Towards Studying the Performance Effects of Design Patterns for Service Oriented Architecture

Nariman Mani, Dorina C. Petriu, Murray Woodside

Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, Ontario, Canada

{nmani | petriu | cmw}@sce.carleton.ca

ABSTRACT

Patterns employed for the development of a service oriented system may affect its non-functional properties, including performance. Service Oriented Architecture (SOA) design patterns provide generic solutions for many architectural, design and implementation problems, and any pattern may have an impact on performance, either positive or negative. This research considers how to characterize the performance impact of a SOA design pattern, which includes characterizing some aspects of the design and usage environment as a whole (for example, the scale of the workload and the availability of concurrent platforms for the eventual deployment). The approach uses performance models to characterize the application and the impact of the pattern on it. The planned approach exploits the context of model driven engineering (MDE) to give rapid feedback to developers about the potential impact of a pattern. Model transformations are used to generate the performance model, and to propagate the effect of applying a SOA design pattern to the performance model. The approach is sketched here with a preliminary case study, demonstrating its feasibility.

Categories and Subject Descriptors

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

General Terms

Performance, Design, Experimentation, and Verification.

Keywords

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

1. INTRODUCTION

Service Oriented Architecture (SOA) provides many architectural benefits to the design of a distributed system including reusability, adaptability, and maintainability. A service is a coarse-grained piece of logic providing a distinct business function, which autonomously implements the functionality promised by the contracts it exposes [1, 2].

SOA raises various challenges. The first set of challenges is related to non functional properties of distributed systems such as availability, security, scalability, performance, etc. The second set of challenges is related to issues around the architectural design of service oriented systems, such as providing service aggregation and a centralized view in an environment which promotes autonomy, encapsulation, and privacy.

SOA design patterns as collected for example in [1] help to address these challenges. Each pattern is specified by:

- ***Problem:*** describe the domain of problems that pattern aims to solve and their impacts.
- ***Solution:*** describes the design solution proposed by the pattern to solve the problem.
- ***Application Instruction:*** provides generic guidance on how to change the design in order to apply the pattern.

This work considers the performance impact of patterns. For patterns which address a performance problem, we try to characterize the amount of improvement, and what it depends on in the pattern and in the larger application. For other patterns (which may not addressing performance aspects of a design indirectly), we try to characterize their performance impact, for instance the throughput impact of overhead introduced by the pattern.

Using MDE, performance impacts can be evaluated through performance models [3]. A software design model (SModel) can be transformed by known techniques (which require some additional information) to a performance model (PModel) and evaluated using the existing techniques and methodologies in [3]. This work uses UML-based SModels and Layered Queuing Network (LQN) PModels created by transformations in the PUMA framework [3] (see examples in [3]).

It is essential to explore the use of different patterns, to find those that are effective and avoid those that introduce new problems. To streamline this process, this paper considers propagating incremental SModel changes due to the application of a SOA design pattern, from the SModel to the corresponding PModel of the system. Using the application instructions provided by the SOA design patterns, a set of application rules are extracted from design patterns. The application rules are used to identify the changes in SModel and the associated changes in the PModel. On the other hand, the performance model may help indentifying performance problems which will guide the developers to select appropriate performance-enhancing SOA patterns.

The remainder of this paper is structured as follows. The related work is discussed in Section 1. An illustrative example is discussed in Section 1. The proposed approach (overview) and examples of its applications are discussed in Section 4. Finally we conclude the paper in Section 5.

2. RELATED WORK

Beside the patterns which are recognized as the best practices for software development, Smith and Williams [4] introduced general performance anti-patterns that exclusively focus on performance concerns. Anti-patterns are defined as common design mistakes that consistently occur, causing undesirable results.

There are only a few works on studying the impact of the patterns/anti-patterns on the performance of software applications. Cortellessa et al [5] presented an approach, based on anti-patterns, that aims at identifying performance problems based on OCL rules in UML models and removing them. Also in their approach, the identification of an anti-pattern suggests the architectural alternatives that can remove that specific problem.

