to create the trace-links between the elements of UML and CSM during the model transformation. The traceability is implemented as an Eclipse plugin using Java. The trace-links for the other two groups (CSM2LQN and LQN2UML) have been specified but not implemented. There are approximately 80 trace-link specifications presented in Section 6.3 and the technical report [ALH14].

As part of our thesis, we have tested the model transformation and partially the traceability model by using a set of test cases and an electronic prescription system case study. We also validated our approach by comparing the performance results obtained by solving the LQN generated model for an insurance broker service system case study introduced in [LI13] with measurements from an implementation of the same system running on the Biztalk service engine.

8.1 Scope and Limitations

Although our approach has achieved its goals, some limitations have been found due to its modeling features, model transformation capabilities and the usage of traceability model. The following are the limitations of the PUMA4SOA approach:

- An UML activity diagram is used to represents the workflow model, 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, because its metamodel does not provide a

wide range of business process meta-elements compared to other modeling languages, such as BPMN. However, one has to take into account the need to extend the workflow model with performance annotations and the fact that other views of the input model are represented in UML.

- The implementation of the mapping between the UML and CSM metamodels does not cover all the elements of the UML-AD activity and UML-SD. Some of the UML elements do not have clear equivalent in the CSM metamodel.
- Our implementation can handle the transformation of an unstructured activity diagram up to 2^{nd} order nesting. The transformation of an unstructured activity diagram with more than 2^{nd} order nesting to CSM will produce an unexpected (and probably incorrect) result.
- Few of PC feature model relationships and constraints are used; i.e., the mandatory, optional and alternative. A complete propotinal constraints, grammars and consistency checks are not defined.
- Although our approach can be used to generate different kinds of performance models, it is only used in this research to generate an LQN performance model.
- Previous instances of the traceability model which are stored in the physical storage cannot be used any more, so the trace-links need to be generated again when reloading the model.

8.2 Future work

Here is a list of future work directions to improve the tool support for PUMA4SOA:

- Applying modeling constraints and conditions to define the relationships between the aspect features in the PC feature model. The constraints can be used to automate the composition of multiple aspects during the aspect composition process.

- Generating the performance model and analyzing performance of SOA systems which are realized with other middleware technologies (e.g., CORBA, REST, DCOM, etc.)
- Implementing the model transformation from CSM to LQN model in PUMA4SOA.
- Completing the implementation of defining trace-links between: a) the CSM and the LQN models, and b) the LQN and UML models.
- The traceability model of PUMA4SOA, in this research, is used only to propagate changes of the element properties between different models. There is a chance to improve the traceability approach of PUMA4SOA to handle changes caused by adding/deleting elements and changing the behavior.
- Improve the usage of the traceability model in PUMA4SOA by defining a mechanism which allows for the usage of old instances of the traceability model.

Appendices

Appendix A: MARTE stereotypes and their mapping to CSM

UML Stereotype		CSM element		Model Transformation
Stereotype	**Property**	**Element**	**property**	
PaStep		*Step*		
	hostDemand		*hostDemand*	
	prob		*probability*	
	rep		*repCount*	
	blockT		*latency*	
	msgSize			Used to calculate the transmission overheads
	isAtomic			Step if it is true, or Scenario if it is false
	noSync			Described in section 4.2.2 on page 73
	behavDemand			
	behavCounts			
	servDemands			
	servCounts			
	extOpDemand			
	extOpCounts			
PaCommStep		*TxOvhStep* + *commStep* + *RcvOhStep*		Described below
	hostDemand			hostDemand is calculated based on commTxOvh, commRCVOvh, and capacity of ‹‹GaExecHost›› and ‹‹GaCommHost›› .
	prob		*probability*	
	rep		*repCount*	
	blockT		*latency*	
	msgSize			Used to calculate the transmission overheads
	isAtomic			Step if it is true, or Scenario if it is false
	noSync			Described in section 4.2.2 on page 73

