POSAML: A Visual Modeling Language for Middleware Provisioning ⋆

Aniruddha Gokhale [a,*] Dimple Kaul [a] Arundhati Kogekar [a]
Jeff Gray [b] Swapna Gokhale [c]

[a] *Dept of EECS, Vanderbilt University, Nashville, TN, USA*
[b] *Dept of CIS, University of Alabama at Birmingham, Birmingham, AL, USA*
[c] *Dept of CSE, University of Connecticut, Storrs, CT, USA*

Abstract

Next generation distributed applications are often hosted on heterogeneous platforms including different kinds of middleware. Due to the applications' growing functional complexity and their multiple quality of service (QoS) requirements, system developers are increasingly facing a substantial number of middleware provisioning challenges, which include configuring, optimizing and validating the middleware platforms for QoS properties. Traditional techniques for middleware provisioning tend to use non-intuitive, low-level and technology-specific approaches, which are tedious, error prone, and non-reusable across different technologies. Quite often the middleware provisioning activities are carried out by different actors without much interaction among them, which results in an iterative trial-and-error process to provisioning. Higher level abstractions, particularly those that use visual models, are effective in addressing these challenges. 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 configurations and optimizations while enabling automated middleware-specific validation of system QoS properties.

Key words: Model-driven Engineering, Visual Domain-specific Modeling Languages, Generative Tools

⋆ This work was supported by a collaborative NSF CSR-SMA grant
* Contact Author: a.gokhale@vanderbilt.edu

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 [1], .NET [2] and CORBA Component Model (CCM) [3]) coupled with advances in integration technologies (e.g., web services and XML-based standards) enable the construction of application functionality by connecting individual services that are deployed across distributed resources.

Several non-functional or systemic concerns must be addressed simultaneously when provisioning (i.e., configuring, optimizing and validating) complex distributed applications on these middleware platforms. The non-functional provisioning concerns comprise the problem of choosing the correct set of configuration and composition parameters of the middleware platforms, optimizing the platforms for the QoS needs, and validating that these decisions meet the QoS requirements of the applications. Traditional approaches to middleware provisioning typically use low-level, non-intuitive, and technology-specific mechanisms, which often end up being fixed-point solutions that are not reusable across different applications and multiple middleware technologies.

From our experience collaborating with industry practitioners [4], 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 or optimization phase. Moreover, the validation phase uses processes that do not leverage decisions made at the configuration phase, which limits the 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 and its quality.

A solution to address this problem is to provide a mechanism that raises the level of abstraction at which system provisioners can provision middleware. Visual aids are one of the best known techniques to intuitively reason about any system [5]. 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 also hold promise for middleware provisioning.

A desirable visual tool for the provisioning problem is one that can provide clean separation of concerns [6] between the configuration, optimization and QoS validation phases, yet unify the three phases by allowing the intent and decisions in one phase to seamlessly transfer to the next one where they can be used for automation and optimization of the phase under consideration. Having 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) [7,8], and an associated set of generative programming tools [9] 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 [10,11]. 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 for different application contexts.

The rest of the paper is organized as follows: Section 2 compares our work to existing research; Section 3 introduces the challenges in designing a visual framework for middleware provisioning; Section 4 describes the design of the POSAML visual framework for middleware provisioning; Section 5 illustrates the steps to use POSAML in middleware provisioning; and Section 6 concludes with a description of lessons learned and future work.

2 Related Work

In this section we survey related research that addresses one or more phases of the provisioning problem, and uses intuitive notations including visual languages to address these challenges.

At the model and program transformation level, the work by Shen and Petriu [12] investigated the use of aspect-oriented modeling techniques to address performance concerns that are woven 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 [12].

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 that 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 [13,14]. 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* [15] 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.

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.* [16] 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.

There are various middleware specialization techniques described in the literature, which can be leveraged to customize middleware. For example, Feature-Oriented Programming (FOP) [17] is an appropriate technique to design and implement program families, which uses incremental and stepwise refinement approaches [18]. FOP aims to cope with the increasing complexity and lack of reusability and customizability of contemporary software systems. Aspect-Oriented Programming [19] is another related programming paradigm and has similar goals: It focuses primarily on separating and encapsulating crosscutting concerns to increase maintainability, understandability, and customizability.

