Exploring SOA Pattern Performance using Coupled Transformations and Performance Models

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

***Abstract*—Service Oriented Architecture (SOA) patterns can be applied to improve different qualities of SOA designs. The performance impact of a pattern (improvement or degradation) may affect its use, so we assess its impact by automatically generated performance models for the original design and for each candidate pattern and pattern variation. This paper proposes a technique to incrementally propagate the changes from the software to the performance model. The technique formally records the refactoring of the design model when applying a pattern, and uses this record to generate a coupled transformation of the performance model. The SOA design is modeled in UML extended with two profiles, SoaML and MARTE; the patterns are specified using Role Based Modeling and the performance model is expressed in Layered Queuing Networks. Application of the process, and pattern performance exploration, is demonstrated on a case study.**

Keywords- Software performance model, service oriented systems, SOA pattern, coupled transformation, LQN

I. INTRODUCTION

In designing SOA (Service Oriented Architecture) systems, SOA patterns [1] are proposed as generic solutions to problems in the architecture, design and implementation. The patterns may have a substantial impact on performance, and we wish to evaluate this with a performance model (PModel) generated automatically from a software design model (SModel) and the pattern description. The baseline PModel may be created by an automated transformation as in PUMA (Performance from Unified Model Analysis) [2]. The automated refactoring of the PModel to reflect application of a pattern, using coupled transformations of the SModel and the PModel, is the subject of this work. Automating the refactoring makes it easier to consider the performance issues, and to rapidly consider a (possibly large) set of variations on a pattern. It also reveals the causal connections between the design changes and the performance issues, which may be of value to the designer. Manually refactoring the SModel and then regenerating the PModel using PUMA is a viable alternative but may suffer from inconsistency in the refactoring. In [3] we studied the impact of SModel changes to PModel due to application of SOA design patterns.

This research describes a coupled transformation technique to incrementally propagate design changes to the PModel by: (A) definition of the pattern using a role-based modeling technique; (B) formal recording the SOA design refactoring; (C) automatic derivation of the corresponding performance model changes; (D) application of the changes to the PModel. This paper describes (A) – (C) but does not address the implementation of the transformation in step (D). The SOA SModel is captured in UML with the OMG profiles SoaML (Service Oriented Architecture Modeling Language) [4] and MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [5] for performance information. The Role Based Modeling Language (RBML) [6] is used to formally define each SOA design pattern in terms of first, the set of SModel elements that represent the *problem* addressed by the pattern and second, those that constitute the *solution*. The novel contributions of this work are the coupled transformation in Section VI, and the process (systematic and automatic) that supports its use, including the formal recording of the changes for SModel refactoring.

The paper is organized as follows: Section II presents related work, Section III surveys the approach; Section IV describes the models; Section V describes the SModel transformation rules; Section VI presents the coupled transformation; Section VII describes a case study. Finally Section VIII concludes the paper.

II. RELATED WORK

The relationship of PModels to SModels, and the derivation of one from the other, is the subject of considerable work, including diverse target PModel types such as Queuing Networks, Layered Queuing Networks (LQNs) [7] and Stochastic Petri nets [8]. The general approach of PUMA integrates diverse types of PModel and SModel [2]. This work uses it with UML SModels (for the SOA designs) annotated with MARTE, and LQN PModels. The SModel-to-PModel mapping of [9] is extended here to support the coupling of the refactoring transformations.

The impact of design patterns on software performance has been studied only indirectly, through the concept of performance anti-patterns, introduced in [10]. Anti-patterns are defined as common design errors that cause undesirable results. An approach based on anti-patterns for identifying performance problems and removing them is described in [11]. An OCL query is created to identify each anti-pattern and applied to the design model. The anti-pattern removal is special for each anti-pattern and is not automated.

Xu [12] described a rule-based system which discovered performance problems and automatically improved the

DOI reference number: 10.18293/SEKE2015-140

design as represented by the PModel. However the rules are slightly different from patterns or anti-patterns and the changes were not propagated automatically to the SModel.

