Model-Driven Performance Analysis Methodology for Distributed Software Systems

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

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

Jeff Gray
Yuehua Lin
Dept. of CIS
Univ. of Alabama at 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 is of paramount importance in many distributed software systems. Despite this considerable influence, currently there is a lack of significant research to analyze performance of middleware at design time, where performance issues can be resolved at a much earlier stage of the application life cycle and with substantially less costs. The present project seeks to develop a performance analysis methodology for design-time performance analysis for distributed software systems implemented using middleware patterns and their compositions. The methodology is illustrated on a producer/consumer system implemented using the Active Object (AO) pattern in middleware. Finally, broader impacts of the methodology for middleware specialization are also described.**

I. Introduction

Society today is increasingly reliant on the services provided by distributed software systems. These services 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 set of services with good performance, while simultaneously reducing their time-to-market, service providers are increasingly favoring the assembly-centric approach over the traditional development-centric approach. A key facilitator of this assembly-centric approach has been *QoS-enabled middleware* [18]. 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 [4], [20], 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 many software 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 is comprised of two steps. The first step consists of formulating and solving performance models of individual middleware patterns. In the second step, strategies to compose the performance models of individual patterns mirroring their composition 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) [19] where the systems developer is provided artifacts that are intuitive and closer to their domain to compose the systems from building blocks. Generative tools [2] supported by the MDE approach can then automate the synthesis of performance analysis metadata that is subsequently used by back-end analysis tools. The illustration of the first step of the methodology on Reactor, Proactor and Active Object (AO) patterns demonstrates the feasibility of conducting performance analysis of a system implemented using a middleware pattern at design time using the model-driven paradigm.

The rest of the paper is organized as follows: Section II; Section II provides an overview of the performance analysis process of a middleware pattern. Section III illustrates the process using the AO pattern. Section IV discusses broader impacts of the project. Section V offers concluding remarks and directions for future research.

1-4244-0910-1/07/$20.00 ©2007 IEEE

II. Performance analysis of a middleware pattern

In this section we discuss the process we follow for performance analysis of an individual pattern. We describe the various steps involved in the process shown in Figure 1 and how they support each other.

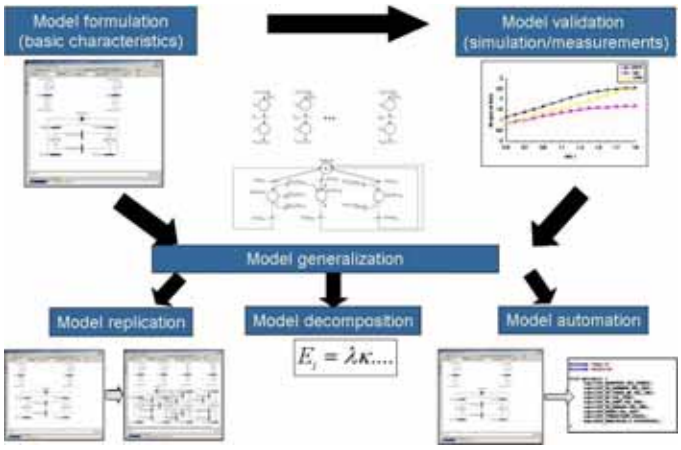

Fig. 1. Performance analysis process of a middleware pattern

- **Model formulation:** The model formulation step is comprised of capturing the basic or the invariant characteristics of a pattern into a performance model. Different modeling paradigms such as Stochastic Reward Nets (SRNs) [17], Layered Queuing Networks [23], and Colored Petri Nets [11] may be used for this purpose. The performance model can then be solved/simulated using tools such as SPNP [10] and DesignCPN [12].
- **Model validation:** The performance estimates obtained by solving the performance model are validated in this step using simulation or experimentation. Experimentation is conducted by implementing a system with the pattern on the ACE framework `www.dre.vanderbilt.edu/ACE`. Simulation is conducted using a general-purpose simulation language such as CSIM [21].
- **Model generalization:** In this step the the process of formulating the model is generalized to enable the system developer to customize the model according to the system at hand and to determine system boundaries and limits.
- **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 [1]. 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) [14] 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 for simulation, analysis or empirical benchmarking [15], [13].
- **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 [9] 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 (Constraint-Specification Aspect Weaver) [8] tool for this purpose.