UML Stereotype		CSM element		Model Transformation
Stereotype	**Property**	**Element**	**property**	
	behavDemand			
	behavCounts			
	servDemands			
	servCounts			
	extOpDemand			
	extOpCounts			
PaRunTInstance		*Component:*		instance is property of PaLogicalResource
	host		*host*	
	poolSize	*Name=instance*	*multiplicity*	
PaLogicalResource		*LogicalResource*		
	resMult		*multiplicity*	
PaResPassStep		No equivalent mapping, used to *pass the ResourceAcquire*		Pass the resource to another path without releasing it.
	hostDemand		*hostDemand*	
	prob		*probability*	
	rep		*repCount*	
	blockT		*latency*	
	msgSize			Used to calculate the transmission overheads
	isAtomic			Step if it is true, or Scenario if it is false
	noSync			Described in section 4.2.2 on page 73
	behavDemand			
	behavCounts			
	servDemands			
	servCounts			
	extOpDemand			
	extOpCounts			
PaRequestedService		*Step*		
	hostDemand		*hostDemand*	
	prob		*probability*	

UML Stereotype		CSM element		Model Transformation
Stereotype	**Property**	**Element**	**property**	
	rep		*repCount*	
	blockT		*latency*	
	msgSize			Used to calculate the transmission overheads
	isAtomic			Step if it is true, or Scenario if it is false
	noSync			Described in section 4.2.2 on page 73
	behavDemand			
	behavCounts			
	servDemands			
	servCounts			
	extOpDemand			
	extOpCounts			
GaAnalysisContext	*contextParams*	No equivalent mapping, used to define the variable		The list of variables are substituted to their equivalent in the CSM model
GaWorkloadEvent		*Workload*		
	pattern		*openWorkload*	arrivalParam1, arrivalParam2,
			closedWorkload	population
GaExecHost		*ProcessingResource*		
	commRcvOvh			used to calculate hostDemand of RcvOvh step
	commTxOvh			used to calculate hostDemand of TxvOvh step
	resMult		*multiplicity*	
GaCommHost		*CommHost*		
	blockT		*latency*	
	capacity			used to calculate the commStep hostDemand
	resMult		*multiplicity*	
GaScenario		*Scenario*		
	start		*Step*	
	finish		*Step*	
	steps		*list of Steps*	

UML Stereotype		CSM element		Model Transformation
Stereotype	**Property**	**Element**	**property**	
GaAcqStep		*ResourceAcquire*		
	acqRes			The LogicalResource name = acqRes.getname()
	blockT		*latency*	
	hostDemand		*hostDemand*	
	prob		*probability*	
	Rep		*repCount*	
GaRelStep		*ResourceRelease*		
	relRes			The LogicalResource name = relRes.getname()
	blockT		*latency*	
	hostDemand		*hostDemand*	
	prob		*probability*	
	Rep		*repCount*	
SchedulableResource		*Component*		
	host		*host*	
	resMult		*multiplicity*	

References

[ABU07] V. Abu-Eid, “An Aspect Oriented Approach for Applying Features to Web Services”, IEEE International Conference on Web Services (ICWS), DOI 10.1109/ICWS.2007.34, pp 607 - 614, July 2007.

[ALH08] M. AlHaj, “Aspect Composition in Core Scenario Models”, Master Thesis, Department of Systems and Computer Engineering Carleton University, 2008.

[ALH10] M. Alhaj, D. C. Petriu, “Approach for generating performance models from UML models of SOA systems”, CASCON '10 Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research, 2010.

[ALH11] M. Alhaj, “Automatic Generation of Performance Models for SOA Systems”, Department of Systems and Computer Engineering, WCOP '11: Proceedings of the 16th international workshop on Component-oriented programming, pp 33-40, USA, 2011.

[ALH12a] M. Alhaj, D. Petriu, “Aspect-oriented Modeling of Platforms in Software and Performance Models”, In proceeding of: International Conference on Electrical and Computer Systems (ICECS'12), Canada, August 2012.