III. PROCESS OVERVIEW

The overall process is shown in Figure 1. This paper describes stages B and C, shown in grey. The inputs include a SOA SModel (top left), and a library of pattern definitions with formal roles (bottom left). The designer steps are given on the left and the automated steps on the right side.

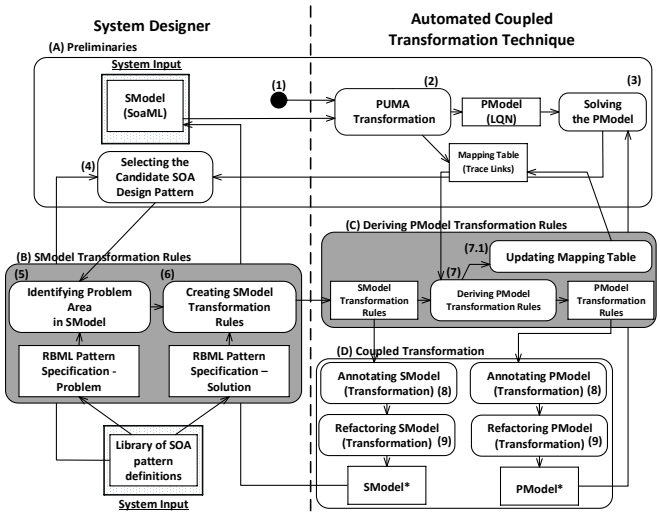


Figure 1: Proposed Approach Overview

The designer steps are supported by tools that have been implemented in this work. There are four stages:

A) Preliminaries: This stage uses the SModel to create the base PModel using PUMA, and creates the SModel/PModel mapping table. Pattern application begins at step (4), where the designer selects a candidate pattern for its own reasons (e.g. maintainability).

B) Model Transformation Rules: The selected pattern is specified using RBML. The designer indicates where the pattern is applied by binding pattern roles to entities in the SModel (step (5)) and then specifies SModel transformation rules that will satisfy the solution specification (step (6)).

C) Deriving the PModel Transformation Rules: Using the mapping table from (A) and the SModel transformation rules from (B), the PModel transformation rules are derived automatically.

D) Coupled Transformations: Both sets of transformation rules are executed via coupled transformations to refactor the SModel and PModel into SModel* and PModel*, respectively. The PModel* results can be used to select the pattern to be applied. Therefore, Stages B, C and D may be repeated until the designer gets the desired results.

IV. MODELS

A. SOA Models

From the range of diagrams used to model SOA systems, we use the Business Processes Model (BPM) for behavior and the Service Architecture Model (SEAM) for structure and contracts, together with a UML deployment diagram. Figure 2 shows examples.

The BPM is specified as a UML activity diagram (Figure 2.B). Service invocations are modeled as operation calls, using three types of UML actions: a *CallOperationAction* transmits a request to the target and waits for the reply via its input/output pins; an *AcceptCallAction* is an accept event action waiting for the arrival of a request; and a *ReplyAction* returns the reply values to the caller. The called operation appears in parentheses after the action name as *"(class-name::operation-name)"*. We assume all BPM edges between ActivityPartitions are between these three Action types and represent calling interactions.

Performance information by MARTE annotations are given in shaded notes. They describe the behavior as a sequence of steps *«PaStep»* with a workload attached to the first step (*«GaWorloadEvent»*). *«PaStep»* has attributes *hostDemand* (the required CPU time), *rep* (the mean repetitions) and *prob* (its probability if it is an optional step). The workload *«GaWorloadEvent»* defines a population of *Nusers* users, each with a thinking time *ThinkTime* defined by MARTE variables. Concurrent runtime instances *«PaRunTInstance»* are identified with swimlane roles.