This process has been illustrated on the Reactor [5], [7], [6], [14], [9], Proactor [15], [16] and AO patterns.

III. Performance analysis of an AO-based System

We illustrate the performance analysis process using the AO pattern in this section.

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 AO 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 by decoupling method invocation from method execution and creating the shared resource in its own thread of control.

The AO [18] is composed of the following components: Proxy, Activation List, Scheduler, Servant, and Method Requests. The interactions between the components is initiated by a client thread invoking a method on the proxy to the AO. 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 AO. 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. It also has guards or synchronization constraints. The Activation List resides in the thread of the AO and is a buffer holding all the pending Method Requests. A Scheduler monitors the Activation List for Method Requests that meet their synchronization constraints. It chooses a Method Request to be executed, dequeues the request and dispatches it to the servant, which initiates the actual execution of the method called by the client.

The AO pattern can be used to implement a class of producer/consumer, read/write and publish/subscribe systems.

B. Producer/consumer system

Figure 2 shows a producer/consumer system with two producers and a single consumer. The system is implemented using middleware to foster scalability, evolvability and interoperability [18]. 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 is comprised 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 (MO) pattern [18] 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 MO, 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 case, the access times to the message queue are defined by the network latency, which can fluctuate alongside a busy work environment. As a result of unpredictable latency, the entities that access the message queue can be starved from access, resulting in a message loss.

We now describe how the AO pattern could be used to implement the producer/consumer system shown in Figure 2. We also discuss how the performance issues could be alleviated by the use of the AO pattern. Figure 3 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.

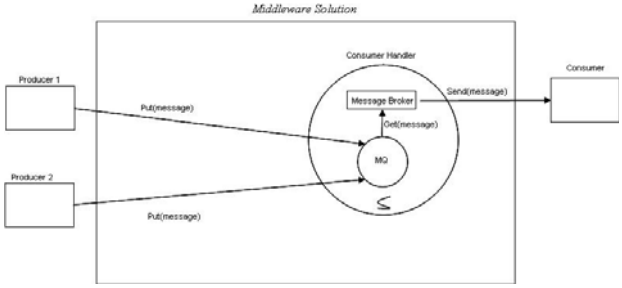

Fig. 2. A producer/consumer system

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 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 message queue is reduced to the internal access time of the middleware. The impact of the network latency on the system is thus eliminated.

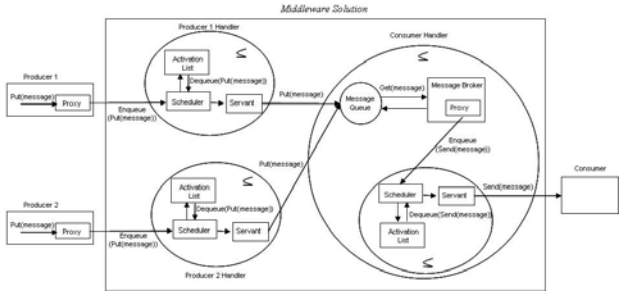

Fig. 3. A producer/consumer system implemented using AO pattern

The system also implements the Sending Service of the Message Broker as an AO as shown in Figure 3. A proxy interface containing the send method is implemented 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 relinquishes control of the message queue after getting the message and invoking the send command on the proxy. From the Message Broker's perspective, because 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. The AO thus shields the system from the impact of network latency on the consumer side.

Figure 4 depicts a model of our example in the POSAML language which allows us to use POSAML's generative capabilities to synthesize metadata for back-end analysis tools, such as simulation and empirical benchmarking.