[ALH12b] M. Alhaj, D. Petriu, “Using Aspects for Platform-Independent to Platform-Dependent Model Transformations”, International Journal of Electrical and Computer Systems Volume 1, Issue 1, Year 2012, DOI: 10.11159/ijecs.2012.005.

[ALH13] M. Alhaj, D. C. Petriu, “Traceability Links in Model Transformations between Software and Performance Models”, SDL 2013: Model-Driven Dependability Engineering, Volume 7916, pp 203-22, Canada, June 2013.

[ALH14] M. Alhaj, D.C. Petriu, “PUMA4SOA: Specifications of model transformations and traceability model”, Technical Report SCE-14-01, Department of System and Computer Engineering, Carleton University, 2014.

[ALM08] E. Al-Masri, Q H. Mahmoud, “Investigating Web Services on the World Wide Web”, WWW '08 Proceeding of the 17th international conference on World Wide Web, pp 795-804, USA, 2008.

[ANY03] K. Anyanwu, A. Sheth, J. Cardoso, J. Miller, K. Kochut’ “Healthcare Enterprise Process Development and Integration”, Journal of Research and Practice in Information Technology, Vol. 35, No. 2, May 2003.

[ARS12] A. Arsanjani, “Service-oriented modeling and architecture: How to identify, specify, and realize services for your SOA”, IBM, Software Group, available from http://www.ibm.com/developerworks/library/ws-soa-design1/ , Nov. 2012.

[BAL04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, “Model-based performance prediction in software development: a survey”, IEEE Transactions on Software Engineering, vol. 30, N.5, pp 295-310, 2004.

[BEC09] S. Becker, H. Koziole, R. Reussner, “The Palladio component model for model-driven performance prediction”, Journal of Systems and Software, Volume 82, Issue 1, January 2009.

[BRE04] P. Bresciani, A. Perini, P. Giorgini, “Tropos: An Agent-Oriented Software Development Methodology”, Journal in Autonomous Agents and Multi-Agent Systems archive, Volume 8 Issue 3, May 2004.

[CHA07] S. Ho Chang and S. Dong Kim, “A Variability Modeling Method for Adaptable Services in Service-Oriented Computing”, SPLC '07 Proceedings of the 11th International Software Product Line Conference, pp 261-268, 2007.

[CLA01] M. Clauss, “Modeling variability with UML”, in GCSE 200–Young Researchers Workshop, September 2001.

[CLA01a] M. Clauss, “Generic Modeling using UML extensions for variability”, in Workshop on Domain Specific Visual Languages at OOPSLA, USA, 2001.

[CLA09] A. Clark, S. Gilmore, M. Tribastone, “Quantitative Analysis of Web Services Using SRMC”, Book Formal Methods for Web Services, pp 296-339 Springer-Verlag Berlin, 2009.

[COL11] M. Colan, “Service-Oriented Architecture expands the vision of Web services”, IBM, available from http://www.ibm.com/developerworks/webservices/ library/ws-soaintro2/, March 2011.

[COR00] V. Cortellessa, R. Mirandola, “Deriving a queuing network based performance model from UML diagrams”, WOSP '00 Proceedings of the 2nd international workshop on Software and performance, pp 58-70, NY, USA, 2000.

[DAM07] A. D'Ambrogio, P. Bocciarelli, “A model-driven approach to describe and predict the performance of composite services”, WOSP '07 Proceedings of the 6th international workshop on Software and performance, pp 78 – 89, USA, 2007.

[DUR11] S. Durvasula, M. Guttmann, A. Kumar, J. Lamb, T. Mitchell, B. Oral, Y. Pai, T. Sedlack, Dr H. Sharma, S. Ram Sundaresan, “SOA Practitioners’ Guide Part 2 SOA Reference Architecture”, available from http://www.soablueprint.com/whitepapers/SOAPGPart2.pdf, Jan 2011.

[EAR05] T. Earl, "Service-Oriented Architecture: Concepts, Technology, and Design", Pearson Education, 2005.