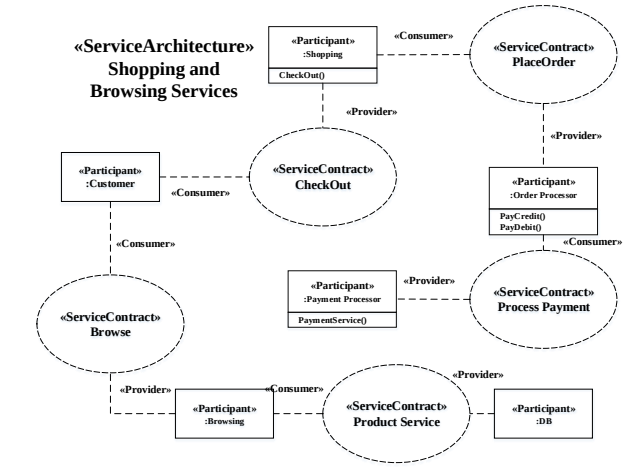
The SEAM is specified as a UML collaboration diagram (Figure 2.A) with service participants and contracts (stereotyped *«Participant»* and *«ServiceContract»* respectively; these are not from MARTE but are specific to this process). Each participant plays a role of Provider or Consumer with respect to a contract. Participants correspond to pools, participants and swimlanes in the BPM.

Deployment is also defined, as in Figure 2.C. Processing nodes are stereotyped *«GaExecHost»* and communication network nodes are stereotped *«GaCommHost»*, with attributes for processing capacity, message latency and communication overheads.

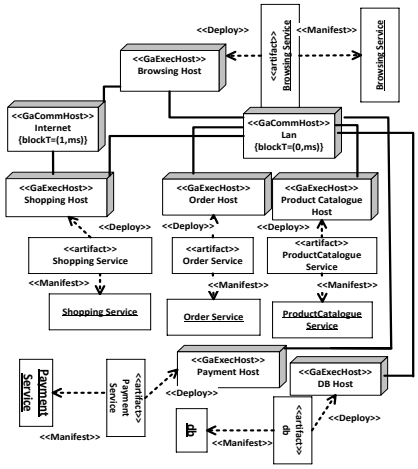
B. Performance Models

PModels are expressed in an extended queueing notation called Layered Queuing Networks (LQNs) [2], selected because of its close coupling to the high-level software architecture. An LQN estimates waiting for service due to contention for host processors and software servers, and provides response time and capacity measures.

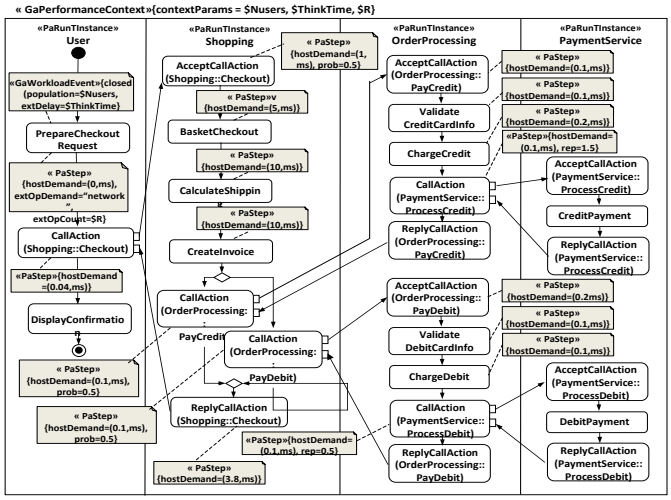
Figure 2.D shows the LQN model for the example. For each service there is a task, shown as a bold rectangle, and for each of its operations (contracts) there is an entry, shown as an attached rectangle. The task has a parameter for its multiplicity or thread pool size (e.g. {'1'}). Each entry has a parameter for its host CPU demand, equal to the total *hostDemand* of the set of *«PaSteps»* for the same operation in the SModel. Calls from one entry to another are indicated by arrows between entries (a solid arrowhead indicates a synchronous call for which the reply is implicit, while an open arrowhead indicates an asynchronous call). The arrow is annotated by the number of calls per invocation of the sender. For deployment, an LQN host node is indicated by a round node associated to each task.