3 Designing Visual Tools for Middleware Provisioning

Middleware provisioning is the activity that comprises the configuration, optimization and validation of the middleware platform for QoS properties of the hosted applications. We are interested in the systemic (i.e., non-functional) properties of applications that are assured by the middleware provisioning process. 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 in middleware provisioning, 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).

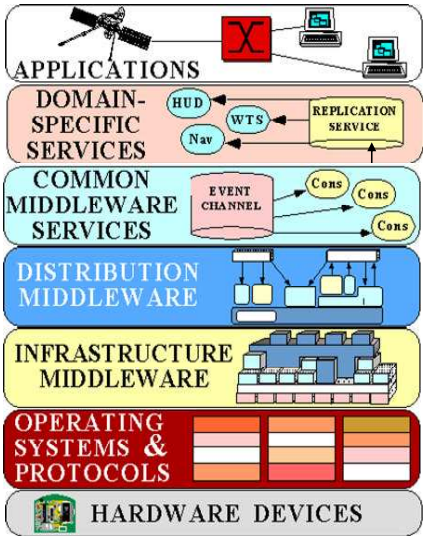

Fig. 1. Middleware Structure

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 configured individually 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 [10] that form the building blocks of the different layers of middleware. Section 4.1 describes how we leverage and formalize these insights to design and

implement the POSAML visual tool for middleware provisioning.

• **Criterion 2: Separation of concerns:** Each phase of middleware provisioning is filled with numerous accidental complexities. For example, the configuration phase requires the actors to make the right decisions on the choice of configuration options to select. When confronted with multiple different middleware technologies, the actor is required to have detailed understanding of the flexibility and configurability of the middleware stack. Similarly, optimizations require different kinds of fine tuning and are driven by the QoS needs of the application. Once again we observe that different middleware technologies use different data models and messaging standards which require different kinds of optimizations. The QoS validation phase involves analytical and empirical evaluation of QoS properties. It requires the systems provisioners to construct appropriate application testing and middleware benchmarking code in accordance to the configuration decisions. Many different techniques exist for such evaluations and different middleware stacks will need potentially different methods to validate QoS. Each of these phases thus incurs substantial accidental complexity as described, which is further compounded by the fact that they are all driven by the same QoS requirements of the application and decisions at one phase impact the other. There is a need to disentangle the phases at an intuitive level of abstraction that addresses the challenges incurred by the accidental complexities in each phase. Section 4.2 describes how POSAML provides these separation of concerns at the user interface level using visual notations of middleware abstractions and their features.

• **Criterion 3: Need for a unified framework:** Although traditional approaches to middleware provisioning decouple the configuration, optimization and validation phases, such a decoupling is not useful since each subsequent phase tends not to keep any knowledge of the earlier decisions. Moreover, these activities are typically carried out by a different set of actors. Although decoupling is necessary to separately address the challenges in each phase, it is necessary that the intent of the actors and their decisions be made available to all phases. As noted earlier, however, middleware provisioning needs to be guided by the application QoS needs. This requires that the three stages of provisioning be performed in a unified manner (although visually separated) such that decisions made in the configuration phase are leveraged in the optimization phase, and the decisions of these two phases are leveraged in the QoS validation phase. Thus, it is important that the separation of concerns be restricted to an intuitive level of abstraction, such as at the level of visual perception to the actors. This requirement adds a new dimension of variability to the challenges described in Criterion 1. Moreover, it requires that the separation of concerns called for in Criterion 2 remain at the visual abstraction level so that the process is more intuitive but remains unified internally. To address these criteria, the visual mechanisms should provide a unified framework that can address the configuration, optimization and QoS validation concerns that arise in middleware provisioning. Such a framework must provide the means to transfer the intent and decisions at each stage to the next in the entire provisioning lifecycle. Section 4.3

describes how POSAML provides intuitive abstractions of the middleware stacks to the system integrators, which eases the task of provisioning.

4 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 3.

4.1 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 [10,11]. 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) [11]. 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 [11] and pattern languages [20] enable us to identify and resolve the variability in middleware provisioning.

4.1.1 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 la-

tency and timeliness requirements for real-time systems, or throughput for other enterprise systems (e.g., telecommunications call processing). We refer to this dimension of provisioning variability as *middleware compositional variability*.