[ECL13] The PEPA Plug-in Project, available from http://www.dcs.ed.ac.uk/pepa/tools/plugin/index.html, June 2013.

[EMF13] Eclipse Modeling Framework available from http://wiki.eclipse.org/EMF ,2013.

[EMI07] C.Emig, K. Krutz, S. Link, C. Momm, S. Abeck, "Model-Driven Development of SOA Services", Cooperation & Management, Universität Karlsruhe (TH), Germany, 2007.

[ERL08] T. Erl, "SOA Principles of Service Design", Prentice Hall, 2008.

[FRA04] R. France, I. Ray, G. Georg, S. Ghosh, "An Aspect- Oriented Approach to Early Design Modeling," IEE Proceedings - Software, Special Issue on Early Aspects: volume 151, issue 4, pp 173-185, August 2004.

[FRA12] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, M. Mroz, "Layered Queuing Network Solver and Simulator User Manual", February 27, 2012, Revision: 10441

[GAL07] I. Galvao, A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering", 11th IEEE International Enterprise Distributed Object Computing Conference, pp 313, 2007.

[GIL06] S. Gilmore, M. Tribastone, "Evaluating the Scalability of a Web Service-Based Distributed e-Learning and Course Management System", Lecture Notes in Computer Science Volume 4184, pp 214-226, 2006.

[GIO05] P. Giorgini, J. Mylopoulos, R. Sebastiani, "Goal-oriented requirements analysis and reasoning in the Tropos methodology", Journal in Engineering Applications of Artificial Intelligence archive, Volume 18 Issue 2, March, 2005.

[GOM06] E. Gómez-Martínez, J. Merseguer, “Impact of SOAP Implementations in the Performance of a Web Service-Based Application”, Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops Lecture Notes in Computer Science Volume 4331, pp 884-896, 2006.

[GRA08] V. Grassi1, R. Mirandola, E. Randazzo, A. Sabetta, "KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability", the Common Component Modeling Example, pp 327-356, Springer-Verlag Berlin, 2008.

[HAP08] J. Happe, H. Friedrich, S. Becker, R. H. Reussner “A pattern-based performance completion for Message-oriented Middleware”, WOSP '08 Proceedings of the 7th international workshop on Software and performance, pp 165-176, 2008.

[HAP10] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, R. Reussner, “Parametric performance completions for model-driven performance prediction“, Performance Evaluation 67(8): 694-716, 2010.

[HIG12] S. Higels, D. Lewis, “State of the Art: Service Composition”, Knowledge and Data Engineering Group, available from http://www.m-zones.org/deliverables/d1_1/papers/4-02-svc_composition.pdf, Jul. 2012.

[HIL96] J. Hillston, “A compositional approach to performance modeling”, Book A compositional approach to performance modelling, ISBN:0-521-57189-8, 1996.

[IBM04] IBM Business Consulting Services, “IBM Service-Oriented Modeling and Architecture”, 2004.

[IMP13] ImPrESS, available from http://www.q-impress.eu/wordpress/, June 2013.

[IPC13] ipc: Imperial PEPA Compiler, available from http://www.doc.ic.ac.uk/ipc/, June 2013.

[ISR05] T. A. Israr, "Transformation of UML 2.0 Models with Performance Annotations to Core scenario Models", Master Thesis, Department of Systems and Computer Engineering Carleton University, Canada, 2005.

[ITU12] ITU-T, Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition, Geneva, Switzerland, 2012.

[KAN90] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study”, Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[KAS08] A.S. Kassim, “Facilitating Web Service Discovery in Distributed Web Service Registries”, Thesis, Carleton University, January, 2008.

[KIE09] J. Kienzle, W. Al Abed, J. Klein, “Aspect-oriented multi-view modeling”, AOSD '09 Proceedings of the 8th ACM international conference on Aspect-oriented software development, pp 87-98, 2009.