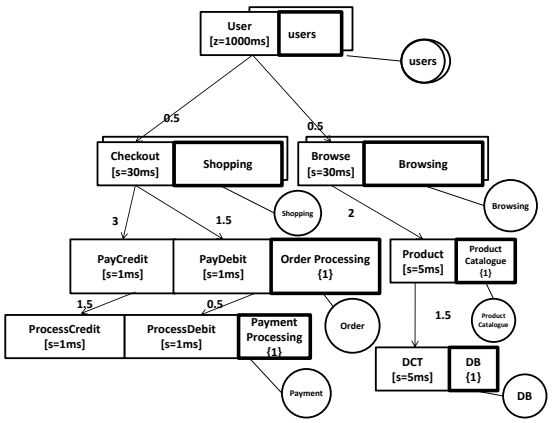
(A) Service Architecture Model (SEAM) for online shop



(C) UML Deployment Diagram for online shop



(B) Checkout Business Process Model for the Online Shop



(D) LQN (PModel) corresponding to (A) (B) and (C)

Figure 2 : SModel views (A, B, and C) and PModel (D)

C. SModel to PModel Mapping Table

When the PModel is derived from the SModel using the PUMA [2] process, the mapping between the corresponding elements of the two models is recorded as described in [9], extended to identify a set of Actions initiated by a Call (an ActivitySet), and pairs of Call and Reply Actions. There are three mapping sub-tables, for StructuralElements, Calls, and Attributes. Each row in a table represents a link between an SModel element or set and a corresponding PModel element (because the PModel is more abstract, one element may correspond to a set of SMEs). Table 1 shows a few of the traceability links for the example in Figure 2.

D. Role-Based Models for SOA Patterns

To formalize the definition of SOA design patterns without resorting to a new language, we use Role-Based Modeling RBML [6], where the pattern is expressed with generic roles acting as formal parameters which must be bound to actual parameters from the application context to which the pattern is applied.

Table 1: Partial Mapping Table between SModel and PModel for Shopping and Browsing

Sub-table (A) StructuralElements Trace Links		
Link	**Set of SModel Elements**	**PModel Element**
DTL3	Deployment Node: Order	LQN Host: Order
DTL2	Deployment Artifact: Browsing	LQN Task: Browsing
BTL1	ActivitySet: Checkout = {AcceptCall, BasketCheckout, CalculateShipping, CreateInvoice, CallOperation(OrderProcessing::PayCredit), CallOperation(OrderProcessing::PayDebit}, Reply}	LQN Entry: Checkout
Sub-table (B) Calls Trace Links		
Link	**Set of SModel Calls**	**PModel Call**
BCTL1	Call fromCallOperationAction(Shopping::Checkout) to AcceptCallAction (Shopping::Checkout) and the corresponding reply from ReplyAction(Shopping::Checkout) back to CallOperationAction(Shopping::Checkout)	LQN synchronous Call from Entry:User to Entry: Checkout

Three UML views are used for each pattern: **BPS** (Behavioral Pattern Specification) for behavior, corresponding to the BPM; **SPS** (Structural Pattern Specification) for structure, corresponding to the SEAM; and **DPS** (Deployment Pattern Specification), not described here due to space limitations. Each view has two specifications: *Pattern Problem* (the view before pattern application) and *Pattern Solution* (after application). Figure 3 shows the role-based specification for the Service Façade pattern (which is described in Section VII), with the *problem* on the left and the *solution* on the right. As in [6] the names of generic roles start with the character '|'.

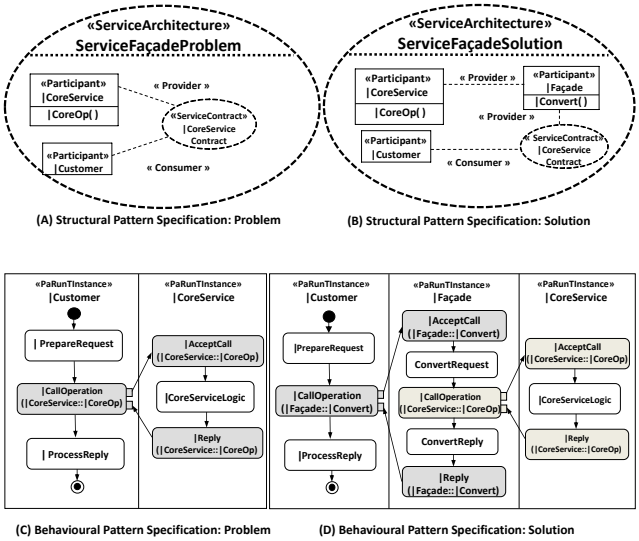


Figure 3: Service Façade pattern specification

V. SMODEL TRANSFORMATION RULES

The first step in applying a pattern is to identify the model elements to which it can be applied, based on the pattern problem. From these, particular elements are chosen as the *area of application* by binding them to roles in the RBM definition. It is not our goal to automate this process of selection and binding and then applying the pattern solution, but to make it systematic and to support it with a construction tool (as shown below in Figure 4).