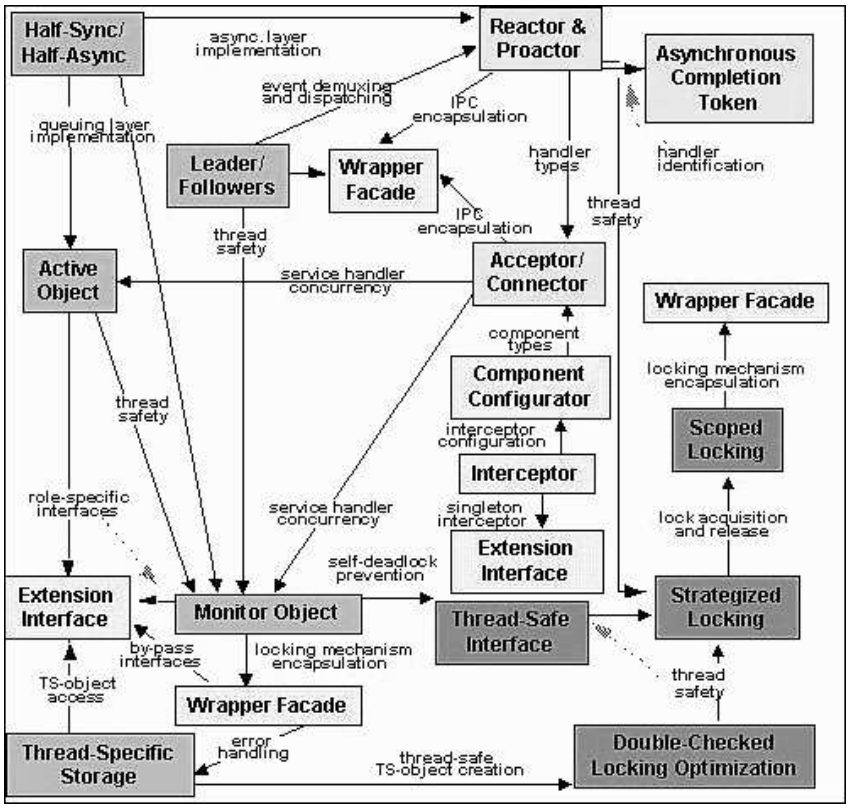

Fig. 2. Middleware Patterns and Pattern Languages

Figure 2 illustrates a family of interacting patterns that form a pattern language [20] for middleware designed to support such applications. The middleware can be customized by composing compatible patterns-based building blocks. For example, event demultiplexing and dispatching via the Reactor or Proactor patterns [11] can be composed with the concurrent event handling provided by the Leader-Follower or Active Object patterns [11]. However, an Asynchronous Completion 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.

4.1.2 Identifying Variability in Building Block Configurations

Middleware provisioners 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 in the previous section – 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.

4.2 Meeting Criterion 2: Separation of Concerns

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 3 described the need for separation of concerns in the provisioning stages. Model-based solutions based on visual aids can provide the desired solution. Model-driven Engineering (MDE) [7,21], 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, optimizing and validating large systems hosted on middleware platforms.

POSAML enables the modeling of middleware stacks and their configurations, optimizations and validation phases 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.

4.2.1 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) [22]. 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" ele-

ment 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 from Section 3. The same GME environment can be used by provisioners to model the provisioning decisions using the POSAML metamodel.