Menascé et al [6] presents a framework called SASSY whose goal is to allow designers to specify the system requirements using a visual activity-based language and to automatically generate a base architecture that corresponds to the requirements. The architecture is optimized with respect to quality of service requirements (i.e. as measured by several performance metrics such as execution time and throughput, etc.) through the selection of the most suitable service providers and application of quality of service architectural patterns.

Parsons and Murphy [7] introduce an approach for the automatic detection of performance anti-patterns by extracting the run-time system design from data collected during monitoring by applying a number of advanced analysis techniques.

Xu [8] applied rules to performance model results to identify performance problems and to propose solutions for fixing them, which resulted in changes at the PModel level.

In this work, we examine the impact of changes made by a SOA design pattern on the SOA design model (SModel) and identify the associated performance model (PModel) elements which are affected by this change. In this paper we illustrate the proposed approach by performing the necessary steps "by hand"; future work will attempt to automate the process. To the best of our knowledge, there have not been any works on studying the incremental change propagation due to the application of design patterns from a SModel to a corresponding PModel.

3. ILLUSTRATIVE EXAMPLE

In this section of paper, we present an illustrative example of a service oriented system in Section 3.1. Using the presented example, the performance characterizations of the design models are discussed in Section 3.2.

3.1 Design Model Scenarios

Figure 1 and Figure 2 show UML Sequence Diagrams (SD) scenarios for product catalogue browsing and shopping services. Users browse the catalogue and create their own shopping cart using the "Browsing Service". Then the user asks for checkout by using "Shopping Service" to place the order and pay for it. Figure 1 shows the sequence diagram for the browsing scenario in detail. First, the user sends the request for browsing a specific product catalogue based on filtration criteria to "Browsing Service". The "Browsing Service" sets up the user session and passes its request to the "Catalogue Service" which is responsible retrieving the products list from the back-end database and formats them into a page. The "Catalogue Service" sends the request for retrieving the products information to back-end database. The "Catalogue Service" formats the retrieved data into a product catalogue and sends it to the user for browsing and making the shopping cart.

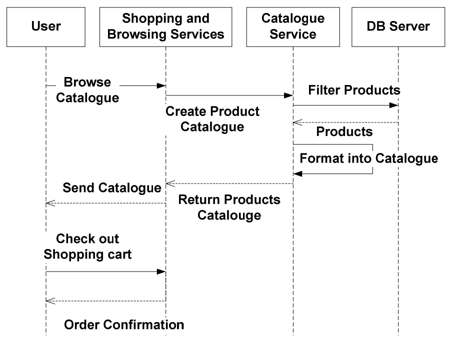

Figure 1. Browsing Service sequence diagram

Figure 2 shows the message sequence for the shopping scenario. Once done with browsing the catalogue and creating the shopping cart, the user asks for check out from the "Shopping Service". The "Shopping Service" collects the shopping information (e.g. user, shipping, and payment information) and sends them to "order processing service". The "order processing service" validates the credit card information user provided send the payment information to "Payment Processing". It receives the payment confirmation and informs the "Shopping Service" that the order has been placed. "Shopping Service" sends the confirmation to user.

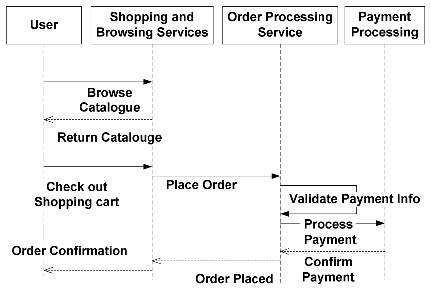

Figure 2 . Shopping Service sequence diagram

Figure 1 and Figure 2 are a simple version of the SModel for the service system, which must be augmented by CPU demand data and a deployment specification which is omitted here for space reasons.