[KIE10] J. Kienzle, W. Al Abed, F Fleurey, J. Jézéquel, J. Klein, “Aspect-oriented design with reusable aspect models”, Transactions on aspect-oriented software development VII Springer-Verlag Berlin, Heidelberg, Volume 6210, 2010, pp 272-320, 2010.

[KOL06] D. S. Kolovos, R.F. Paige, F.A.C. Polack, “On-Demand Merging of Traceability Links with Models”, In: From: 3rd ECMDA Traceability Workshop (2006).

[KON09] M. Koning, C. Sun, M. Sinnema, P. Avgeriou, “VxBPEL: supporting variability for Web services in BPEL”. Journal in Information and Software Technology, Volume 51 Issue 2, February, 2009.

[KOZ11] H. Koziolek, B. Schlich, C. Bilich, R. Weiss, S. Becker, K. Krogmann, M. Trifu, R. Mirandola, A. Koziolek, “An industrial case study on quality impact prediction for evolving

service-oriented software" , ICSE '11 Proceedings of the 33rd International Conference on Software Engineering, pp 776-785, 2011.

[LEE09] M.Lee, "Semantic Model-Driven Approach of web Service systems", NISS '09 Proceedings of the 2009 International Conference on New Trends in Information and Service Science, pp 427 – 432, 2009.

[LI13] J. Li, "Performance Measurement and Modeling of BPEL Orchestrations", Mater thesis, Department of Systems and Computer Engineering, 2013.

[LIU05] R. Liu, A. Kumar, "An Analysis and Taxonomy of Unstructured Workflows", Proceedings 3rd International Conference, Springer-Verlag Berlin, pp 268-284, France, 2005.

[LIU08] H Liu, "Transformation of UML 2.0 Models with MARTE to Core Scenario Models", Master Thesis, Department of Systems and Computer Engineering Carleton University, 2008.

[MA09] C. Ma, Y. He, "An Approach for Visualization and Formalization of Web Service Composition", WISM '09 Proceedings of the 2009 International Conference on Web Information Systems and Mining, 2009.

[MAN03] D. J. Mandell and S. A. McIlraith, "Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation", In Proceedings of the Second International Semantic Web Conference (ISWC2003), Volume 2870, 2003, pp 227-241, 2003.

[MAR08] E. A. Marks, M. Bell, "Service Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology", John Wiley & Sons, Nov 3, 2008.

[MAY10] P. Mayer, N. Koch, A. Schroeder, A. Knapp, "The UML4SOA Profile", Technical Report, 2010.

[MEN03] D. A. Menascé, “Security Performance”, IEEE Computer Society, MAY-JUNE 2003.

[MET12] Methodologies Corporation, “Service-Oriented Modeling Framework: An Enterprise Modeling Solution for Business and Technology”, available from http://www.modelingconcepts.com/pdf/SOMF_for_EA.pdf, Nov. 2012.

[MET13] Methodologies Corporation, “Service-Oriented Modeling Framework Language Version 2.0”, available from http://www.modelingconcepts.com/pdf/SOMF_Language_Opt1.pdf , Jan 2013.

[MEY05] H. Meyer, H. Overdick, and M. Weske, “Plængine: A System for Automated Service Composition and Process Enactment”, wscomps05 Proceedings of WWW Service Composition with Semantic Web Services, 2005.

[MOS11] S. Mosser, G. Mussbacher, M. Blay-Fornarino, D. Amyot, “From aspect-oriented requirements models to aspect-oriented business process design models: an iterative and concern-driven approach for software engineering”, AOSD '11 Proceedings of the tenth international conference on Aspect-oriented software development, pp 31-42, 2011.

[MUS09] G. Mussbacher, D. Amyot, “Goal and scenario modeling, analysis, and transformation with jUCMNav”, ICSE Companion 2009, IEEE CS, pp 431-432 2009.

[NAR08] N.C. Narendra, K. Ponnalagu, B. Srivastava and G.S. Banavar, “Variation-Oriented Engineering (VOE): Enhancing Reusability of SOA-based Solutions”, IEEE International Conference on Services Computing, Volume 1, pp 257-264, 2008.