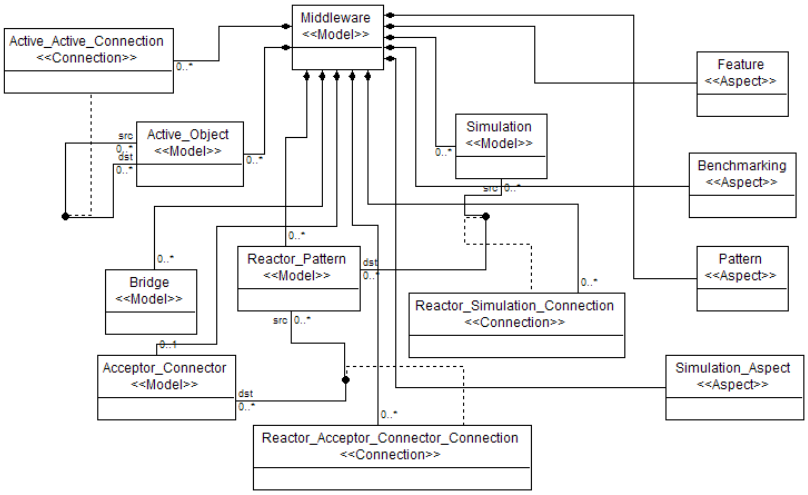

Fig. 3. Top-level Metamodel of Middleware Structure

The metamodel illustrated in Figure 3 provides 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 metamodel also comprises other elements (shown as GME aspects), such as the feature modeling capability that enables configuring and optimizing each block, simulation modeling capability that enables validation via simulation, and benchmarking modeling capability that enables validation via empirical benchmarking. By providing each of these capabilities as a separate first-class element of the metamodel, POSAML separates the concerns of modeling the pattern, its configuration, optimizations and their compositions, and system QoS validation by virtue of using GME aspects. We describe these visual separation of concerns below.

4.2.2 Enabling Separation of Concerns

We now describe how POSAML provides visual separation of concerns that are useful for the different provisioning phases.

(a) Pattern Modeling in POSAML

Figure 3 showed how the high-level POSAML metamodel enables a system architect to design a composition of patterns. Due to the hierarchical nature of POSAML, each pattern itself has a metamodel associated with it. This metamodel enables that particular pattern to be modeled in the "Pattern" GME Aspect. This provides visual mechanisms to concentrate on individual building blocks of the middleware and

their configurations and optimizations. We illustrate this capability with an example metamodel of the Reactor [11] building block.

The ability to handle and dispatch simultaneously occurring events effectively without any additional resource overhead is an integral part of systems designed for use in real-time, event-driven and performance-critical environments. The Reactor 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. Figure 4 illustrates the metamodel of the Reactor building block in POSAML.

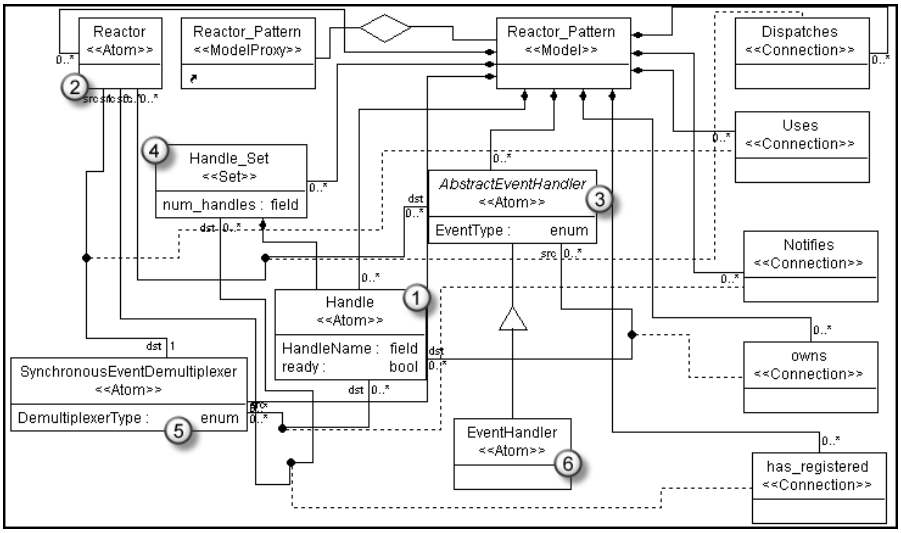

Fig. 4. Meta-model of the Reactor Pattern

This metamodel enables the designer to model the following participants and their relationships of a Reactor building block:

(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) *Abstract 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 model interpreters 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.

(b) Feature Modeling in POSAML

A Feature model [9] 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 choice of network transport, listening end-points, concurrency requirements, and periodicity of requests, all represented as higher-level visual 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.

Feature modeling paves the way to select the desired configuration and customization parameters of a middleware building block and the compositions of these building blocks. The selected features provide the means for optimization tools to specialize the middleware code so that the hosting platform is fine tuned to support application QoS.

Developing a feature model out of a metamodel involves defining valid entity and relationships in the schema. All these features are pattern-specific. For example, Concurrency, Thread Queue, and Reactor Type are Reactor-specific; Active Object map size is Active Object pattern-specific; and End-points are Acceptor and Connector pattern-specific.