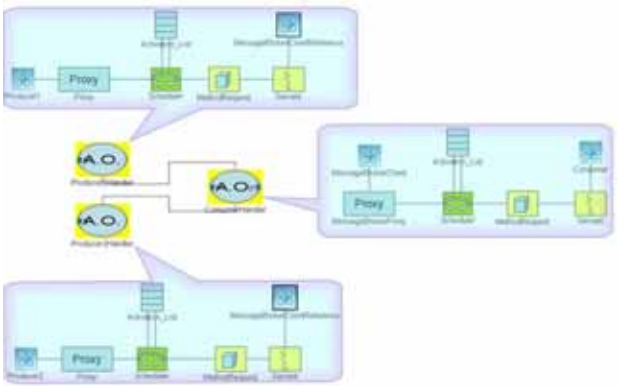

Fig. 4. A POSAML model of the Producer/Consumer AO System

C. Queuing models

In this section we describe the queuing models of the producer/consumer system implemented using the MO and AO patterns. Figure 5 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.

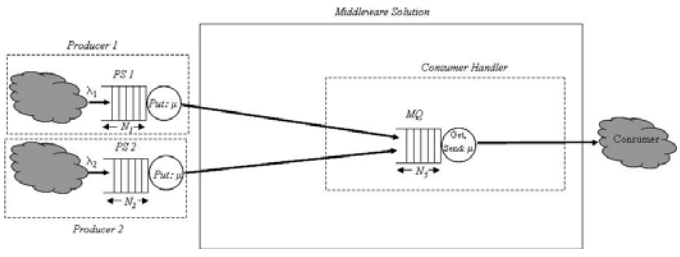

Fig. 5. Queuing model for producer/consumer system w/o AO

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. It can gain access to MQ as long as the queue 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 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, because it is expected that these times will be dominated by the network characteristics. The queuing discipline at producer-side buffers and at the MQ is first-come, first-serve.

Figure 6 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.

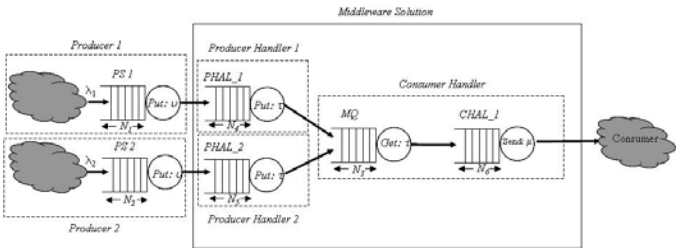

Fig. 6. Queuing model for producer/consumer system with AO

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

Figure 7 shows the SRN model of the system implemented using the MO pattern, the model of the system using the AO pattern is not shown here due to space limitations. In the figure transitions T_{Arr1} and T_{Arr2} represent the arrival of data messages at the producers, which are buffered in the queues, represented by places P_{S1} and P_{S2} while they wait to be sent to the Consumer Handler's Message Queue. Transitions $T_{PutAcc1}$, $T_{PutAcc2}$, and T_{Get} represent the threads that are continually contending for access to the Consumer Handler's Message Queue. When either $T_{PutAcc1}$ or $T_{PutAcc2}$ gain access to the queue, they place a token in place P_{Put}, which

represents the data message that is currently being put onto the message queue by transition T_{Put} at a rate of μ. Transition T_{Get} represents the get function of the Message Broker, which then uses the Sending Service represented by place P_{Send} and transition T_{Send}, which fires at the rate of μ.

An inhibitor arc from $P_{S1}(P_{S2})$ to $T_{Arr1}(T_{Arr2})$ has a multiplicity of $NN1(NN2)$, which sets the capacity of the producer-side buffers. Additionally, there are inhibitor arcs from P_{Put} to $T_{PutAcc1}$ and $T_{PutAcc2}$ of to allow only one token into the place P_{Put}, since only one data message can be put at a time. The final inhibitor arc is from P_{Send} to T_{Get} with multiplicity TP, allowing as many simultaneous transmission of the messages as the size of the thread pool.

