POSAML: A Visual Modeling Framework for Middleware Provisioning

Dimple Kaul, Arundhati Kogekar and Aniruddha Gokhale
ISIS, Dept of EECS
Vanderbilt University
Nashville, TN 37235
{dkaul,akogekar,gokhale}@dre.vanderbilt.edu

Jeff Gray
Dept of CIS
Univ of Alabama at Birmingham
Birmingham, AL
gray@cis.uab.edu

Swapna Gokhale
Dept of CSE
Univ of Connecticult
Storrs, CT
ssg@engr.uconn.edu

Abstract

Effective provisioning of next generation distributed applications hosted on diverse middleware platforms incurs significant challenges due to the applications' growing complexity and quality of service (QoS) requirements. An effective provisioning of the middleware platform includes a composition and configuration of the middleware services that meets the application QoS requirements under expected workloads. Traditional techniques for middleware provisioning tend to use non-intuitive, low-level and technology-specific approaches, which are tedious, error prone, non-reusable and not amenable to ease of QoS validation. Additionally, most often the configuration activities of the middleware platform tend to be decoupled from the QoS validation stages resulting in an iterative trial-and-error process between the two phases. This paper describes the design of a visual modeling language called POSAML (Patterns Oriented Software Architecture Modeling Language) and associated tools that provide an intuitive, higher level and unified framework for provisioning middleware platforms. POSAML provides visual modeling capabilities for middleware-independent provisioning while allowing automated middleware-specific QoS validation.

1 Introduction

Rapid advances in hardware and networking technologies are fostering an unprecedented growth in complex applications and services with different quality of service (QoS) requirements. Standardized middleware technologies (e.g., J2EE [22], .NET [13] and CORBA Component Model (CCM) [14]) coupled with advances in integration technologies (e.g., web services described by XML-basedstandards) enable the construction of application functionality by connecting individual services spread across distributed resources.

Several functional and non-functional concerns must be addressed simultaneously when provisioning complex distributed applications on these middleware platforms. The non-functional provisioning concerns comprise the problem of choosing the right set of configuration and composition parameters of the middleware platforms and validating that these meet the QoS requirements of the applications. Traditional approaches to middleware provisioning typically use low-level, non-intuitive, and technology-specific mechanisms, which are not reusable across multiple middleware technologies.

From our experience collaborating with industry practitioners [20], we have seen approaches to provisioning that required the manual configuration of XML files that are several thousand lines long. In traditional approaches to provisioning, often the QoS validation phase is decoupled from the configuration phase. Moreover, the validation phase uses processes that do not leverage decisions made at the configuration phase, which limits the optimizations and fidelity of the QoS validation phases. Overall, this results in an ad hoc, iterative process for middleware provisioning, which may adversely impact application time-to-market.

A solution to this problem is to provide a mechanism that raises the level of abstraction at which system integrators can provision middleware. Visual aids are one of the best known techniques to intuitively reason about any system [7]. Visual tools have been highly successful in the domain of simulations (e.g., Matlab Simulink) and design (e.g., Cadence tools for circuit design or AutoCAD). Visual tools thus hold promise for middleware provisioning also.

A desirable visual tool for the provisioning problem is one that can provide a clean separation of concerns [15] between the configuration and QoS validation phases, yet unifies the two phases such that decisions at one phase can automate and optimize steps at subsequent phases. These qualities largely eliminate the overhead of the trial-and-error, iterative process incurred by traditional methodologies.

This paper describes POSAML (Patterns-oriented Software Architecture Modeling Language), which is a visual domain-specific modeling language (DSML) [9, 10], and an associated set of generative programming tools [4] that meet the desired properties of a visual tool for middleware provisioning. Fundamental to POSAML is the notion of middleware building blocks that are viewed as being made up of software patterns [6, 19]. Reasoning about middleware systems in terms of visual models of patterns not only raises the level of abstraction, but also offers a technology-independent solution to middleware provisioning. Furthermore, capturing the essence of the provisioning decisions in visual models and using generative programming tools improves the potential reuse capabilities that can be applied across contemporary middleware platforms.

The rest of the paper is organized as follows: Section 2 introduces the challenges in designing a visual framework for middleware provisioning; Section 3 describes the design of the POSAML visual framework for middleware provisioning; Section 4 compares our work to existing research; and Section 5 concludes with a description of lessons learned and future work.

2 Designing Visual Tools for Middleware Provisioning

Middleware provisioning is the activity that comprises the configuration and customization of the middleware platform and validating that these meet the QoS needs of the application under expected workloads. The motivation for visual tools in middleware provisioning stem from the non-intuitive, non-reusable and error-prone nature of traditional approaches. For visual tools to be effective, they must meet a set of criteria described below and resolve the challenges arising in meeting these criteria.