(c) QoS Validation Modeling in POSAML

To enable the performance analysis of a configured and customized system, the modeling language provides a method to model simulation and benchmarking char-

acteristics. The POSAML metamodel enables the systems provisioner to model the workload (e.g., number of threads, data, and the number of data exchanges), as well as which metric to measure (e.g., 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 later. This file can be used by an existing benchmarking library within the middleware which is being benchmarked.

4.3 Meeting Criterion 3: Unified Framework

We discussed earlier how the middleware configuration is accomplished through POSAML's feature modeling [9,17] capability, which assists a system provisioners in configuring a variety of different middleware features (e.g., choosing the pattern and its configuration parameters). However, for other visually modeled concerns (e.g., benchmarking) to leverage the modeled system in the feature aspect requires unifying the different concerns internally. For example, the benchmarking capability (though visually decoupled from the feature modeling 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.

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 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 capabilities of the GME environment, different middleware-specific 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 the specific instance of middleware. One of the generated artifacts is a configurator file and the other 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) Optimizing Interpreter: Figure 4.3 shows an example of how we can model a building block with the different specializations. In this example we have mod-

eled the abstract base class `ACE_Reactor_Impl` with its concrete implementation of subclasses like `ACE_Select_Reactor`, `ACE_WFMO_Reactor` and `ACE_TP_Reactor`. It also shows two modeled specialization aspects named as `Single_Thread_Aspect` and `TP_Thread_Aspect`. `Single_Thread_Aspect` specialization is for single threaded implementation of Reactor and `TP_Thread_Aspect` specialization is for threadpool implementation of Reactor.

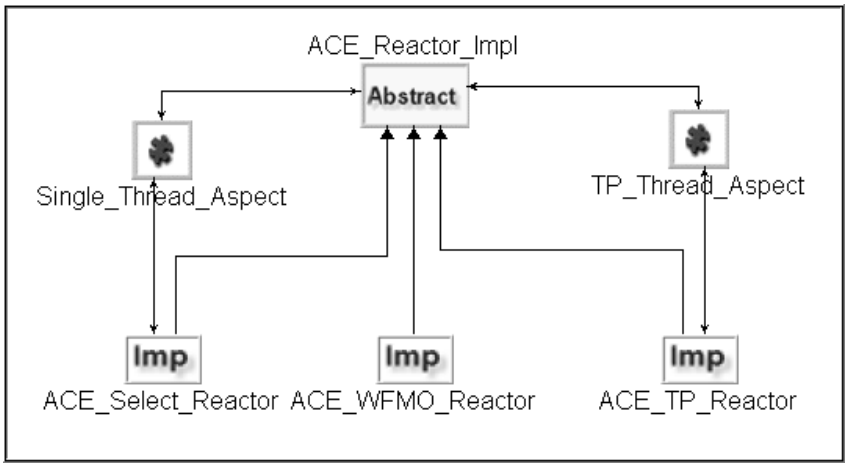

Fig. 5. Modeling of Build Block Specializations in POSAML

The optimizing interpreter consults the feature model for the configuration parameters deducing the type of the building block being selected, such as the reactor type. Using the information from the specialization model, the optimization interpreter generates metadata used by aspect-oriented tools, like AspectJ [23] and AspectC++ [24] so that the middleware code can be optimized based on the selected features of a building block.

(c) 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). The parameters are generated in XML format and can be used to parametrize an existing benchmarking library.

5 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 6 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 demonstrates 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.