A. Problem Identification and Role Binding

The designer chooses a pattern to apply and, using its RBM definition from a pattern library, binds the elements of its problem specification (SPS, BPS and DPS) to the elements of the SModel. An element can be bound if:

1. Its type matches the RBM element type.
2. It has all the attributes and operations defined by the RBM element.
3. Any constraints defined for the two matching elements are compatible (that is, the pattern does not impose additional constraints when applied to the SModel).
4. For the SModel behavioral view (BPM), the execution flow and the ActivityPartitions (swimlanes) must match.

Not every pattern specification element is defined as a role. Those which are not (e.g. Calls, Replies, Attributes) are also bound, governed by the role bindings. These "derived bindings" may be determined by the binding of a single element (e.g. its Attributes) or from the bindings of multiple elements (e.g. a Connector between two elements). Some bindings for the BPM of the example (involving the pattern specification in Figure 3 and the SModel BPM in Figure 2.B) are given by the following pairs including a derived binding found between the RBM Call and SModel Call, which is implied by the binding of the core operation:

RBM Element	SModel Element
|CallOperation(|CoreService::|CoreOp)	CallOperationAction(Shopping::Checkout)
|AcceptCall(|CoreService::|CoreOp)	AcceptCallAction(Shopping::Checkout)
|CoreServiceLogic	Sequence of all Actions in Shopping swimlane
(Derived Binding) Call **from** |CallOperation(|CoreService::|CoreOp) **to** |AcceptCall(|CoreService::|CoreOp)	Call **from** CallOperationAction (Shopping::Checkout) **to** | AcceptCallAction (Shopping::Checkout)

B. Creating the SModel Transformation Rules

The designer creates the SModel transformation, (governed by the RBM bindings and the pattern problem and solution specifications) as a set of operations to add, delete, and modify model elements. An operation is defined for each element type (eg. addAssoc/deleteAssoc for adding/deleting associations). Depending on the element type it applies to, an operation is applied to the services and interactions of the SEAM and to the ActivityPartitions, Activities, Actions and ActivityEdges of the BPM. Transformation operations indicated by the designer are recorded using the tool shown in Figure 4 as follows:

- Remove elements that are present in the problem but not in the solution, by applying *delete* actions (such as *deleteParticipant* or *deleteAssoc*) to them
- Create new elements that are defined in the solution but are not present in the problem, by *add* actions (such as *addParticipant* or *addActivityPartition*),
- Modify elements present in both problem and solution, by *modify* actions (such as *modifyActionCall).*

SModel elements which are not in any of the above groups remained untouched. Figure 4 shows a screen shot of the tool support for the technique in this section with a set of operations recorded for the application of Service Façade pattern to the example in Figure 2, with the role bindings shown above.

VI. COUPLED TRANSFORMATION

A. Coupled PModel Refactoring Rules

This section describes the automated translation of the SModel transformation rules into PModel transformation rules, based on the mapping table described in Section IV.C. Each SModel transformation rule has an operation name and some arguments, which are processed as follows:

1. The operation name is translated into one or more PModel transformation operations. The action part of the name (add/delete/modify) is retained, and the operand-type part (e.g. Participant) is mapped according to the type correspondences of the Mapping Table. A partial list of these is:

SModel Type	PModel Type
Participant	Task
ActivitySet	Entry
Call/Reply pair of Actions	Call (sync)
Call (no Reply)	Call (async)
ExecHost	Host

Thus the SModel operation addParticipant is translated to addTask, and deleteActivitySet to deleteEntry.

2. The arguments of the PModel operation (e.g. the entity or entities to be added, deleted, or modified) are translated from the arguments of the SModel operation using the correspondences in the Mapping Table. For "add" operations the name of the new PModel element is taken as the name of the corresponding SModel element.