3.2 Performance Model of the Design

The PModel image of the SModel scenario shown in Figure 1 is presented in form of the Layered Queuing Network (LQN) [4] in Figure 3. Each large rectangle represents a LQN *task* (roughly, a process), named in a sub-rectangle at its right-hand end. Other sub-rectangles represent *entries* (service functions) with their host CPU demand. Requests to other entries are shown by arrows labeled by the number of calls, per entry invocation. Tasks correspond to lifelines in the SDs and service functions correspond to messages between objects. Circles represent processors, attached to their deployed tasks; as a "default" deployment one processor is shown per task.

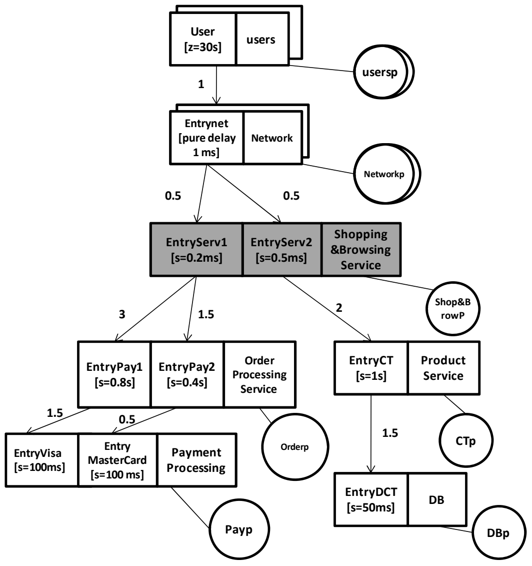

Figure 3. LQN model for Shopping and Browsing Service Oriented Application

PModel parameters such as the number of invocations of entries and the mean CPU demands in each entry come from data provided through performance annotations in the SModel. There are 3 invocations of EntryPay1 and 1.5 invocations of EntryPay2 per request from shopping service (i.e. EntryServ1). Upon each invocation, on average, EntryPay1 uses the EntryVisa 1.5 times, whereas, on average, EntryPay2 uses the EntryMasterCard 0.5 times in average. For every invocation of the browsing service (i.e. EntryServ2) the product service is invoked 2 times on average. For every invocation of the product service, the EntryDB 2 is used 1.5 times on average. Results for solving the Figure 3 LQN model (Using LQN solver called LQNS[4]) with 50 users are shown in Table 1. They show that the "Shopping and Browsing Service task" is saturated, with utilization parameter close to 0.99.

Table 1. Performance Analysis Results for 50 users

	Throughput (req/sec)	**Utilization**	**Response Time (sec)**
Entire System	0.35408	39.3776	141.211 s
Shopping & Browsing	0.35408	0.989389	Shopping: 3.43801s
			Browsing: 2.1505s

4. PERFORMANCE EFFECT OF A DESIGN PATTERN

The impact of introducing a design pattern, including a SOA pattern, comes from its effect on system attributes **D**, **R** and **S**:

- **D**: CPU demands
- **R**: available resources
- **S**: execution sequence, including order, parallelism, and numbers of calls to service functions.

The pattern may modify existing objects in the design, or introduce new ones. A direct approach to evaluating a pattern would be to apply it to the design, find the new PModel image, evaluate it, and compare.

Here we seek a more efficient process by considering how the pattern application rules (in the SModel space) imply corresponding changes in the PModel space. These changes can be termed the *pattern image* in the PModel space, and the rules for introducing them. The pattern image will take the form of new objects, and PModel parameters representing **D, R** and **S** above.

Figure 4 provides a high-level overview of the approach that we propose in this paper. The top of the figure (not included in the grey box) shows the direct approach applied to the initial SModel. The PModel is solved with existing solvers and the analysis results are produced, as in the example above.