Criterion 1: Accounting for variability across a range of middleware technologies: Figure 1 illustrates the structure of contemporary middleware technologies. It depicts multiple layers of middleware each of which addresses specific requirements and provides reusable functional capabilities. For example, the host infrastructure middleware provides a uniform layer of abstraction to mask the heterogeneity arising from different operating systems, hardware and networks; the distributed middleware provides location transparency; common services include directory services, messaging services, and transaction services among others; and domain-specific services include additional reusable capabilities that are specific to a domain (e.g., avionics or telecom).

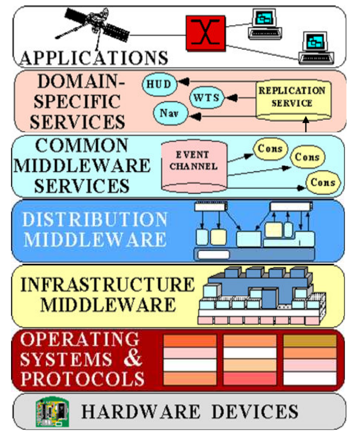

Figure 1. Middleware Structure

Distributed applications are typically hosted on multiple heterogeneous middleware platforms in a networked environment. For each host in the deployment environment, the middleware stacks on which an application is hosted may need to be fine-tuned in different ways to meet the different QoS requirements of applications. To support a wide range of application QoS needs, contemporary middleware technologies provide several different reusable capabilities that can be individually configured and composed with each other. This flexibility offered by individual middleware technologies gives rise to variability that a middleware provisioner faces when provisioning applications on the platforms.

The visual tool used for middleware provisioning must handle this variability in the context of application QoS needs, and provide an intuitive user interface to the middleware provisioner to eliminate provisioning errors. Our approach to resolve these challenges is based on abstracting away the implementation- and technology-specific details of contemporary middleware solutions and focus on the patterns of reuse [6] that form the building blocks of the different layers of middleware. Section 3.1 describes how we leverage and formalize these insights to design and implement the POSAML visual tool for middleware provisioning.

Criterion 2: Need for a unified framework: Middleware provisioning needs to be guided by the application QoS needs. This requires that the QoS validation must be performed based on decisions made in the configuration and customization phase. Thus, the QoS validation phase needs to have complete knowledge of the configuration decisions. The QoS validation phase requires systems developers to develop appropriate application testing and middleware benchmarking code in accordance to the configuration decisions.

This requirement adds a new dimension of variability and dependability to the challenges described in Criterion 1. To address both of these criteria, visual mechanisms should provide a unified framework that can address both the configuration and QoS validation concerns that arise in middleware provisioning. Such a framework must provide the means to capture the configuration decisions and make them available in the QoS validation phase. Section 3.2 describes how POSAML provides intuitive abstractions of the middleware stacks to the system integrators, which eases the task of configuration and customization. Additionally, it enables the QoS validation of the configured system using the same visual capability.

Criterion 3: Separation of concerns: As noted earlier, provisioning involves both the configuration and the validation phases. Criterion 1 illustrated the variability demonstrated by contemporary middleware technologies and calls for a visual mechanism that provides intuitive user interfaces to middleware provisioners. Criterion 2 discussed the need for a unified framework since the QoS validation phase is dependent on the configuration phase. Addressing these two criteria requires careful design since any ad hoc design decisions may tangle the QoS validation activities with the configuration activities defeating the goals of providing intuitive mechanisms for middleware provisioning.

Although traditional approaches to middleware provisioning decouple the configuration and validation phases, such a decoupling is not useful since the QoS validation activities do not have any knowledge of the configuration decisions, and these activities are typically carried out by a different set of actors. To address the need to decouple the two phases yet meet the earlier two criteria requires that the visual tool provide a clean separation of the QoS validation phase from the configuration phase so that the two concerns do not tangle with each other yet unifies them at a level that does not impact the user perception. Section 3.3 describes how POSAML provides these separation of concerns at the user interface level within the unified framework.

3 The POSAML Visual Framework for Middleware Provisioning

This section describes the POSAML (Pattern Oriented Software Architecture Modeling Language) visual modeling framework for middleware provisioning. We discuss the design of POSAML and how it meets the criteria described in Section 2.

3.1 Meeting Criterion 1: Accounting for Variability Across a Range of Middleware

Figure 1 illustrated the structure of contemporary middleware, which are made up of different layers of software performing various functions, such as data marshaling, event handling, brokering, concurrency handling and connection management. In an object-oriented design of a middleware framework, these capabilities are realized by building blocks based on proven patterns of software design [6]. A software pattern codifies recurring solutions to a particular problem occurring in different contexts, which is embodied as a reusable software building block in middleware. The architectural patterns found in contemporary middleware systems are discussed extensively in the book *Pattern Oriented Software Architecture: Patterns for Networked and Concurrent Systems* (POSA) [19]. We leverage these characteristics of middleware design to develop our visual provisioning framework.