[NAR10] N. C. Narendra, K. Ponnalagu, “Towards a Variability Model for SOA-based Solutions”, SCC '10 Proceedings of the 2010 IEEE International Conference on Services Computing, pp 562-569, USA, 2010.

[OAS02] OASIS Standard, “UDDI Version 2.04 API Specification”, UDDI Committee Specification, 19 July 2002.

[OAS07] OASIS Standard, “Web Services Business Process Execution Language Version 2.0”, 11 April 2007.

[OMG03a] Object Management Group, “Common Object Request Broker Architecture (CORBA) Specification, Version 3.1”, OMG document: formal/2008-01-08, 2008.

[OMG03b] Object Management Group, “MDA Guide Version 1.0.1”, OMG document: omg/2003-06-0, June 2003

[OMG05] Object Management Group, “UML Profile for Schedulability, Performance, and Time Specification”, version 1.1, OMG document: formal/05-01-02, January 2005.

[OMG08] Object Management Group, “Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1”, OMG Document: formal/2011-01-01, January 2011

[OMG09] Object Management Group, “UML: Superstructure specification, Version 2.2”, OMG document: formal/2009-02-02, 2009.

[OMG09a] Object Management Group, “UML: Infrastructure specification‖, Version 2.2”, OMG document: formal/2009-02-04, 2009.

[OMG09b] Object Management Group, “Service oriented architecture Modeling Language (SoaML) Specification”, Version 1.0, March 2012.

[OMG11] Object Management Group, “Business Process Model and Notation (BPMN) Version 2.0”, OMG Document: formal/2011-01-032011, 2011.

[OMG11a] Object Management Group, “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems”, Version 1.1, OMG Document: formal/2011-06-02, 2011.

[OWL13] OWL-S: Semantic Markup for Web Services, available from http://www.w3.org/Submission/OWL-S/, June 2013.

[PAE02] B. Paech, A. von Knethen, "A Survey on Tracing Approaches in Practice and Research", Technical Report IESE Report Nr. 095.01/E, Fraunhofer - Institute of Experimental Software Engineering, 2002.

[PAI11] R. F. Paige, N. Drivalos, D.S. Kolovos, K.J. Fernandes, C. Power,G. K. Olsen, S. Zschaler, "Rigorous identification and encoding of trace-links in model-driven engineering", Software and Systems Modeling (SoSyM), volume 10, issue 4 , pp 469–487, 2011.

[PAR10] C. Partridge & J. Bassi, "Electronic Prescribing Workflow Analysis Handbook v1.0", February 8, 2010.

[PEL10] C. Peltz, "Web Services Orchestration and Choreography", Hewlett-Packard Company, IEEE Computer Society, 2010.

[PER05] M. Pérez Reséndiz, J.Oscar, O. Aguirre, "Dynamic invocation of Web services by using aspect-oriented programming", 2nd International Conference on Electrical and Electronics Engineering (ICEEE) and XI Conference on Electrical Engineering (CIE 2005), pp 48-51, Mexico City, Mexico, September, 2005.

[PET03] D.B. Petriu, D. Amyot, and M. Woodside, "Scenario-Based Performance Engineering with UCMNav", 11th SDL Forum (SDL'03), LNCS 2708, pp18-35, Germany, July 2003.

[PET04] D.B. Petriu, M. Woodside, "A Metamodel for Generating Performance Models from UML Designs", Proc. UML 2004-Modelling Languages and Applications, 7th Int. Conference, Portugal, LNCS 3273, pp 41-53, Springer 2004.

[PET07] D. B. Petriu, M. Woodside, “An Intermediate metamodel with scenarios and resources for generating performance models from UML designs”, Software and Systems Modeling, Volume 6, Nb. 2, pp163-184, 2007.