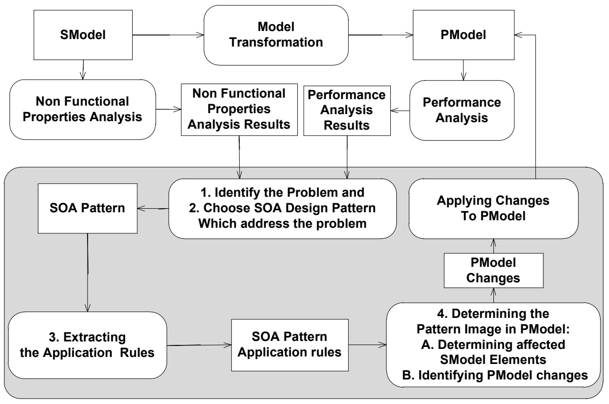
SModel
Model Transformation
PModel
Non Functional Properties Analysis
Non Functional Properties Analysis Results
Performance Analysis Results
Performance Analysis
SOA Pattern
1. Identify the Problem and
2. Choose SOA Design Pattern Which address the problem
Applying Changes To PModel
PModel Changes
3. Extracting the Application Rules
SOA Pattern Application rules
4. Determining the Pattern Image in PModel:
A. Determining affected SModel Elements
B. Identifying PModel changes

Figure 4. Technique Overview

The operations in the grey box in Figure 4 describe our main focus in this research.

1. We begin with a Problem in the SModel that needs to be fixed by applying SOA patterns. This could be either a performance problem (to be identified from the Performance Analysis Results) or other kind of problem (to be identified from other non-functional requirement analysis).
2. Depending on the type of problem, an appropriate design pattern is chosen (this is conventional application of patterns to design).
3. The application rules of the SOA design pattern are extracted. As mentioned in the introduction section, each pattern provides a list of generic application rules which specify the changes that should be made to the design model in order to address the problem.
4. For the chosen pattern and its application rules, the Pattern Image in PModel space is determined:
    a) Using the application rules of the pattern, the affected elements in the design model (affected SElements) and the SModel changes are determined,
    b) From these, the affected PModel elements and the PModel changes are determined, based on the affected SElements.
5. The identified changes are applied to give the transformed PModel and its performance analysis.

In this paper we apply two specific SOA patterns, "Functional Decomposition" (Sections 4.1) and "Asynchronous Queuing" and (Section 4.2) to the case study system designed for this research (scenarios shown in Figure 1 and Figure 2), online shopping system. These two patterns are briefly presented here. A more detailed discussion on patterns is in [1].

4.1 The Performance Effects of Functional Decomposition Pattern

In general, functional decomposition pattern [1] discusses how to design a service solution for a large business problem without having to build a standalone body of solution logic.

Problem: To serve a large and complex business task, a corresponding amount of solution logic (service) needs to be created, resulting in a self-contained application with traditional governance and reusability constraints.

Solution: The large business task should be broken down into a set of smaller, related tasks, leading to a corresponding set of smaller, related services which satisfy those tasks.

Application Instruction: The large services in SOA which carry a very high load should be identified and be broken down into a set of smallest services, each satisfying a functionality of the large service. The affected element here is the "large" service. This may be large in terms of complexity, but also for performance purposes it may be large in the sense of heavily utilized, so that its thread resources are saturated.

The Pattern Image in PModel space is to break down the task implementing the large service, into one task for each service function (or, break it down as much as feasible).

Pattern Image Application Rule:

Condition: For the selected "large" service with multiple service functions,

Actions: Split the task associated to service into smaller tasks, one for each entry of the large task. The entries of the smaller tasks have the same properties as they had before.

The "Functional Decomposition" design pattern is now applied to the service oriented design model described in Section 3. In Table 1 it can be seen that the Shopping and Browsing Service is a "large" task in the sense of high utilization (The task shown by grey color in Figure 3). The SElement which is affected by this pattern would be Shopping and Browsing Service use case in the sequence diagram (See Figure 1 and Figure 2). Figure 5 shows the Shopping Service sequence diagram after applying the pattern to the SModel.