In our approach, a visual representation of these patterns enables system provisioners to view the middleware stacks at a higher level of abstraction, which is independent of any specific middleware technology. The QoS validation mechanisms associated with the visual capability subsequently map these abstractions to technology-specific platforms. In the following we discuss how patterns [6] and pattern languages [1] enable us to identify and resolve the variability in middleware provisioning.

(a) Identifying Variability in Composing Functionality

When deploying complex applications, systems provisioners must decide the composition and customization of middleware that hosts the application. Middleware composition includes assembling individual but compatible building blocks of middleware at multiple layers. The systems provisioner chooses a block based on various factors including the context in which the application will be deployed, the concurrency and distribution requirements of the application, the end-to-end latency and timeliness requirements for real-time systems, or throughput for other enterprise systems (e.g., telecommunications call processing). We refer to this provisioning variability as *middleware compositional variability*.

Figure 2 illustrates a family of interacting patterns that form a pattern language [1] for middleware designed to support such applications. The middleware can be customized by composing compatible patterns. For example, event demultiplexing and dispatching via the Reactor or Proactor patterns [19] can be composed with the concurrent event handling provided by the Leader-Follower or Active Object patterns [19]. However, an Asynchronous Completion

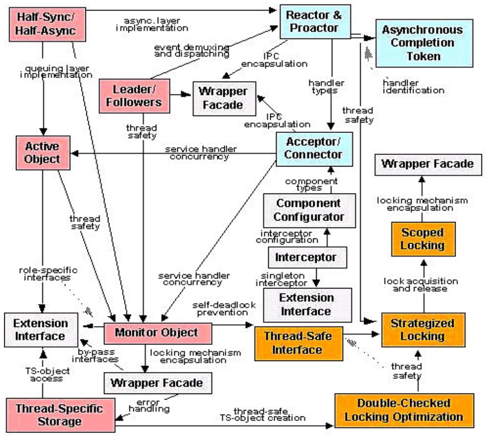

Figure 2. Middleware Patterns and Pattern Languages

Token (ACT) pattern works only with asynchronous event demultiplexing provided by the Proactor. Thus, a combination of Reactor and ACT is invalid. Our POSAML visual framework formalizes the concept of a pattern language to resolve the middleware compositional variability concerns in middleware provisioning.

(b) Identifying Variability in Building Block Configurations

Middleware developers provide numerous configuration options to customize the behavior of individual building blocks. This flexibility further exacerbates the already incurred variability in design choices that the systems provisioner is required to make. Since the variability is on a per building block basis – as opposed to a composition described above – we refer to this as *building block configuration variability*.

As a concrete example, the Reactor pattern can be configured in many different ways depending on the event demultiplexing capabilities provided by the underlying OS and the concurrency requirements of an application. For example, the demultiplexing capabilities of a Reactor could be based on the `select()` or `poll()` system calls provided by POSIX-compliant operating systems or `WaitForMultipleObject()` available on Windows. Moreover, the handling of the event in the Reactor's event handler can be managed by a single thread of control or handed over to a pool of threads depending on the concurrency requirements. POSAML captures the representation of individual patterns and their variations to address the per building block configuration variability.

3.2 Meeting Criterion 2: Unified Framework

In this section, we discuss how the insights we gain from exploring the patterns and pattern languages become part of our visual framework for middleware provisioning. Section 2 describes the need for a unified framework that enables the QoS validation phase of provisioning to leverage the decisions made at the configuration phase. Model-based solutions based on visual aids can provide the desired solution. Model-driven Engineering (MDE) [10, 17], which is a model-based solution, has gained prominence in providing the capabilities to model systems and use generative programming techniques to synthesize artifacts that result from the models. This paper describes the POSAML MDE framework for assisting provisioners to make the right choices in configuring and composing large systems and validating that these meet the applications' QoS.

Our research contributions within the unified MDE framework include the design of a visual domain-specific modeling language (DSML) called POSAML (*Patterns-oriented Software Architecture Modeling Language*), which enables the modeling of middleware stacks and their configurations by providing intuitive visual abstractions of middleware building blocks. POSAML also provides middleware-specific QoS validation by virtue of allowing different model interpreters to be plugged into the MDE framework.

3.2.1 Visual Modeling Capabilities in POSAML

Figure 3 shows the metamodel for the top-level view of POSAML, which has been developed using the Generic Modeling Environment (GME) [12]. GME is a tool that enables domain experts to develop visual modeling languages and generative tools associated with those languages. The modeling languages in GME are represented as metamodels. A metamodel in GME depicts a class diagram using UML-like constructs showcasing the elements of the modeling language and how they are associated with each other. For example, the "Model" element defines an element that can comprise other elements. The "Connection" element describes the type of association between other modeling elements of the language. The "Aspect" element describes a specific view provided by the modeling environment thereby promoting separation of concerns within a modeling environment. By providing such views, the modeling environment effectively allows visual separation of concerns which is a criterion that we had to meet. The same

GME environment can be used by provisioners to model the provisioning decisions using the POSAML metamodel.