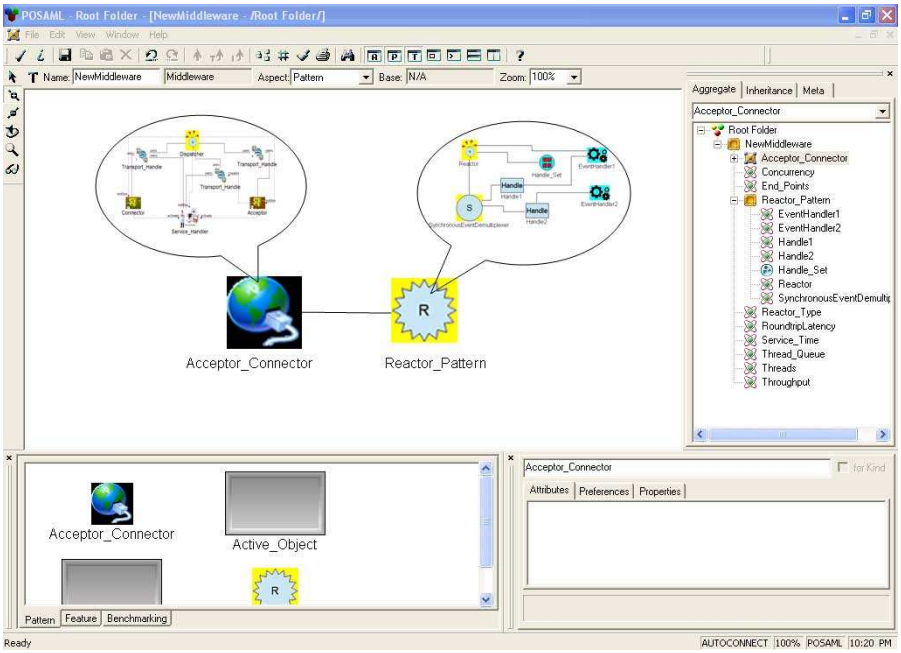

Fig. 6. Middleware Provisioning in POSAML

Step 2 (a): Modeling the Reactor Pattern:

The metamodel for the Reactor pattern has been illustrated in Figure 4. Figure 7 shows the use of this metamodel in modeling the Reactor for a POSAML model of our sample client-server application. The various participants of the Reactor pattern have been described in Section 4.2.

In Figure 7, the provisioner has modeled two event handlers corresponding to a handle set consisting of two handles. Both event handlers are connected to the Reactor, which indicates that both of them are ready to handle events of the appropriate type. The handles are connected to the synchronous event demultiplexer, which indicates that both the handles are active and ready to accept events of the corresponding type. It is also possible to model different types of reactor (e.g.,

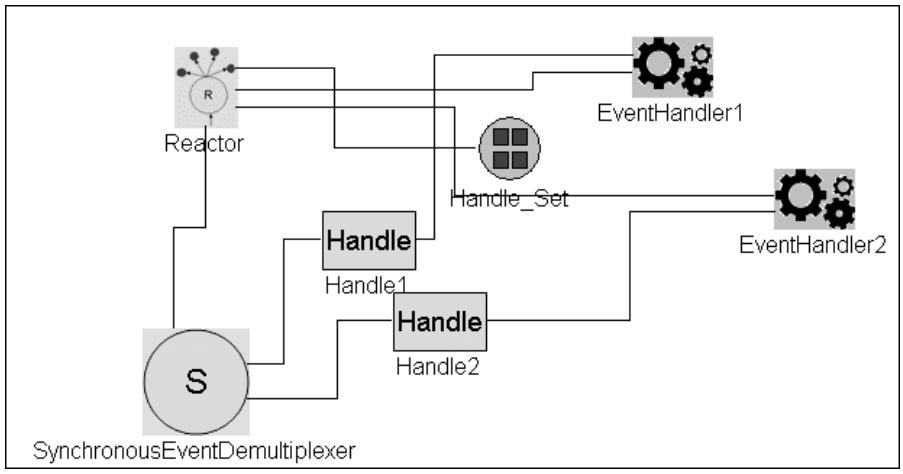

Fig. 7. Model of the Reactor Pattern

single-threaded/multi-threaded).

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 8. 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:

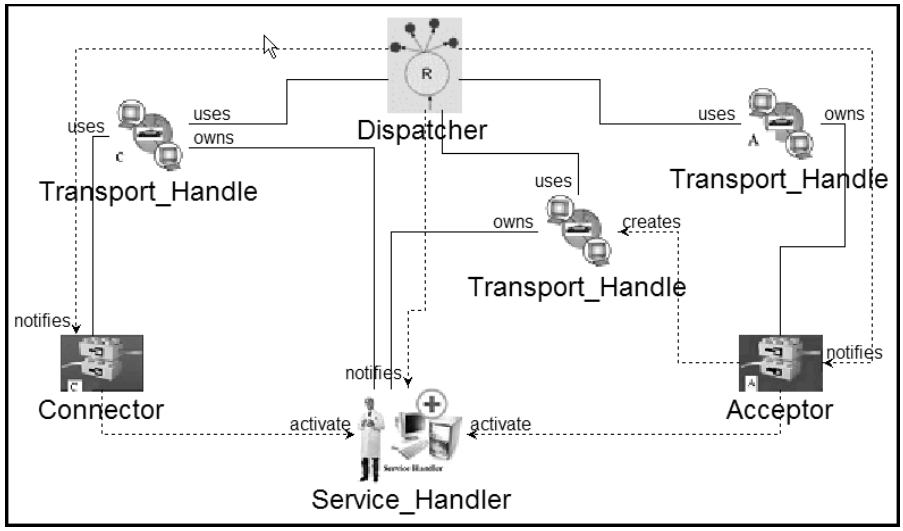

Fig. 8. Model of Acceptor-Connector

a. *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.
b. *Connector*: A Connector is a factory that implements the active strategy to es-

tablish 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.

c. *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.

d. *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.

e. *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 system provisioner uses the feature aspect of POSAML as a visual tool to select different pattern-specific features of the middleware. Figure 9 illustrates an instance of a feature model. The system provisioner 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. In order to minimize the risk of choosing wrong connections and options, various constraints are embedded in the POSAML metamodel. Some of these constraints are checked using OCL, i.e., checking for non-null references or proxies, and some of them are checked at the time of execution of interpreters, i.e., check if a feature is connected to the correct pattern or not. The selected features set with different options are exported into files using the configuration interpreter and are used to configure a middleware system.

Step 4: Modeling the QoS Validation: A sample model that can be constructed using POSAML is shown in Figure 10. 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.