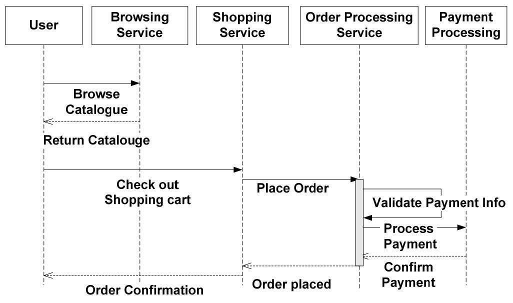

Figure 5. Shopping Service sequence diagram after applying Function Decomposition Design Pattern

The image PElement is the Shopping and Browsing task in the PModel, which will be partitioned. The PModel change can be described by these application rules:

a) Add a new task
b) Move the second entry with all its connections to the new task.
c) Name the new task for its entry name
d) Name the original task for its remaining entry name

Figure 6 shows the changed LQN model, and Table 2 shows the performance effect found by solving the PModel. Table 2 shows considerable improvements in the throughput, utilization, and response time of the whole system. The effect of the change is strong because the Shopping and Browsing Task was in fact the bottleneck in the performance model.

Table 2. Performance Analysis Results Summary for 50 users after applying Function Decomposition Design Pattern

	Throughput (req/sec)	Utilization	Response Time (sec)
Entire System	0.555463	33.1209	90.015s
Shopping and Browsing	0.277731	0.99173	Shopping: 3.57083s
	0.277731	0.59726	Browsing: 2.150s

In general: the component to be decomposed points directly to the corresponding decomposition in the PModel.

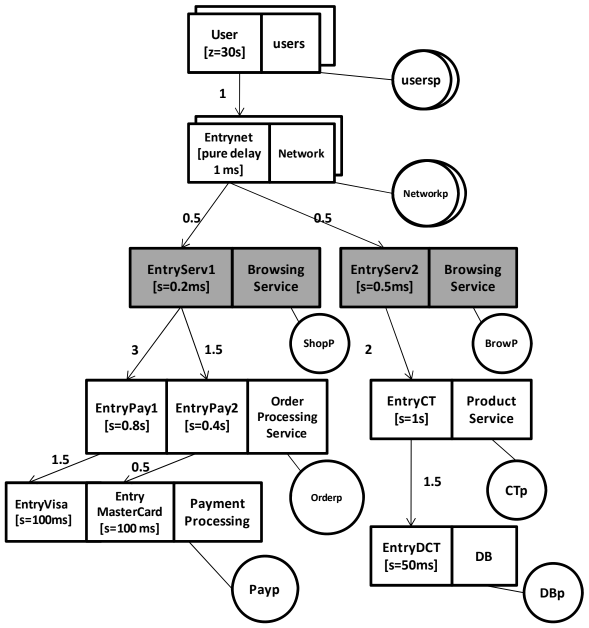

Figure 6. The LQN Model after Applying the Functional Decomposition Pattern

4.2 The Performance Effects of Asynchronous Queuing Pattern

A different kind of pattern provides a second look at the process. The Asynchronous Queuing [1] pattern introduces capabilities allowing a service and its consumers to accommodate failures independently and avoid unnecessarily blocking resources.

Problem: When the service functionality requires that consumers interact with it synchronously, the performance, reliability, or availability can be affected.

Solution: The requestor can be provided with an intermediate response confirming that the request will be taken care of latter. While keeping the user informed about the task status, the request will be processed without the requestor waiting and blocked. The requester must pick up the response when it is ready.

Application Instruction: The task portion that can be processed without blocking the requestor should be identified and postponed to after an intermediate response message to the requestor.

Application Rule

For the SModel**:**

Condition: If there is any task with a long processing time in the system

Actions: Provide the requestor with an intermediate response and postpone the rest of processing after sending the intermediate response.

This is a kind of delayed synchronous interaction.