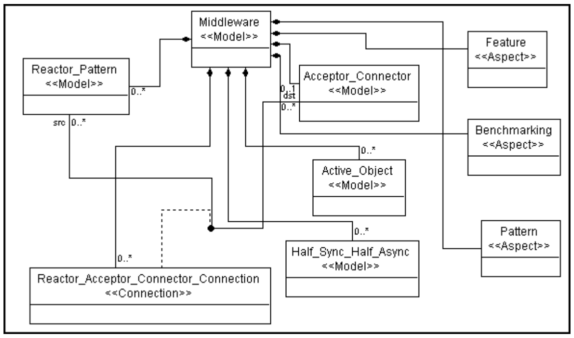

Figure 3. Top-level Metamodel of Middleware Structure

The metamodel illustrated in Figure 3 consists of the visual syntactic and semantic elements that describe individual patterns, and specifies how they can be connected to each other according to the pattern language. The figure also illustrates how the metamodel separates the concerns of modeling the pattern, its configuration and their compositions, and system QoS validation by virtue of using GME aspects. This separation is shown as three aspects; namely Pattern, Feature and Benchmarking.

The unified framework allows middleware provisioners to plug in different model interpreters that can automatically synthesize artifacts that are specific to a particular middleware technology.

3.2.2 Generative Capabilities in POSAML

A unified framework must provide the mechanisms for the decisions made at configuration time to be available at QoS validation time, and enable the synthesis of validation artifacts. In an MDE framework, such capabilities are realized via generative programming capabilities. Within the GME DSML development environment, in particular, these capabilities are realized by GME model interpreters, which traverse the graphical hierarchy of a model. The POSAML metamodel is a middleware-independent modeling language. By leveraging the GME environment's capabilities, different middleware-specific QoS validation interpreters can be plugged in. The following describes two model interpreters that we have developed:

(a) Configurator Interpreter

This interpreter is used to generate two artifacts which are required to configure middleware. One of the generated artifacts is a configurator file and the other one is a script file. The configurator file is used to set QoS related configuration policies using middleware-specific mechanisms for different applications. The middleware provisioner is shielded from these details since the interpreters automate the task of generating the platform-specific details.

(b) Benchmark Interpreter

The configuration and QoS modeling capabilities in POSAML serve as inputs to determine the benchmarking artifacts necessary to validate QoS. Using the benchmarking interpreter, which can be plugged into POSAML via the GME environment, the provisioner can generate benchmarking parameters for an existing benchmarking library. These parameters include artifacts, such as the number of data exchanges, the number of client threads, the data to be sent, the number of event handlers and the service time (in case of reactor). They are generated in XML format and can be used to parametrize an existing benchmarking library.

3.3 Meeting Criterion 3: Separation of Concerns

Section 3.2 described the POSAML unified framework. The visual modeling capabilities allow a systems provisioner to model middleware provisioning decisions while the generative capabilities automate the QoS validation. In this section we describe how the unified POSAML framework separates the concerns of configuration and QoS validation within the unified framework.

The middleware configuration is accomplished through POSAML's feature modeling [4,5] capability, which assists a systems engineer in configuring a variety of different middleware features (e.g., choosing the pattern and its configuration parameters). The benchmarking capability (though visually decoupled from the provisioning capability) is internally integrated and is used in automatically synthesizing empirical benchmarks for the provisioned system to perform QoS validation. The remainder of this section describes these capabilities of POSAML.

3.3.1 Feature Modeling in POSAML

A Feature model [4] is defined as an abstraction of a family of systems in a particular domain capturing commonalities and variabilities among the members of the family. In our POSAML modeling language, a feature modeling aspect provides domain-specific artifacts to model a system, in contrast to using low-level platform-specific artifacts. The feature modeling capabilities in POSAML provide structural representations of different possible middleware pattern properties. In our case, the feature modeling comprises several non-functional and QoS requirements, such as the