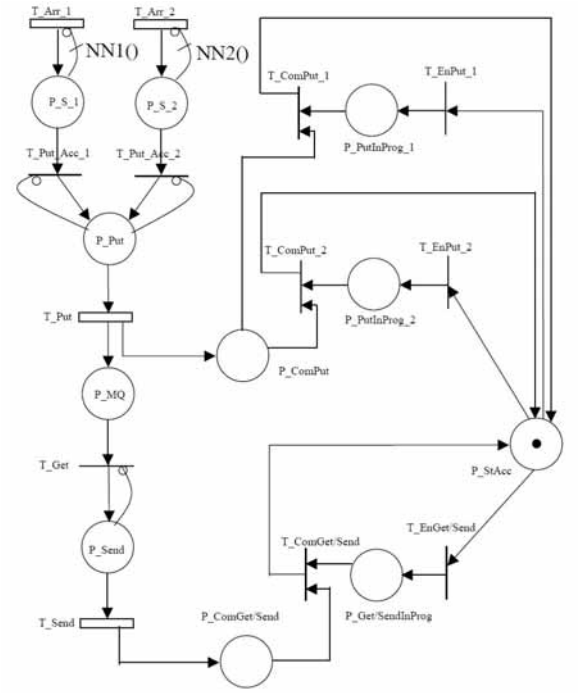

Fig. 7. SRN model for producer/consumer system w/o AO

E. Model automation

Model automation is achieved by modeling the SRN shown in Figure 7 in the SRNML modeling language. Figure 8 illustrates a partial model of the SRN in SRNML which enables the synthesis of metadata used by the SPNP solvers.

IV. Broader Impacts

In this section we outline an example of the broader impact of our work. It deals with specialization of distributed computing infrastructures, such as middleware, operating systems, and virtual machines, which 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 QoS demands (e.g., latency, fault tolerance, and throughput), however, find this feature richness and flexibility a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains.

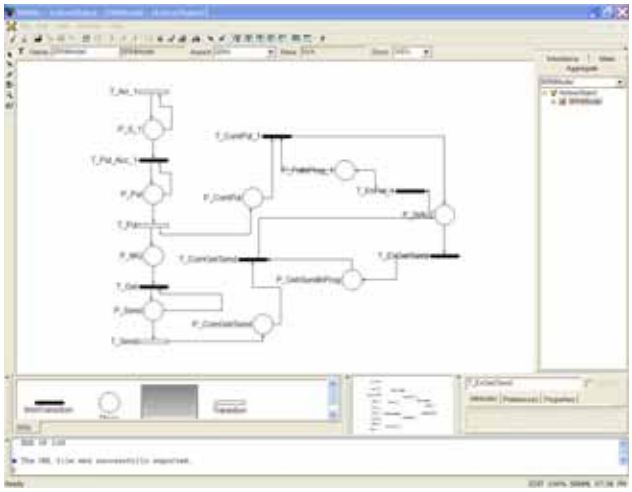

Fig. 8. Partial SRNML Model for Producer/Consumer System w/o AO

We have leveraged the POSAML capabilities from this project in novel ways to automate middleware specialization. We achieve this by integrating model-based and aspect oriented software development (AOSD) techniques [3]. 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 how AOSD tools like AspectC++ [22] 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 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 9 demonstrates the initial set of results we obtained.

V. Conclusions and Future Research

In our collaborative work supported by the CSR-SMA grant, we have 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. 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.

Our future research is concerned with demonstration of the second step of the methodology, namely, composition

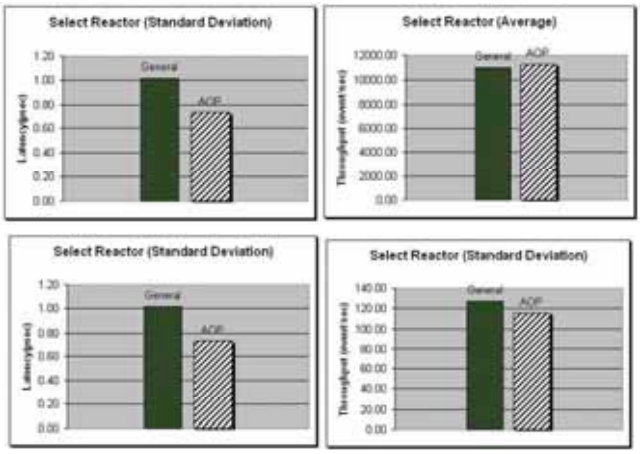

Fig. 9. Single threaded reactor

of performance models using a composition of Reactor and MO patterns. We are also 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.