[PET09a] D.C. Petriu, C.M. Woodside, D.B. Petriu, J. Xu, T.Israr, Geri Georg, Robert France, Siv Hilde Houmb, Jan Jürjens, “Performance Analysis of Security Aspects by Weaving Scenarios Extracted from UML Models,” Journal of Systems and Software, Volume 82, pp 56-74, 2009.

[PET09b] D. C. Petriu, “Software Model-based Performance Analysis”, International School on Model-driven Development for Distributed, Realtime, Embedded Systems, 2009.

[PET12] D. C. Petriu, M. Alhaj, R. Tawhid, “Software Performance Modeling”, Formal Methods for Model-Driven Engineering, Lecture Notes in Computer Science Volume 7320, pp 219-262, 2012.

[PON02] S. R. Ponnekanti and A. Fox, “SWORD: A Developer Toolkit for Web Service Composition”, Proceedings of the 11th International WWW Conference WWW2002 Honolulu, USA, 2002.

[PON08] K. Ponnalagu and N. C. Narendra, “Discovering and Deriving Service Variants from Business Process Specifications”, ICSOC '08 Proceedings of the 6th International Conference on Service-Oriented Computing, Volume 5364, pp 691-707, 2008.

[RAF09] V. Rafe, R. Rafeh, P. Fakhri, S. Zangaraki, “Using MDA for Developing SOA-Based Applications”, ICCTD '09 Proceedings of the 2009 International Conference on Computer Technology and Development - Volume 01, pp 196 - 200, 2009.

[RAM07] E. Ramollari, D. Dranidis, and A. J. H. Simons, “A Survey of Service Oriented Development Methodologies”, the 2nd European Young Researchers Workshop on Service Oriented Computing, 2007.

[RAO04] J. Rao and X. Su, “A Survey of Automated Web Service Composition Methods”, In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC, Volume 3387, 2005, pp 43-54, 2004.

[RIC07] L. Richardson, S. Ruby, “RESTful Web Services Web services for the real world”, O'Reilly Media, 2007.

[ROB02] S. Robak and B. Franczyk, “Modeling Web Services Variability with Feature Diagrams”, Revised Papers from the Node 2002 Web and Database-Related Workshops on Web, Web-Services and Database Systems, Volume 2593, 2003, pp 120-128 , 2002.

[ROB05] S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine, F. Terrier, “The CORBA connector model”, SEM '05 Proceedings of the 5th international workshop on Software engineering and middleware, pp 76-82, USA, 2005.

[SAP06] B. Sapkota D. Roman, D. Fensel, “Distributed Web Service Discovery Architecture”, Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services, Feb. 2006.

[SCH11] S. Schreier, “Modeling RESTful applications”, WS-REST ‘11 Proceedings of the Second International Workshop on RESTful Design, pp 15-21, USA, 2011.

[SER13] Service-oriented Architecture, available at http://www.bestpricecomputers.co.uk/glossary/service-oriented-architecture.htm, Apr. 2013.

[SIN05] S. Singh, J. Grundy, J. Hosking, J. Sun, “An Architecture for Developing Aspect-Oriented Web Services”, ECOWS '05 Proceedings of the Third European Conference on Web Services, 2005.

[SIN06] M. Sinnema, S. Deelstra, P. Hoekstra, “The COVAMOF derivation process”, In Proc. of the 9th International Conference on Software Reuse, Vol. 4039, pp. 101-114, 2006.

[SIR05] E. Sirin and B. Parsia and J. Hendler, “Template-based Composition of Semantic Web Services”, In AAAI Fall Symposium on Agents and the Semantic Web, pp 85-92, 2005.

[SKO04] D. Skogan, R. Grønmo, I.Solheim, “Web Service Composition in UML”, EDOC '04 Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International, pp 47 – 57, 2004.

[SMC13] SMC: The State Machine Compiler, available from http://smc.sourceforge.net/, June 2013.

[TAG09] F. Taghizadeh1, S. R. Taghizadeh, “A Graph Transformation-Based Approach for applying MDA to SOA”, FCST '09 Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology, pp 446 – 451, 2009.