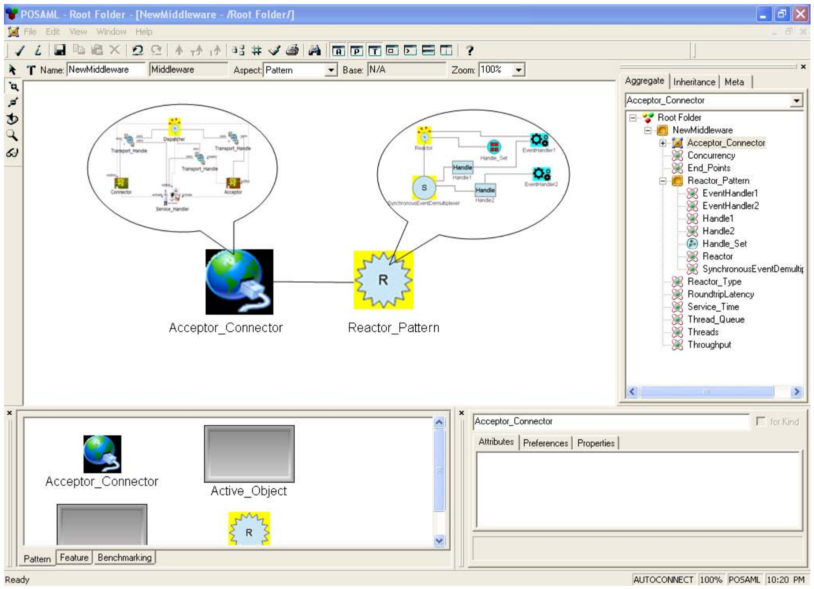

Figure 4. Middleware Provisioning in POSAML

choice of network transport, listening end points, concurrency requirements, and periodicity of requests, all represented as higher-level artifacts.

This level of modeling enables system provisioners to select various strategies, resource settings and factories within the middleware that can be parameterized according to user needs by driving the selection process using the visual feature modeling framework. For example, the designer can specify the "End points" feature for the Acceptor-Connector Pattern to describe the ports and communication mechanisms used by the client and server to communicate with each other.

3.3.2 QoS Validation Modeling in POSAML

To enable the performance analysis of a configured and customized system, the modeling language provides a method to model benchmarking characteristics. The POSAML metamodel enables the systems provisioner to model the workload (number of threads, data, and the number of data exchanges), as well as which metric to measure (latency or throughput). It is also possible to set the service times for event handlers within the application. In the QoS validation view, the provisioner can select which benchmarking parameters to select for the performance analysis of the modeled system. These parameters are then automatically written to an XML file by the benchmarking interpreter described earlier. This file can be used by an existing benchmarking library within the middleware which is being benchmarked.

3.4 POSAML in Action

This section describes the workflow of activities performed by a systems provisioner using the capabilities of POSAML. We use a sample client-server application to demonstrate concretely the capabilities of POSAML. We focus on a subset of the middleware blocks used by the sample application.

Step 1: Modeling Application Structure

Figure 4 shows an example where the provisioner has modeled the sample application as a composition of the Reactor and Acceptor-Connector patterns. The Reactor exemplifies the event handling within the server, while the Acceptor-Connector demonstrate the communication mechanisms between the client and server. In addition to this high-level view, the user can click on any one of the patterns and model its internals, as shown in the ovals in the figure (whose details are described below). From the figure, it can be seen that POSAML follows a hierarchical modeling structure. At the top-most level one can model inter-pattern relationships and constraints. At the lower level, a provisioner can drill down into each pattern to model the participants of the pattern and the inter-pattern relationships between them.

Step 2 (a): Modeling the Reactor Pattern

The ability to handle and dispatch simultaneously occurring events effectively without any additional resource overhead is an integral part of middleware used in real-time, event-driven and performance-critical environments. The Reactor [19] allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients. The Reactor pattern inverts the flow of control in a system during event handling.

The Reactor model for a POSAML model of our sample client-server application is shown in Figure 5, where the provisioner configures the following participants:

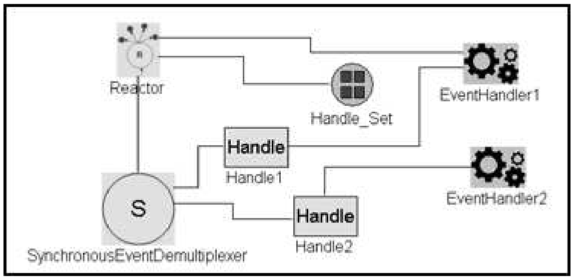

Figure 5. Model of the Reactor Pattern

1. *Handle*: The handle uniquely identifies event sources such as network connections or open files. Whenever an event is generated by an event source, it is queued up on the handle for that source and marked as "ready."
2. *Reactor*: The reactor is the dispatching mechanism of the Reactor pattern. In response to an event, it dispatches the corresponding event handler for that event.
3. *Event Handler*: The event handlers are the entities which actually process the event. These are registered with the reactor and are dispatched by the reactor when the event for which they are registered occurs.
4. *Handle Set*: The registered handles form a set called the "Handle Set."
5. *Synchronous Event Demultiplexer*: This entity is actually implemented as a function call, such as `select()` or `WaitForMultipleObjects()` (in case of Windows-based systems). It waits for one or more indication events to occur, and then propagates these events to the reactor.
6. *Concrete Event Handlers*: The concrete event handlers specialize the generalized Event Handler. They are responsible for processing specific types of events, such as input data or timeouts.

