Model-Driven Performance Analysis Methodology for Distributed Software Systems

Swapna S. Gokhale
Paul Vandal
Dept. of CSE
Univ. of Connecticut
Storrs, CT
{ssg@cse.uconn.edu}

Aniruddha Gokhale
Dimple Kaul, Arundhati Kogekar
Dept. of EECS
Vanderbilt Univ.
Nashville, TN
{a.gokhale@vanderbilt.edu}

Jeffrey Gray
Yuehua Lin
Dept. of CIS
Univ. of Alabama, Birmingham
Birmingham, AL
{gray@cis.uab.edu}

***Abstract*— A key enabler of the recently popularized, assembly-centric development approach for distributed real-time software systems is QoS-enabled middleware, which provides reusable building blocks in the form of design patterns that codify solutions to commonly recurring problems. These patterns can be customized by choosing an appropriate set of configuration parameters. The configuration options of the patterns exert a strong influence on system performance, which particularly for real-time systems is of paramount importance. Despite this significant influence, currently there is a lack of significant research to analyze performance of middleware at design time, where performance bottlenecks can be resolved at a much earlier stage of the application lifecycle and with substantially less costs.**

I. Introduction

Society today is increasingly reliant on the services provided by distributed real-time software systems. These services have permeated our lives and have become prevalent in many domains including health care, finance, telecommunications and avionics. In many of these domains, the performance of a service is just as important as the functionality provided by the service.

To counter the dual pressures of developing systems which offer a rich menu of services with superior performance, while simultaneously reducing their time-to-market, service providers are increasingly favoring the assembly-centric approach to software over the traditional development-centric approach. A key facilitator of this assembly-centric approach has been *QoS-enabled middleware* [?]. Middleware consists of software layers that provide platform-independent execution semantics and reusable services that coordinate how system components are composed and interoperate. Middleware offers a large number of reusable building blocks in the form of design patterns [?], [?], which codify solutions to commonly recurring problems. These patterns can be customized with an appropriate set of configuration parameters as per system requirements.

The choice of configuration parameters have a profound influence on the performance of a pattern and hence a system implemented using the pattern. Despite the influence on system performance, which is crucial for real-time systems, current methods of selecting the patterns and their configuration options are manual, *ad-hoc* and hence error-prone. The problem is further compounded, because there are no techniques available to analyze the impact of different configuration parameters on the performance of a pattern prior to building a system. Performance analysis is thus invariably conducted after a system is assembled, and it is often too late and too expensive to take corrective action if a particular selection of patterns and their configuration parameters cannot satisfy the desired performance expectations. The capability to conduct design-time performance analysis of middleware patterns and the composition of these patterns is thus necessary, especially for systems with stringent performance requirements.

This project seeks to develop an analysis methodology for design-time performance analysis of a system implemented using middleware patterns. The methodology comprises of two steps. The first step consists of building and solving performance models of individual middleware patterns. In the second step, strategies to compose the performance models of individual models mirroring the composition of patterns and methods to solve the composite model to estimate system performance are developed. Our goal is to automate these processes via model driven engineering (MDE) [?] where the systems developer is provided artifacts that are intuitive and closer to their domain to compose the systems from building blocks. Generative tools [?] supported by the MDE approach can then automate the synthesis of performance analysis metadata that is subsequently used by backend analysis tools.

The illustration of the first step of the methodology on Reactor, Proactor and Active Object patterns and of the second step on the composition of Reactor and Active object patterns demonstrates the feasibility of conducting system performance analysis at design time using the model-driven paradigm.

The rest of the paper is organized as follows: Section II;

II. Performance analysis of a pattern

In this section we discuss the process envisioned for the performance analysis of an individual pattern. We describe the various steps involved in the process and how they support each other. We illustrate the performance analysis methodology on a producer/consumer system built using the Active Object pattern.