[TAW11] R. Tawhid, and D.C. Petriu, “Automatic Derivation of a Product Performance Model from a Software Product Line Model”, Proc. of the 15th International Conference on Software Product Line (SPLC’11), Germany, pp 80 – 89, 2011.

[THA04] B. H. Thacker, S. W. Doebling, F. M. Hemez, M. C. Anderson, J. E. Pepin, E. A. Rodriguez, “Concepts of Model Verification and Validation”, National Nuclear Security Agency (NNSA), 2004.

[TRI08a] M. Tribastone, S. Gilmore, “Automatic Translation of UML Sequence Diagrams into PEPA Models”, Fifth International Conference on Quantitative Evaluation of Systems, pp 205-214, 2008.

[TRI08b] M. Tribastone, S. Gilmore, “Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile”, Proceedings of the 7th international workshop on Software and performance (WOSP '08), pp 67-78, USA, 2008.

[TRI10] M. Tribastone, P. Mayer, M. Wirsing, “Performance prediction of service-oriented systems with layered queuing networks”, ISoLA'10 Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II, Springer-Verlag Berlin, 2010.

[UDD04] UDDI Version 3.0.2, “UDDI Spec Technical Committee Draft”, October 2004.

[VER05] T. Verdickt, B. Dhoedt, F. Gielen, P. Demeester, “Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models”, IEEE Transactions on Software Engineering, Vol. 31, No. 8, August 2005.

[WOO02] C.M. Woodside, D.C. Petriu, K.H. Siddiqui, “Performance-related Completions for Software Specifications”, Proceedings of the 22rd Int Conference on Software Engineering, ICSE 2002, pp 22-32, Orlando, Florida, USA, 2002.

[WOO05] M. Woodside, D.C. Petriu, D.B. Petriu, H. Shen, T. Israr, J. Merseguer, “Performance by Unified Model Analysis (PUMA)”, WOSP '05 Proceedings of the 5th international workshop on Software and performance, 2005.

[WOO08] C.M. Woodside, D.C. Petriu, J. Xu, T. Israr, J. Merseguer, “Methods and Tools for Performance by Unified Model Analysis (PUMA)”, Technical Report SCE-08-06, Carleton University, Systems and Computer Engineering, pp. 35, 2008.

[WOO09] M. Woodside, D.C. Petri , D.B. Petriu , J. Xu, T. Israr, G. Georg, R. France, J.M. Bieman, S. H. Houmbc, J. Jürjens, “Performance analysis of security aspects by weaving scenarios extracted from UML models”, Journal of Systems and Software, Volume 82, Issue 1, January 2009.

[WOO13] M. Woodside, D. C. Petriu, J. Merseguer, D. B. Petriu, M. Alhaj, “Transformation challenges: from software models to performance models”, Software and System Modeling, (DOI: 10.1007/s10270-013-0385-x), Springer, 2013.

[WSD01] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001

[XU03] J. Xu, C.M. Woodside, D.C. Petriu, “Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time,” Proc. of TOOLS'2003, Springer, Volume 2794, pp 291-307, 2003.

[XU07] Y. Xu, S. Tang, Y. Xu, Z. Tang, “Towards Aspect Oriented Web Service Composition with UML”, 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS), pp 279- 284, 2007.

[ZEN05] Y.X. Zeng, “Transforming Use Case Maps to the Core Scenario Model Representation”, M.C.S. thesis, University of Ottawa, Canada, June 2005.

[ZHA09] C. Zhao, Z. Duan , M. Zhang, “A Model-Driven Approach for Dynamic Web Service Composition”, WCSE '09 Proceedings of the 2009 WRI World Congress on Software Engineering , Volume 04, pp 273- 277, 2009.

[ZIM12] O. Zimmermann, P. Krogdahl, C. Gee, “Elements of Service-Oriented Analysis and Design: An interdisciplinary modeling approach for SOA projects”, available from http://www.ibm.com/developerworks/webservices/library/ws-soad1/ , Nov. 2012.