In order to minimize the risk of choosing incorrect and incompatible features, various constraints are specified within the POSAML metamodel using both OCL, which checks constraints at modeling time, and interpreters, which check constraint violations when the generative tools are used. Constraint checking within the POSAML metamodel includes cardinality and relationship constraints. For example, a reactor can be connected to one and only one synchronous event demultiplexer. These constraints ensure that the modeler does not build an incorrect model thereby ensuring that systems conform to the semantics of the pattern languages.

Step 2(b): Modeling the Acceptor-Connector Pattern

A system engineer can model the Acceptor-Connector pattern in POSAML for the sample application as shown in Figure 6. Various constraints minimize the risk of choosing a wrong combination of elements in the pattern. Only the correct combinations of connections and features are allowed to be added for a particular pattern. For example, only the "End Point" feature can be added to the Acceptor-Connector pattern. The middleware provisioner models the following participants of the Acceptor-Connector pattern:

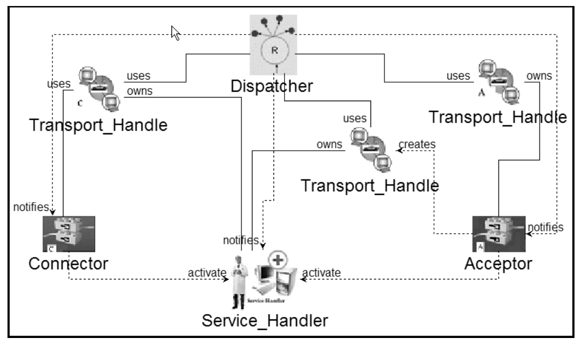

Figure 6. Model of Acceptor-Connector

1. *Acceptor*: The Acceptor is a factory that implements a passive strategy to establish a connection and initialize the associated Service Handler. It creates a passive mode end point transport handle that has necessary end points needed by the Service Handlers.
2. *Connector*: A Connector is a factory that implements the active strategy to establish a connection and initialize the associated Service Handler. It initiates the connection with a remote Acceptor and has synchronous mode (using the Reactor pattern) and asynchronous mode (using the Proactor pattern) connections.
3. *Dispatcher*: The Dispatcher manages registered Event Handlers. In case of the Acceptor, the Dispatcher demultiplexes connection indication events received on transport handles. Multiple Acceptors can be registered within the Dispatcher. For Connector, the Dispatcher demultiplexes completion events that arrive in response to connections.
4. *Service Handler*: A Service Handler is an abstract class that is inherited from Event Handler. It implements an application service playing the client role, server role or both roles. It provides a hook method that is called by an Acceptor or Connector to activate

the application service when the connection is established.

5. *Transport End points*: These represent a factory that listens for connection requests to arrive, accepts those connection requests and creates transport handles that encapsulate the newly connected transport end points. By using these end points data can be exchanged by reading or writing to their associated transport handles. A transport handle encapsulates a transport end point.

Step 3: Modeling the Features of Pattern Participants

A systems developer uses the feature aspect of POSAML as a visual tool to select different pattern-specific features of the middleware. Figure 7 illustrates an instance of a feature model. The system engineer can model zero or more features using this tool. If features are not selected from the model, default values for these features will be selected.

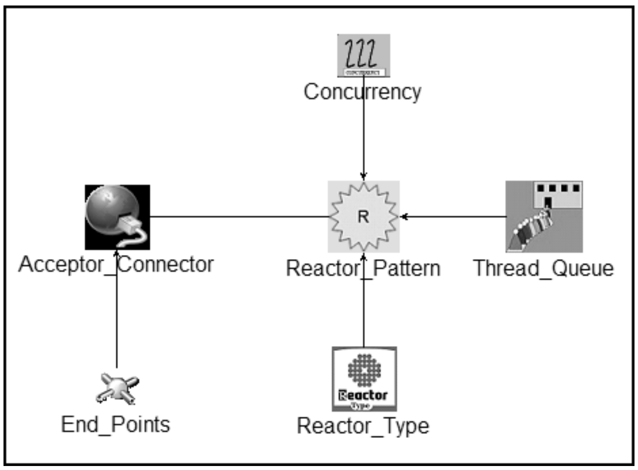

Figure 7. POSAML Model: Feature View

1. **Concurrency:** This feature is important for different middleware to manage concurrency and allow long running operations to execute simultaneously without impeding the progress of other operations. Server concurrency strategies found in contemporary middleware solutions, such as the TAO CORBA middleware [18], support different types of concurrency strategies, including *reactive* and *thread per connection*.
2. **Reactor Type:** This feature is used to specify the kind of reactor used by the system. For example, depending on the concurrency strategy chosen, the reactor could be single-threaded or multi-threaded. Different strategies can be plugged within the reactor for event demultiplexing. This depends on whether the reactor is used to demultiplex network events or GUI events.
3. **Thread Queue:** In the case of concurrent request handling by a reactor, different strategies can be selected for handling queued events (e.g., FIFO or LIFO).
4. **End points:** This feature applies to the acceptor-connector patterns, which instructs the system of the listening end points for the server role. The range of available end points in POSAML include listening ports (e.g., TCP port number), host IP addresses or canonical names, and the protocol used (e.g., TCP, UDP, Shared Memory or other custom transports).