For example, the SModel "addParticipant" operation is mapped to "addTask" in the PModel, and the "addParticipant" argument becomes the new task name.

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

Some of the PModel transformation rules derived for the Façade pattern (shown in Figure 4) are presented in Figure 5 as part of the screenshot from the implemented tool supporting coupled transformations.

B. *Application of the PModel Rules*

Briefly, the PModel transformation rules derived in Section VI.A are applied to the PModel in two steps. First the PModel is annotated with transformation directives indicating the changes, then the changes are applied by a transformation engine implemented using QVT [13] (Query, View, and Transformation, a OMG standard model transformation language) which processes the directives. The implementation of these two steps is not presented here.

VII. CASE STUDY

We suppose that a designer is assigned the task of re-designing the Shopping and Browsing SOA described earlier to support three different user access channels (mobile phone, desktop, kiosk, etc.) through a single multi-channel endpoint. Initially, the designer uses the SOA design pattern "Concurrent Contracts" [1] in which the multi-channel capability is implemented by providing separate shopping and browsing operations for each channel. Separate set of actions are created inside the shopping swimlane (see Figure 2.B) and also the browsing swimlane (not shown Figure 2.B). However, the designer realizes that those three separate operations introduce code duplication in the functional design.

To eliminate this duplication the designer considers using the SOA design pattern "Service Façade" [1]. In the service façade design pattern, the problem is that the tight coupling of the core service logic to its contracts can obstruct its evolution and negatively impact service consumers. As the *solution*, Façade logic is inserted into the service architecture to establish a layer of abstraction that can adapt to future changes to the service contract.