ACKNOWLEDGMENTS

This research was supported by the following grants from NSF: Univ. of Connecticut (CNS-0406376 and CNS-SMA-0509271), Vanderbilt Univ. (CNS-SMA-0509296) and Univ. of Alabama at Birmingham (CNS-SMA-0509342).

REFERENCES

[1] G. Ciardo and K. S. Trivedi. "A decomposition approach for stochastic reward net models". *Performance Evaluation*, 18(1):37–59, July 1993.

[2] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, 2000.

[3] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

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

[5] S. Gokhale, A. Gokhale, and J. Gray. "Response time analysis of an event demultiplexing pattern in middleware for network services". In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM), Symposium on Advances for Networks and Internet*, pages 699–704, St. Louis, MO, November 2005.

[6] S. Gokhale, A. Gokhale, and J. Gray. "Performance analysis of a middleware demultiplexing pattern". In *Proc. of Hawaii Intl. Conference on System Sciences (HICSS)*, January 2007.

[7] S. Gokhale, P. Vandal, U. Praphamontripong, A. Gokhale, and J. Gray. "Performance analysis of the reactor pattern in network services". In *Proc. of the 5th Intl. Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS 2006), co-located with IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006.

[8] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. "An approach for supporting aspect-oriented domain modeling". In *Proc. of the 2nd Intl. Conference on Generative Programming and Component Engineering*, pages 151–168, September 2003.

[9] J. Gray, Y. Lin, J. Zhang, S. Nordstrom, A. Gokhale, S. Neema, and S. Gokhale. "Replicators: Transformations to address model scalability". In *Lecture Notes in Computer Science: Proceedings of 8th International Conference Model Driven Engineering Languages and Systems (MoDELS 2005)*, pages 295–308. Springer Verlag, November 2005.

[10] C. Hirel, B. Tuffin, and K. S. Trivedi. "SPNP: Stochastic Petri Nets. Version 6.0". In *Proc. of Computer Performance Evaluation: Modeling Tools and Techniques, 11th Intl. Conference, Lecture Notes in Computer Science 1786*, 2000.

[11] K. Jensen. *Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer Verlag, Germany, 1997.

[12] K. Jensen. "DesignCPN, version 4.0". Technical report, University of Denmark, Aarhus, 1999.

[13] A. Kogekar and A. Gokhale. "Performance evaluation of the reactor pattern using the OMNeT++ simulator". In *Proc. of the 44th Annual Southeast Conference*, Melbourne, FL, April 2006. ACM.

[14] A. Kogekar, D. Kaul, A. Gokhale, J. Gray, and S. Gokhale. "Managing variability in middleware provisioning using visual modeling languages". In *Proc. of Hawaii Intl. Conference on System Sciences (HICSS)*, January 2007.

[15] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray. "Performance analysis of an asynchronous Web server". In *Proc. of Intl. Conference on Computer Science and Applications (COMPSAC)*, pages 22–25, Chicago, IL, September 2006.

[16] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray. "An analytical approach for performance analysis of an asynchronous Web server". *Simulation: Transactions of the Society of Modeling and Simulation International (Accepted for publication)*, 2007.

[17] A. Puliafito, M. Telek, and K. S. Trivedi. "The evolution of stochastic Petri nets". In *Proc. of World Congress on Systems Simulation*, pages 3–15, Singapore, September 1997.

[18] R. E. Schantz and D. C. Schmidt. "Middleware for distributed systems: Evolving the common structure for network-centric Applications". In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*, pages 801–813. Wiley & Sons, New York, 2002.

[19] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):41–47, 2006.

[20] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[21] H. Schwetman. "CSIM reference manual (revision 16)". Technical Report ACA-ST-252-87, Microelectronics and Computer Technology Corp., Austin, TX.

[22] O. Spinczyk, A. Gal, and W. Schrder-Preikschat. "AspectC++: An aspect-oriented extension to C++". In *Proc. of Intl. Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, February 2002.

[23] C. M. Woodside. "Tutorial introduction to layered modeling of software performance", 2002. `http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialg.pdf`.