Step 4: Modeling the QoS Validation

A sample model that can be constructed using POSAML is shown in Figure 8. In this case the system provisioner has modeled two patterns, the Reactor and the Acceptor-Connector, as well as the benchmarking characteristics to analyze the performance of the Reactor pattern. The latency and throughput metrics are shown attached to the Reactor pattern. In addition, the model also specifies the number of client threads and the service time for each event handler in the Reactor Pattern.

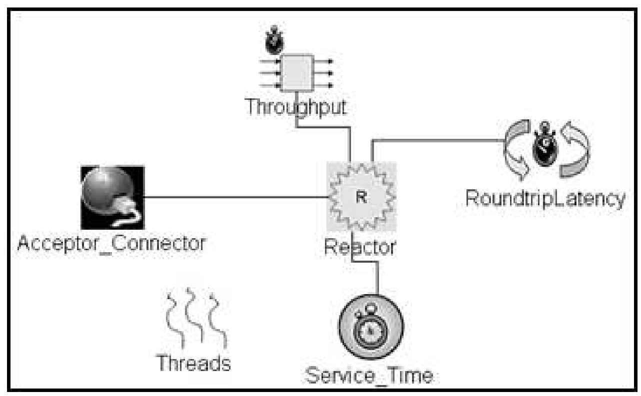

Figure 8. Benchmarking Aspect

Step 5: Using the Generative Capabilities

The generative capabilities in POSAML use the models to synthesize artifacts for concrete middleware platforms. Different services can be configured using this interpreter. For example, it could include some service settings that are used to control the creation of configurable resources used by an object broker. Other settings are used to control the behavior of client and server including the concurrency strategy, demultiplexing strategies, request multiplexing strategies, wait strategies, and connection strategies. We used POSAML to create a configuration file and script file for a CORBA middleware called TAO. A snippet of the generated service configuration file and benchmarking artifacts are shown below.

```
static Advanced_Resource_Factory
   "-ORBReactorType tp -ORBReactorThreadQueue LIFO"
static Server_Strategy_Factory
   "-ORBConcurrency reactive"

- <benchmark_inputs>
    <connections>10</connections>
    <data>ABCDEF</data>
    <data_exchanges>200</data_exchanges>
  - <reactor_inputs>
      <reactor_type>wfmo</reactor_type>
      <handlers>2</handlers>
      <service_time>Uniform</service_time>
    </reactor_inputs>
  </benchmark_inputs>
```

The POSAML model interpreter also generates a script file shown below that contains inputs to run any application (e.g., the Naming service or benchmarking evaluation tool) with proper end points (e.g., listening ports, protocol and host name). These end points are also used by the Acceptor-Connector pattern for different transport handles. For example, in the snippet below, the benchmarking test is run on an endpoint that uses the IIOP protocol used in CORBA.

```
benchmark_test -ORBEndpoint iiop://127.0.0.1:9000
```

4 Related Work

With the growing complexity of component-based systems, composing system-level performance and dependability attributes using component attributes and system architecture is gaining attention. Crnkovic *et al.* [3] classify the quality attributes according to the possibility of predicting the attributes of the compositions based on the attributes of the components and the influence of other factors within the architecture and the environment. However, they do not propose any methods for composing the system-level attributes.

At the model and program transformation level, the work by Shen and Petriu [21] investigated the use of aspect-oriented modeling techniques to address performance concerns that are weaved into a primary UML model of functional behavior. It has been observed that an improved separation of the performance description from the core behavior enables various design alternatives to be considered more readily (i.e., after separation, a specific performance concern can be represented as a variability measure that can be modified to examine the overall systemic effect). The performance concerns are specified in the UML profile for Schedulability, Performance, and Time (SPT) with underlying analysis performed by a Layered Queuing Network (LQN) solver.

A disadvantage of the approach is that UML forces a specific modeling language. The SPT profile also forces performance concerns to be specified in a manner than limits the ability to be tailored to a specific performance analysis methodology. As an alternative, domain-specific modeling supports the ability to provide a model engineer with a notation that fits the domain of interest, which improves the level of abstraction of the performance modeling process.