The Order Processing Service is a good candidate for applying this pattern. Its requestor is the Shopping Service. It has a long processing time because it waits for the functions for Visa and for MasterCard. They use the Payment Processing Service which in both cases has a long service time (100 ms) compared to other processes in the system.

To apply the pattern we divide the Order Processing functions into a part which is synchronous with the shopping service, and an asynchronous part making the final call to payment processing. The final confirmation is returned asynchronously to the shopping service and the user. The sequence of execution in Figure 5 must be modified as indicated in Figure 7. As it can be seen in Figure 7, the reply from "Payment Processing" is sent back directly to the Shopping Service asynchronously in the form of "Confirm Payment" message.

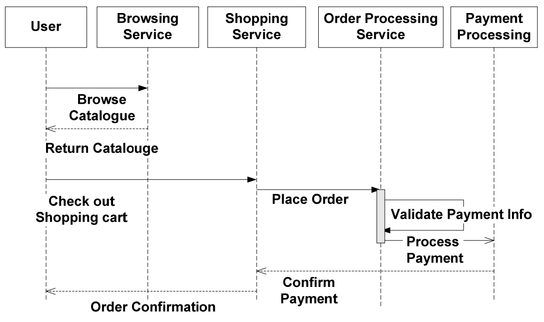

Figure 7. Shopping Service sequence diagram after applying Asynchronous Queuing Design Pattern

For the PModel: LQN can describe delayed synchronous interactions by a "second phase of service" (see [4]). Any entry may have some of its work in second phase, asynchronous with its caller. A final interaction to return data is a separate message.

In applying the pattern, the associated PModel elements for PlaceOrder are EntryPay1 and EntryPay2 (for orders to Visa and MasterCard) in the "Order Processing Service" task. Their CPU demand (just for illustration) be assumed equally divided between the first phase to validate the payment information, and the second phase to handle the processing request. The calls for payment processing are made in the second phase.

The last asynchronous interaction with the Payment server is approximated in LQN by a forwarding interaction. The pattern is: an asynchronous request from Order processing to payment processing, leading to an asynchronous message to the Shopping Service with the result. Where there is more than one request to Payment processing (as with 1.5 requests to VISA processing, on average), the additional requests (on average 0.5 requests) are modeled as synchronous, with the last request being forwarded (See Figure 8). Figure 8 shows a synchronous request to EntryVISA with parameter 0.5 and a forwarding request (dashed arrow) with probability 1.0

The resulting PModel in gives the performance analysis results in Table 3. The response time is only marginally reduced by using this pattern. This is explained also by noting that the critical path for completing a Check Out request is unchanged in Figure 7; if we look deeper we find that concurrency limitations in the design prevent this pattern from being effective.

In general the corresponding PModel elements for this pattern are easily identified. The SModel increment is resolved in the behavior specification, and transferred via demand and call parameters into the PModel.

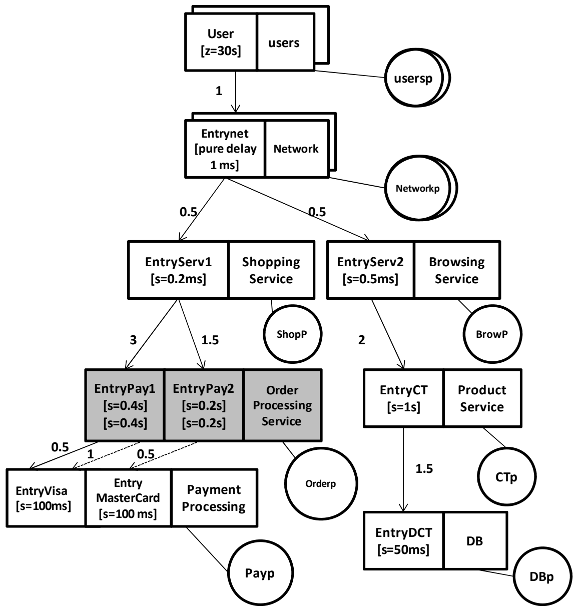

Figure 8. The LQN Model after Applying Asynchronous Queuing Design Pattern

Table 3. Performance Analysis Results for 50 users, for the LQN model with the "Asynchronous Queuing" pattern