A brief description of the steps involved in the performance analysis process shown in Figure 1 are as follows:

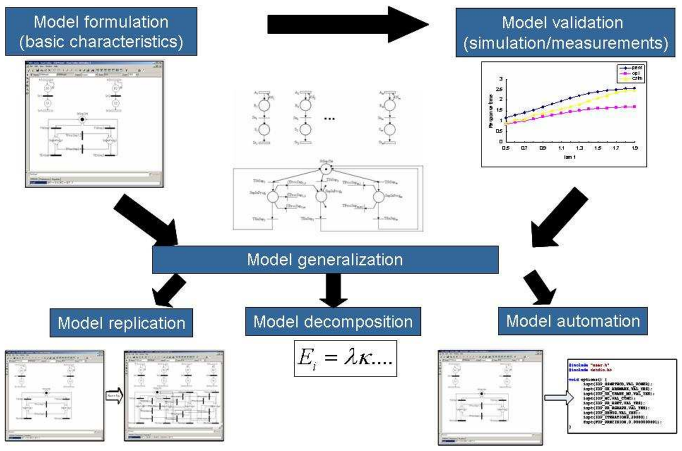

Model formulation (basic characteristics)
Model validation (simulation/measurements)
Model generalization
Model replication
Model decomposition
$E_i = \lambda\kappa....$
Model automation

Fig. 1: Performance analysis process of a middleware pattern

- **Model formulation:** The model formulation step comprises of capturing the basic or the invariant characteristics of the pattern into a performance model. Different modeling paradigms such as the Stochastic Reward Nets (SRNs) [?], Layered Queuing Networks [?], Colored Petri Nets [?] may be used for this purpose. The performance model can then be solved/simulated using tools such as the Stochastic Petri Net Package (SPNP) [?], DesignCPN [?].
- **Model validation:** The performance estimates obtained from the solution of the performance model need to be validated using simulation or experimentation. The experimentation can be conducted by implementing a system with the pattern on the ACE framework `www.dre.vanderbilt.edu/ACE`. Simulation can be built using a general-purpose simulation language such as CSIM [?].
- **Model generalization:**
- **Model decomposition:** It is expected that it would be infeasible to solve the performance model of a practical system without encountering the state space explosion issue. The model decomposition step thus involves developing a strategy to partition the model into sub-models. The sub-models can then be solved separately and their results can be combined to obtain performance estimates.
- **Model automation:** Manually creating the analytical models for performance analysis of patterns-based middleware building blocks becomes infeasible as the complexity of the building block increases. Our two modeling languages POSAML (Patterns-oriented Software Architecture Modeling Language) [?] and SRNML (Stochastic Reward Net Modeling Language) provide the users with intuitive higher level abstractions to model the patterns and their behavior that is useful for analysis. Generative tools associated with these modeling languages then automate the generation of the metadata that can be tailored to provide simulation, analysis or empirical benchmarking [?], [?].
- **Model replication:** In MDE, it is often desirable to evaluate different design alternatives as they relate to scalability issues of the modeled system. A typical approach to address scalability is to create a base model that captures the key elements and their relationships. A collection of base models can be adorned with necessary information to characterize a specific scalability concern as it relates to how the base modeling elements are replicated and connected together. In current modeling practice, replication is usually accomplished by scaling the base model manually. This is a time-consuming process that represents a source of error, especially when there are deep interactions between model components. As an alternative to the manual process, our research [?] has focused on the idea of automated model replication through a model transformation process that expands the number of elements from the base model and makes the correct connections among the generated modeling elements. We have leveraged and expanded the capabilities of the C-SAW (Constraints-Specification Aspect Weaver) [?] tool for this purpose.

The performance analysis process has been illustrated on the Reactor pattern [?], [?], [?], [?], Proactor pattern [?], [?] and Active Object patterns.

III. Performance analysis of an Active Object-based System

We illustrate the performance analysis process using the Active Object pattern in the following subsection.

A. Description of the pattern

In a multi-threaded application, it is common for several threads to require the utilization of the same resource. These threads then compete for mutually exclusive access to the resource and utilize it for the total time of the required operation. For low request rates and short session durations, the performance of this architecture may be acceptable. On the other hand, for high request rates and long access times, the performance degradation may be significant. The Active

Object pattern can be used to alleviate the performance problems encountered in this type of system. This pattern provides concurrency and simplifies synchronization to the shared resource. This is achieved by decoupling method invocation from method execution and creating the shared resource in its own thread of control.

The Active Object [?], shown on the left in Figure 2,[1] is composed of the following group of components: Proxy, Activation List, Scheduler, Servant, and Method Requests. The interactions between the components is shown on the right in Figure 2. This interaction is initiated by a client thread invoking a method on the proxy to the Active Object. This proxy is implemented using the Extension Interface pattern [?]. It lies in the client thread and provides an interface to the publicly available methods on the shared resource. Instead of immediately executing the method upon invocation by the client thread, the proxy constructs a Method Request and enqueues it on the Activation List of the Active Object. Thus, from the client thread's perspective, the method has been executed.