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

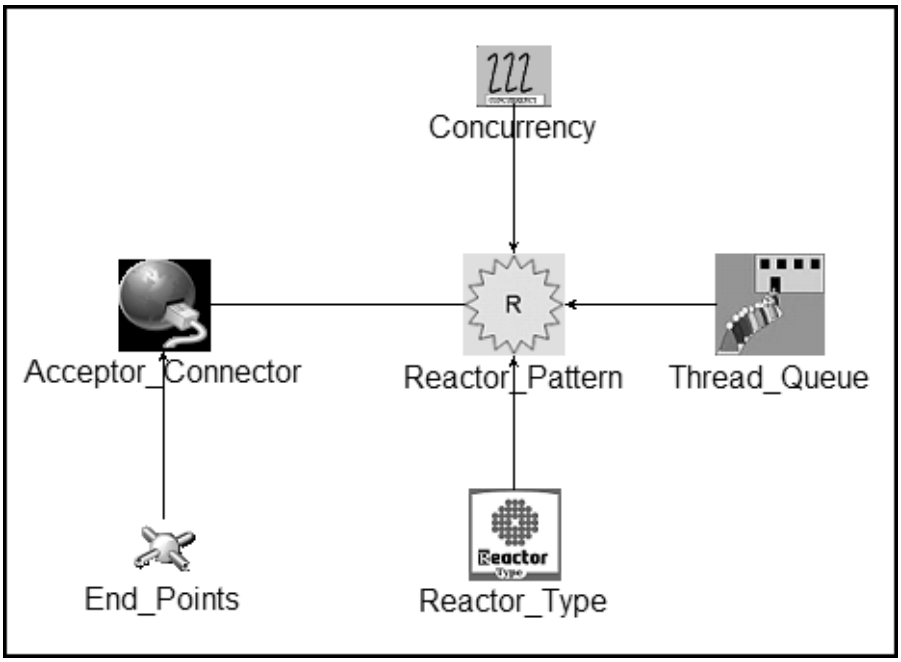

Fig. 9. POSAML Model: Feature View

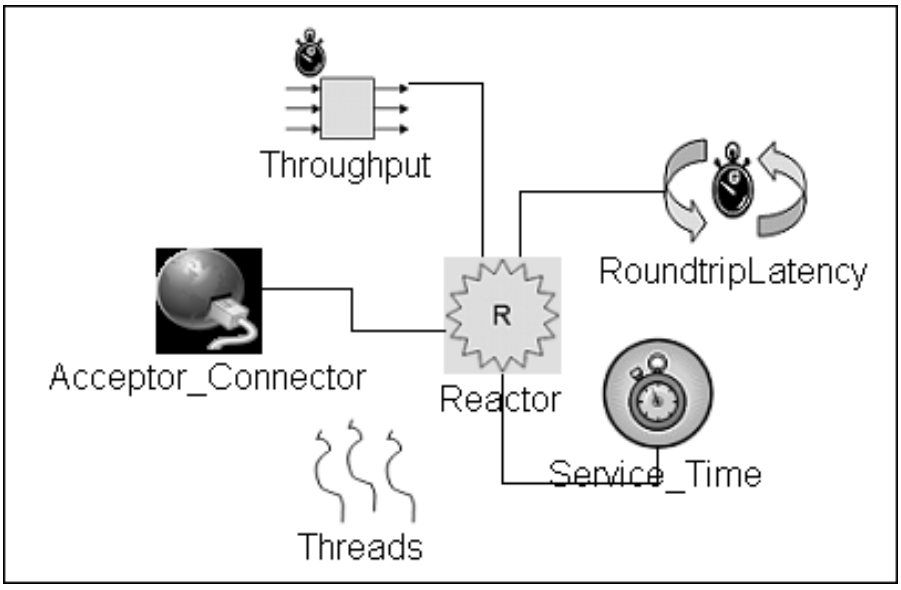

Fig. 10. Benchmarking Aspect

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 [25]. 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 from CORBA.

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

In the following code snippet we illustrate optimization metadata generated by the optimizer interpreter for the `Select_Reactor` implementation.

```
/**
  * Aspect for Single Threaded specialization
  */
aspect Single_Thread_Aspect
{
  /**
    * It redirects purge_pending_notifications
    * method of ACE_Reactor_Impl to same method
    * of ACE_Select_Reactor subclass.
    */

  advice call ("% ACE_Reactor_Impl
    ::purge_pending_notifications(...)"):around ()
  {
    ((ACE_Select_Reactor_Impl *) tjp->target ())->
      ACE_Select_Reactor_Impl
      ::purge_pending_notifications
       (*tjp->arg < 0 >(),*tjp->arg < 1 >());
  }
```

```
}
```

In the code snippet, the generated metadata for AspectC++ [26,24] redirects the method `purge_pending_notification` of the abstract class to the same method name of the concrete implementation directly, thereby removing the virtual table lookup overhead between abstract and base classes.

6 Conclusions

Distributed systems implemented with standardized middleware incur several accidental complexities associated with middleware provisioning (i.e., configuring, optimizing and validating for QoS properties). 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] Sun Microsystems, Java^{TM} 2 Platform Enterprise Edition, java.sun.com/j2ee/index.html (2001).

[2] Microsoft Corporation, Microsoft .NET Development, `msdn.microsoft.com/net/` (2002).

[3] Object Management Group, CORBA Components, OMG Document formal/2002-06-65 Edition (Jun. 2002).

[4] D. C. Sharp, Reducing Avionics Software Cost Through Component Based Product Line Development, in: Proceedings of the 10th Annual Software Technology Conference, 1998.

[5] H. Giese, I. H. Kruger, 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) (2005) 4.

[6] 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, 1999, pp. 107–119.

[7] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-Integrated Development of Embedded Software, Proceedings of the IEEE 91 (1) (2003) 145–164.

[8] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, J. Sprinkle, Domain-Specific Modeling, in: CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.), CRC Press, 2007.

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

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

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

[12] H. Shen, 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, Montego Bay, Jamaica, 2005, pp. 156–170.

[13] S. Gokhale, A. Gokhale, J. 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, 2005.

[14] S. Ramani, K. S. Trivedi, B. Dasarathy, Performance analysis of the CORBA event service using stochastic reward nets, in: Proc. of the 19th IEEE Symposium on Reliable Distributed Systems, 2000, pp. 238–247.

[15] 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.

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

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

[18] D. Batory, J. Sarvela, A. Rauschmayer, Scaling Step-wise Refinement, IEEE Transactions on Software Engineering 30 (6) (2004) 355–371.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming, 1997, pp. 220–242.

[20] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, A Pattern Language, Oxford University Press, New York, NY, 1977.

[21] D. C. Schmidt, Model-Driven Engineering, IEEE Computer 39 (2) (2006) 25–31.

[22] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer (2001) 44–51.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355.
URL `citeseer.nj.nec.com/kiczales01overview.html`

[24] O. Spinczyk, D. Lohmann, Aspect-Oriented Programming with C++ and AspectC++, in: Tutorial at Aspect Oriented Software Development (AOSD), 2005.

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

[26] Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat, AspectC++: An Aspect-Oriented Extension to C++, in: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), 2002.