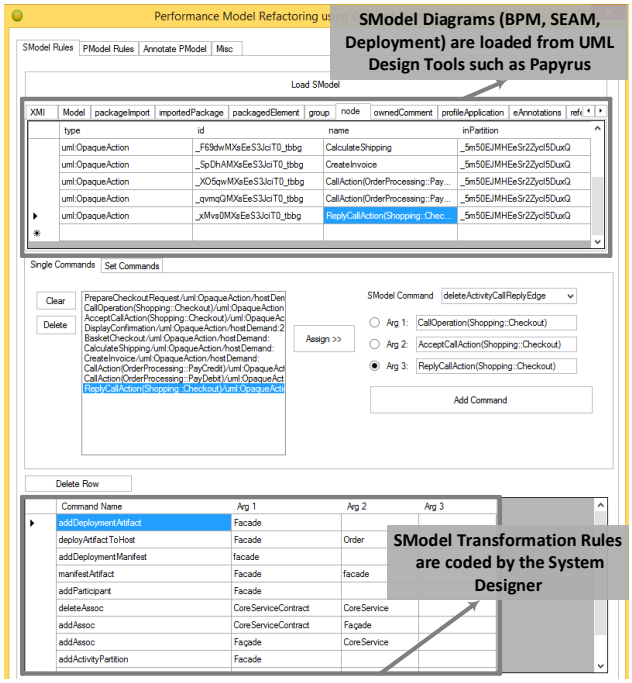


Figure 4 : Tool for Recording SModel Transformation Rules

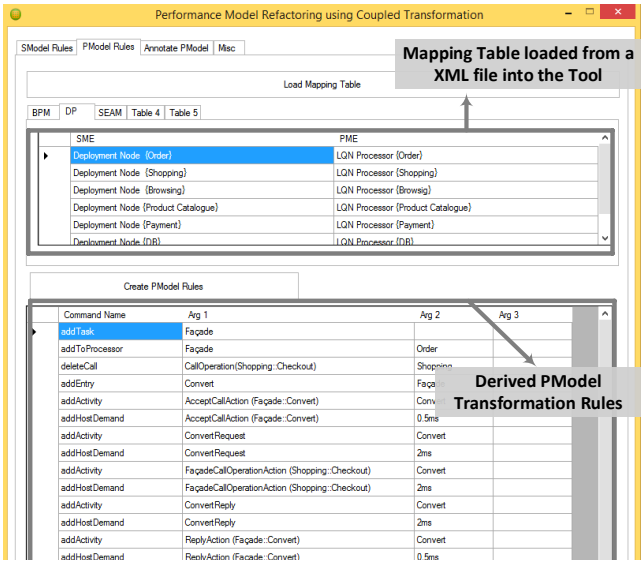


Figure 5 : Tool for automatic derivation of PModel Rules

Concerned that the façade overhead might impair the system performance, the designer applies the present technique. The designer first binds the pattern roles and records the necessary SModel changes (as in Section V, and the screenshot of Figure 4), using the base SOA design loaded from a standard UML modeling tool (e.g. Papyrus). The recorded rules and the Mapping Table are used by the

coupled transformation tool (as in Section VI and the screenshot of Figure 5) to derive the PModel transformation rules. The PModel transformation rules are applied to the LQN model shown in Figure 2.D, giving a performance model which is partly shown in Figure 6 below.

To illustrate how performance issues can be revealed, the performance was estimated for a range of user populations. For each N users in group "users1", there were 2N in "users2", and N/2 in "users3". N ranged from 2 to 220, so the total users ranged from 7 to 770. Figure 7 shows the response times for the three groups of users and for both patterns. It shows that the groups have the same response time, and under heavy loads (which are also the conditions in which the system resources are efficiently utilized) the Façade pattern imposes about 30% additional delay in response time. This penalty is the price for the benefits it provides to the system architecture by preparing it for future changes to the service. An alternative view of the penalty is that it reduces the user population that a deployed system can serve with a given target response time.

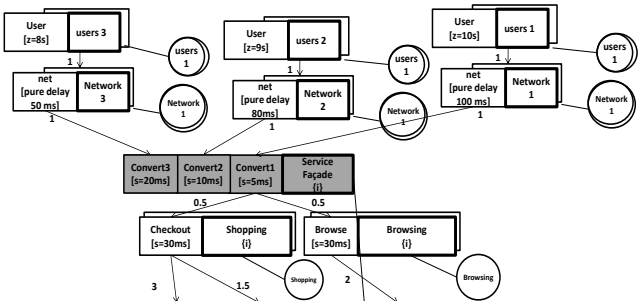


Figure 6: Partial Refactored PModel (Façade applied)

VIII. Conclusion

This paper describes a process and tools for interpreting a software pattern in terms of the corresponding change in a performance model of the software, to support an immediate analysis of the performance effects of using a pattern. It helps the system designer to choose a pattern that has acceptable performance impact, and to choose between alternatives. It provides the system designer with a systematic approach and tool for formally recording those changes for the SOA design and from these it automatically derives the performance model changes. Coupling the transformations ensures that the performance analysis remains in sync with the software changes, and relates the resource and performance changes back to the pattern.

The use of the process and tools was illustrated by an extensive example which applied the Facade pattern to a Browsing and Shopping system design, and by an analysis which compared its impact to that of the Concurrent Contracts pattern. The performance cost of the Facade pattern is a significant increase in response time under load, which could influence the development of the design.

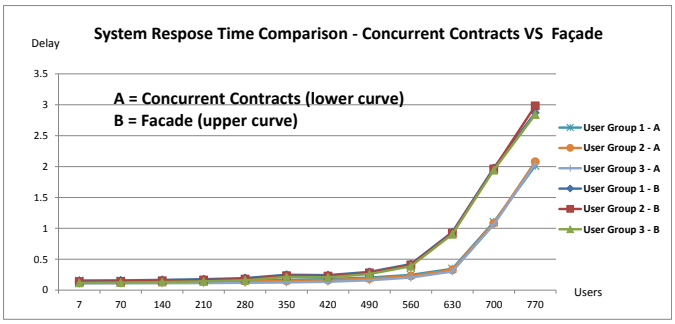


Figure 7: System Response Time (ms) for (A) Concurrent Contracts and (B) Façade patterns

Acknowledgment

This work was supported by the Ontario Centers of Excellence and by the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Discovery Grant program.

References

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

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

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

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

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

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

[7] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks," *IEEE Trans. on Software Eng.*, vol. 35, pp. 148-161, 2009.

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

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

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

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

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

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