The Method Request is a structure that carries the parameters of the method invoked, along with other information necessary to execute the method request later in the process, including bindings to the method it represents and synchronization information. It also has guards or synchronization constraints to prevent the method from being executed when certain requirements have not been met. For example, if the Method Request is a put call on a message queue and the queue is full, the method request will not meet its synchronization constraints, and will therefore be guarded. The Activation List resides in the thread of the Active Object and is a buffer holding all the pending Method Requests. A Scheduler monitors the Activation List for Method Requests that meet their synchronization constraints. When the Scheduler chooses a Method Request to be executed, it dequeues the request from the Activation List and dispatches it to the servant, which initiates the actual execution of the method called by the client.

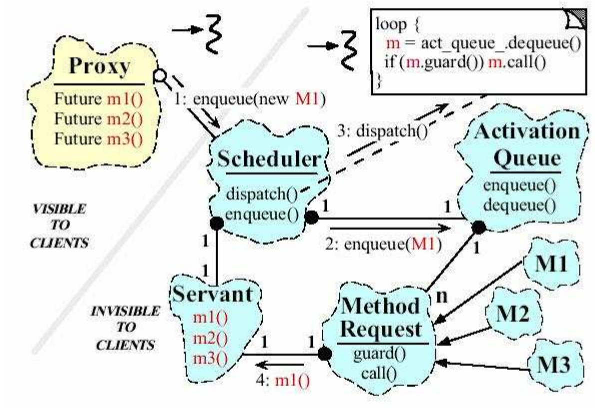


Fig. 2: Components of the AO pattern

[1]Figures obtained from online tutorials on patterns by Doug Schmidt.

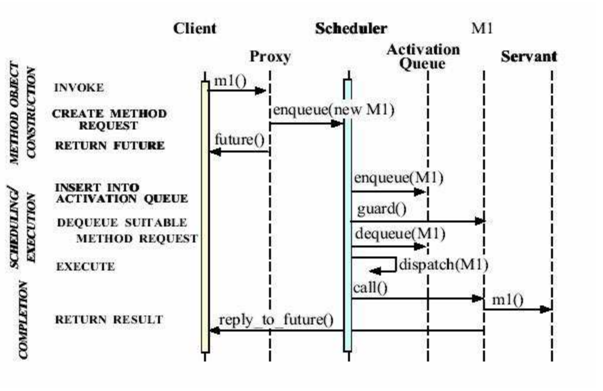


Fig. 3: Interactions among AO components

A method invocation that produces a result is called a two-way method request. In this case, a Future is created in the proxy following the invocation of the two-way method. This Future is a container that sits on the proxy and holds a place for the result produced by the method request. When the corresponding method request is executed, the servant will place the returned result in the Future place holder. The client may employ one of several strategies to wait for the return result, including blocking or polling for the system.

The Active Object pattern can be used to implement a class of producer/consumer, read/write and publish/subscribe systems. In this paper, we discuss its use in a producer/consumer system, which is then used to illustrate the performance analysis methodology.

B. Producer/consumer system

Figure 4 shows a producer/consumer system with two producers and a single consumer. The system is implemented using middleware to foster scalability, evolvability and interoperability [?]. This is achieved through the elimination of point to point communication between communicating entities and also due to the reduction of data interfaces. The middleware solution comprises of a Consumer Handler, which serves as a proxy to the consumer application. This handler contains a message queue for outgoing messages. The two producers "put" messages on the message queue. The message queue also contains a message broker which is responsible for monitoring the queue for new messages to be sent to the consumer. When the message queue contains messages, the message broker "gets" a message from the message queue and "sends" it to the consumer application.

The two producers and the consumer handler/message queue contend for mutually exclusive access to the message queue. The message queue is implemented with the Monitor Object pattern to allow thread safe access. The Consumer Handler exists in a single thread of control and when the message broker is actively involved in getting and sending the sending the message, the message queue is locked. When the message queue is locked by the Consumer Handler, the two producers will be denied access to the message queue.

Similarly, when one producer is putting a message on the message queue the other producer and the consumer handler will be denied access. Thus, once an entity acquires the mutex lock from the Monitor Object, it retains control until the transaction is complete, at which time it releases the lock. Thus, for the duration of the access times of the producers and consumers the message queue is locked.

In many systems, a distribution boundary exists between the entities in the system. In this scenario, the access times to the message queue are defined by the network latency, which can exhibit fluctuations alongside a busy work environment. As a result of the unpredictable latency, the entities that access the message queue can be starved from access, resulting in a loss of messages.