	Throughput	Utilization	Response Time
Entire System	0.610423	31.687	81.9105s
Shopping and Browsing	0.305212	0.996745	Shopping: 3.26575
	0.305212	0.656358	Browsing: : 2.1505s

4.3 Result Analysis

The system throughput and response time comparison for the initial model and two applied design patterns are shown in Figure 9 and Figure 10. The graphs show the system throughput and response time under an increasing load (i.e., the numbers of users is changing from 1 to 120 users). As it can be interpreted from the graphs, there is large difference in the system throughput and system response times between the initial state and after the patterns are applied. However, the first pattern has a considerable effect, while the second pattern (Asynchronous Queuing) makes a small additional improvement. The designers may conclude that the application of the second pattern is not warranted. On the other hand, additional analysis may show that using other patterns first (for example to increase concurrency) may make Asynchronous Queuing more effective.

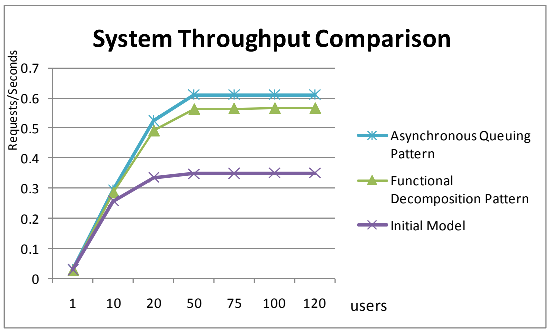

Figure 9. System Throughput Comparison

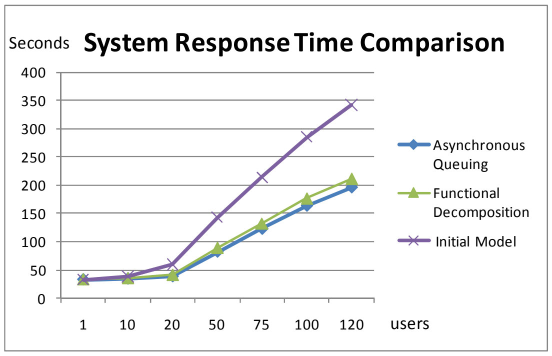

Figure 10. Response Time Comparison

5. CONCLUSION

In this paper, we proposed an approach which propagates changes due to the application of design patterns from the design model of SOA system (SModel) to the associated performance model (PModel) in form of layered queuing network. The proposed approach extracts a set of application rules from the SOA patterns application instructions and uses them for determining the changes to SModel elements, which are then propagated to the associated PModel structural and behavioral elements.

These are preliminary experiments in research which will describe the possible pattern images for the PModel space for different SOA patterns, and to determine the pattern image application rules for the PModel. The examples have demonstrated feasibility and have shown the nature of the pattern image for two patterns, applied in certain cases. These are by no means definitive and a general approach is still to be developed. However the examples show that some pattern applications are effective and some are not. An ideal future system will partly automate the change propagation into the PModel for well-understood patterns, and will screen automatically for improvements.

6. Acknowledgements

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

REFERENCES

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

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

[3] M.Woodside, D. C. Petriu, D.B. Petriu. H. Shen, T. Israr, J. Merseguer, "Performance by Unified Model Analysis (PUMA)", *Proc. ACM Workshop on Software and Performance WOSP'05,* Palma, Illes Balears, Spain , 2005, pp. 1-12

[4] C. U. Smith and L.G. Williams, *Performance Solutions*. Addison Wesley, 2002.

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

[6] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex , J. P. Sousa , "A framework for utility-based service oriented design in SASSY", *Proc 1st WOSP/SIPEW Int Conf on Performance Engineering*, San Jose, CA, pp. 27-36 , 2010.

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

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