There have been efforts to evaluate the performance of middleware patterns analytically by various researchers [16, 23]. A drawback of using analytical models is that it is difficult to predict the behavior of a complex system based on analytical methods alone. Harkema, *et al* [8] have worked on the performance evaluation of the CORBA method invocation and threading models. However, they have not focused on the pattern-based approach towards performance analysis of middleware. Model-driven techniques are increasingly being used for middleware development, but converting static pattern-based middleware models into simulation or empirical models for the purpose of performance evaluation has not yet been a focus in the research community.

There are various middleware specialization techniques described in the literature, which can be leveraged to customize middleware. For example, Feature-Oriented Programming (FOP) [5] is an appropriate technique to design and implement program families, which uses incremental and stepwise refinement approaches [2]. FOP aims to cope with the increasing complexity and lack of reusability and customizability of contemporary software systems. Aspect-Oriented Programming (AOP) [11] is another related programming paradigm and has similar goals: It focuses primarily on separating and encapsulating crosscutting concerns to increase maintainability, understandability, and customizability.

5 Conclusions

Distributed systems implemented with standardized middleware present several challenges with respect to the accidental complexities associated with provisioning (i.e., configuration and QoS validation). In current practice, provisioning of middleware are performed through low-level, non intuitive and non reusable means. The manual nature of these techniques are error prone and tedious, and prohibit a system provisioner from rapidly exploring various design alternatives. To address these challenges, this paper presented POSAML, which is a visual modeling language that addresses the provisioning problem at a higher-level of abstraction.

From our experience, we have found that POSAML allows various provisioning scenarios to be explored in a rapid manner that is middleware-independent. The concerns that are separated among the various aspects in POSAML provide an ability to evolve the configuration in a manner that isolates the effect to a single design change. When a choice is made for a pattern, POSAML removes all of the inconsistent choices among other patterns. This allows the provisioner to work with a narrowed search space and ignore all incompatible configurations. Furthermore, model interpreters associated with POSAML assist in generating the artifacts needed to perform QoS validation.

We have applied POSAML to model several case studies implemented in the ACE/TAO middleware. Although our experience in using POSAML to configure and provision these case studies has been positive, there are still a few limitations that remain. For example, our generative techniques are applied only for the TAO middleware. We are addressing these limitations as part of our planned future work.

POSAML is part of the CoSMIC tool suite and is available for download from `www.dre.vanderbilt.edu/cosmic`.

Acknowledgments

This research was supported by the following grants from the National Science Foundation (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] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, NY, 1977.

[2] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Stepwise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[3] I. Crnkovic, M. Larsson, and O. Preiss. *Book on Architecting Dependable Systems III, R. de Lemos (Eds.)*, chapter "Concerning predictability in dependable component-based systems: Classification of quality attributes", pages 257–278. Springer-Verlag, 2005.

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

[5] Don Batory. Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming. *IBM Systems Journal*, 45(3), 2006.

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

[7] H. Giese, I. H. Kruger, and K. M. L. Cooper. Workshop on Visual Modeling for Software Intensive Systems. *Procedings of 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, page 4, 2005.

[8] Harkema, M. and Gijsen, B. M. M. and van der Mei, R. D. and Hoekstra, Y. Middleware Performance: A Quantitative Modeling Approach. In *International Symposium on Performance Evaluation of Computer and Communication Systems (SPECTS)*, 2004.

[9] Jeff Gray and Juha-Pekka Tolvanen and Steven Kelly and Aniruddha Gokhale and Sandeep Neema and Jonathan Sprinkle. *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*, chapter Domain-Specific Modeling. CRC Press, Boca Raton, Florida, 2006.

[10] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[12] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[13] Microsoft Corporation. Microsoft .NET Development. `msdn.microsoft.com/net/`, 2002.

[14] Object Management Group. *CORBA Components*, OMG Document formal/2002-06-65 edition, June 2002.

[15] P. Tarr and H. Ossher and W. Harrison and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119, May 1999.

[16] S. Ramani, K. S. Trivedi, and B. Dasarathy. Performance analysis of the CORBA event service using stochastic reward nets. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 238–247, October 2000.

[17] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), 2006.

[18] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[19] 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.

[20] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[21] H. Shen and D. C. Petriu. Performance analysis of uml models using aspect-oriented modeling techniques. In *Proc. of Model Driven Engineering Languages and Systems (MoDELS 2005), Springer LNCS 3713*, pages 156–170, Montego Bay, Jamaica, October 2005.

[22] Sun Microsystems. Java^{TM} 2 Platform Enterprise Edition. java.sun.com/j2ee/index.html, 2001.

[23] Swapna Gokhale and Aniruddha Gokhale and Jeff Gray. A Model-Driven Performance Analysis Framework for Distributed, Performance-Sensitive Software Systems. In *Proceedings of the NSF NGS Workshop, International Conference on Parallel and Distributed Processing Symposium (IPDPS) 2005*, Denver, CO, April 2005.