We now describe how the Active Object pattern could be used to implement the producer/consumer system shown in Figure 4. We also discuss how the performance issues could be alleviated by the use of the AO pattern. Figure 5 shows the implementation of the producer/consumer system using the AO pattern. It introduces a Producer Handler which serves as a proxy to the consumer handler's message queue for the producer client. The Producer Handler is implemented as a distributed AO to decouple the producer applications from the consumer handler's message queue.

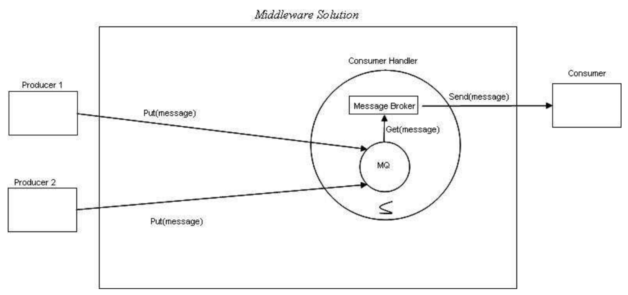


Fig. 4: A producer/consumer system

The AO proxy resides on the client application and provides the interface for the put method which places messages on the consumer handler's message queue. When the "put" command is invoked by the client, the proxy creates the corresponding method request and enqueues it on the Producer handler's Activation List. The synchronization constraint or the guard of of this put method request is the requirement of the proxy to have control of the message queue. When the method request is not guarded, the scheduler will dequeue the request and execute the method to put the message on Consumer Handler's message queue. The AO thus decouples the producer clients from the consumer application. The access time required to add messages on the Consumer Handler's message queue is reduced to the internal access time of the middleware. The impact of the network latency on the system is thus eliminated by the use of the AO.

The system also implements the Sending Service of the Message Broker as an AO as shown in Figure 5. A proxy interface containing the send method method is implemented

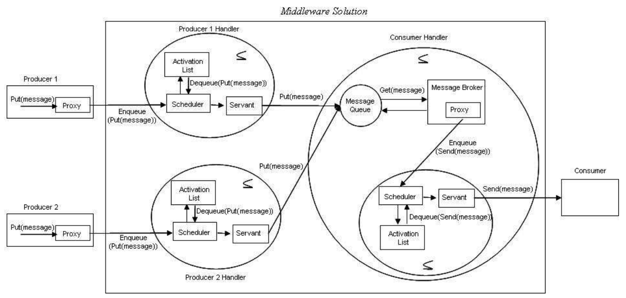


Fig. 5: A producer/consumer system implemented using AO pattern

inside the Message Broker. When the Message Broker invokes the method to send a message, a method request is created by the proxy and enqueued on the Activation List of the Sending Service AO. The send method request in the AO is guarded when the servant is busy sending the message to the consumer. This also allows the Message Broker and the Servant to work asynchronously. The Message Broker's relinquishes control of the message queue after getting the message and invoking the send command on the proxy. From the Message Broker's perspective, since the time taken to complete the send command is negligible, it retains control of the message queue only for the time taken to get the message, which is governed by the internal access time of the middleware. The AO thus shields the system from the impact of network latency on the consumer side.

Figure 6 depicts a model of our example in the POSAML language which allows us to use POSAML's generative capabilities to synthesize metadata for backend analysis tools, such as simulation and empirical benchmarking.

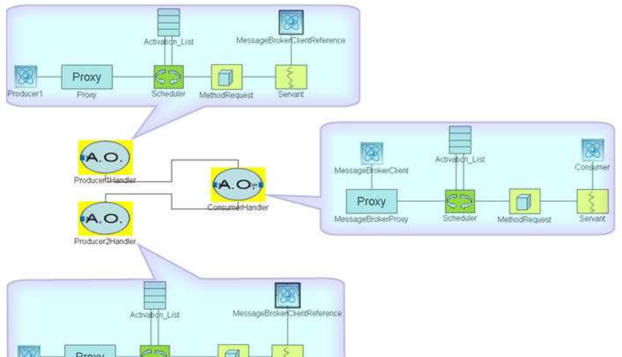


Fig. 6: A POSAML model of the Producer/Consumer AO System

C. Queuing models

In this section we describe the queuing models of the producer/consumer system implemented using the MO and AO patterns. Figure 7 shows the queuing model for system implementation using the MO pattern. We assume that the arrival process of messages at the producers is Poisson with rates λ_1 and λ_2. The producers then store these incoming messages in the producer-side buffers P_{S1} and P_{S2}, with capacities N_1 and N_2. The Consumer Handler's MQ is also modeled as a buffer with capacity Q. A producer can gain access to MQ as long as there is spare capacity. When a producer gains access to MQ, it takes a single message from the buffer and puts it on the message queue, after which it relinquishes control of the MQ.

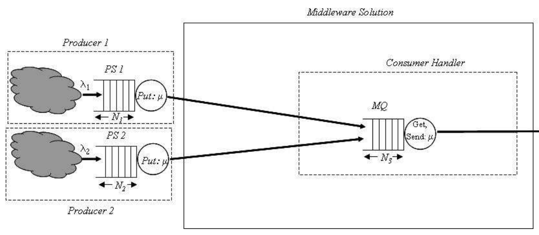


Fig. 7: Queuing model for producer/consumer system w/o AO

We assume that the time taken by a producer to put a message on the queue is exponentially distributed with parameter μ. The message broker also contends for access to MQ to get and send messages to the consumer. The broker can gain access to MQ as long as it is not empty. We assume that the time taken by the broker to send a message to the consumer is also exponentially distributed with parameter μ. The message broker also sends a single message before relinquishing control of the MQ. The completion times for put and send requests are assumed to be identically distributed, since it is expected that these times will be dominated by the network characteristics. The queuing discipline employed at producer-side buffers and at the MQ is first-come, first-serve.

Figure 8 shows the queuing model for system implementation using the AO pattern. In this case the producer-side Activation Lists are modeled as buffers labeled $PHAL_1$ and $PHAL_2$ with capacities N_3 and N_4 respectively. A producer can continue to invoke the put method until the Activation List in its corresponding Producer Handler has spare capacity to enqueue a Method Request. A producer feeds its corresponding Activation List at rate μ. The time taken to enqueue a message on the MQ by executing the put method request internally by the producer-side servant is exponentially distributed with parameter τ.

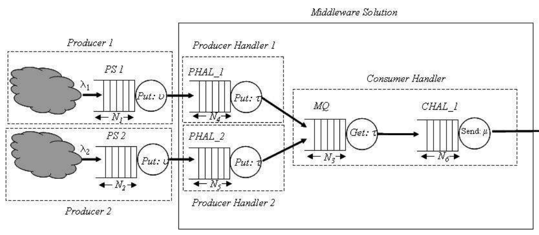
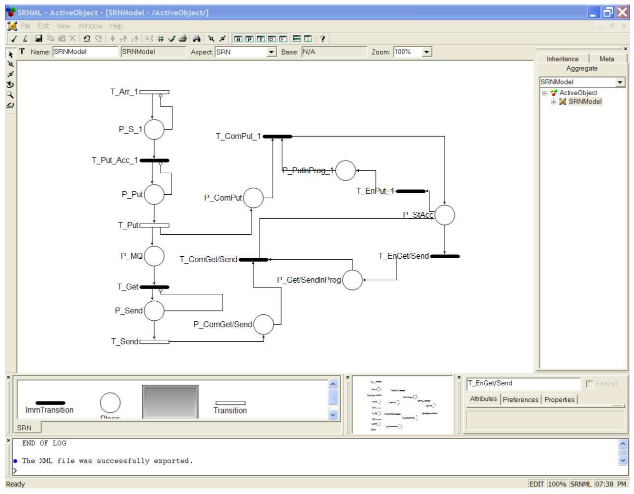


Fig. 8: Queuing model for producer/consumer system with AO

The producer-side side servant can put messages on MQ as long as it is not full. The consumer-side Activation List is also modeled as a buffer labeled $CHAL_1$ with capacity N_5. The time taken by the Message Broker to dequeue a message from MQ by executing the get method request is also exponentially distributed with parameter τ. The rate at which the consumer-side servant sends messages to the consumer is μ. The producer-side servants will not gain access to MQ if their corresponding activation lists are empty. Similarly, the message broker will not gain access to MQ if it is empty.

D. SRN implementation

Model automation for the performance analysis is achieved by modeling the SRN modeling of our example in the SRNML modeling language. Figure **??** illustrates a partial model of the SRN for the producer/consumer system we modeled in SRNML. SRNML enables the synthesis of metadata used by the SPNP solvers to conduct performance analysis.



```
END OF LOG
• The XML file was successfully exported.
```

Fig. 9: Partial SRNML Model for Producer/Consumer System

IV. Performance analysis of pattern composition

We demonstrate the feasibility of conducting model-based performance analysis of a system built using a composition of patterns, we consider a system built using Reactor and MO patterns as shown in Figure **??**.

V. Broader Impacts

In this section we outline an example of the broader impact of our work. This work deals with specialization of distributed computing infrastructures, such as middleware, operating systems, and virtual machines. Distributed computing infrastructures, such as middleware and virtual machines, are designed to be highly flexible and feature-rich to support a wide range of applications and product lines in multiple domains. Applications with stringent quality of service (QoS) demands (e.g., latency, fault tolerance, and throughput), however, find

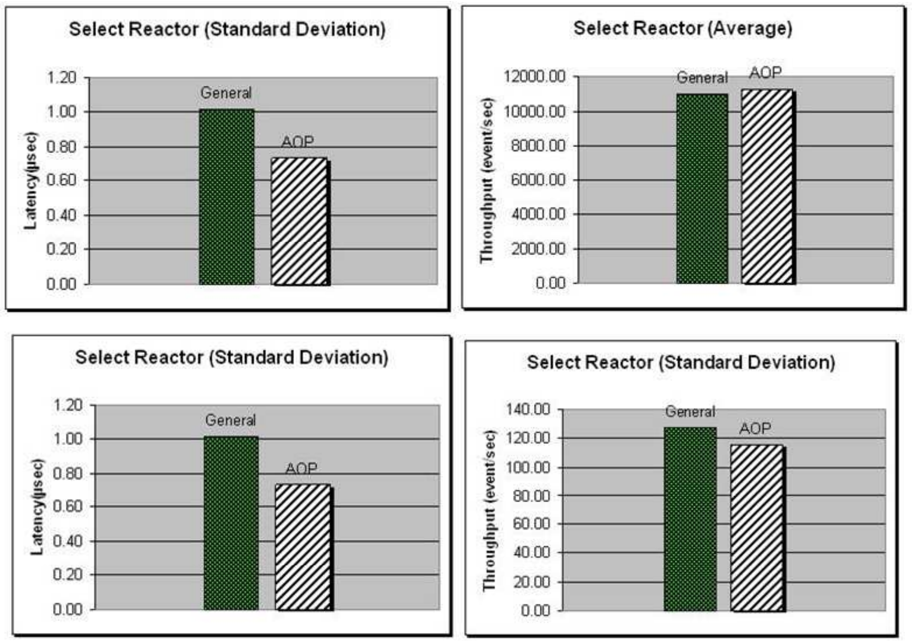


Fig. 10: Single threaded reactor

this feature richness and flexibility to be a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains.

We have leveraged the POSAML capabilities developed in this project in novel ways to automate middleware specialization. We achieve this effect by integrating model-based and aspect oriented software development (AOSD) techniques. As before we use POSAML to model the composition and configuration of a middleware systems stack using the patterns-based building blocks. We have demonstrated earlier [] how AOSD tools like AspectC++ can be used to specialize middleware source code. Our current research is investigating solutions based on generative tools within POSAML that can automate the synthesis of AspectC++ directives required for specialization.

We used our techniques in the context of specializing the Reactor pattern in the ACE middleware framework. We collected empirical data that compared the specialized version of ACE with the original version along different dimensions including end to end latency and throughput. We used the ACE middleware's performance test suite to conduct these performance tests and study the impact of AOP on latency and round trip throughput changes. Figure **??** demonstrates the initial set of results we obtained.

VI. Conclusions and future research

In our collaborative work supported by the CSR-SMA grant, our team has demonstrated an approach for design-time performance evaluation of complex, QoS-intensive systems by focusing on their software patterns-driven structure. We have demonstrated how individual patterns can be evaluated and how to evaluate a composition of these patterns. We have shown the use of MDE techniques to automate and scale a number of tedious and error-prone tasks in this process that results from having to manually develop these performance models. We also demonstrated the broader impact of our techniques for middleware specialization.

We are working on extending POSAML's capabilities to enable pattern composition so that techniques for performance evaluation of pattern composition can be automated. Moreover, we are adding behavioral modeling capabilities in POSAML using the Input/Output Automata to capture the interactions of patterns (both intra and inter). We will use these behavioral abstractions to provide model-to-model transformations so that POSAML models can be automatically converted to SRNML models, a step we currently do manually. Our MDE approaches can also broadly apply to recent NSF focus areas, such as the Cross System Integration.