Managing Middleware Variability Using Visual Modeling Languages ⋆

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

[a] *ISIS, Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Campus Box 1829 Station B, Nashville, TN 37235, USA*

[b] *University of Alabama at Birmingham, Birmingham, AL, USA*

[c] *University of Connecticut, Storrs, CT, USA*

Abstract

Advances in hardware and networks are pushing the envelope on the capabilities provided by modern software systems. These are now increasingly being assembled as a loosely coupled collection of distributed, coordinating subsystems, where each subsystem provides a specific capability or service. What makes the realization of these systems hard is the significant variability in the systems software design space, which arises from having to support different performance attributes, such as latency and throughput, simultaneously in conjunction with other para-functional issues, such as reliability, availability and security. Yet another dimension of variability arises from the scalability challenges and the need to assemble and connect disparate systems on-the-fly while ensuring that the performance attributes of systems are met. System developers must address these sources of variability without overcomplicating their solutions, degrading their software quality, and exceeding their project schedules.

Separation of concerns (SoC), which is a key idea in the aspect-oriented and feature-oriented software development (AO/FOSD) processes, provides the means to deal with these variabilities. This paper describes how visual domain-specific modeling languages (DSMLs) can provide SoC capabilities whereby variability in systems software can be captured intuitively at higher levels of abstraction than that provided by third generation programming languages. For this paper we focus on addressing the variability in the middleware that hosts application components by describing a DSML that enables separating middleware concerns including the basic building blocks used for middleware composition, their configurations, and their performance characteristics.

Key words:
PACS:

1 Introduction

Advances in hardware and networking technologies, and availability of sophisticated tools for different stages of software development lifecycle are pushing the limits on the capabilities of the next generation of large-scale, distributed software systems. Recent trends in software design indicate that the use of prefabricated artifacts for software development is on the rise. A paradigm called *Software Factories* [11] is emerging that envisions the *industrialization* of software development, similar to industrialization in other domains, such as the housing, avionics and automotive industry. The prefabricated artifacts are the off-the-shelf (COTS) software components that one can acquire from different vendors and assemble them to deploy large-scale applications.

Component middleware technologies, such as J2EE [27], .NET [16] and CORBA Component Model (CCM) [18], coupled with advances in technologies, such as SOAP [28], WSDL [5] and XML [1], are increasingly being used to provision such applications since they enable the construction of application functionality dynamically by connecting individual component functionalities spread across distributed resources.

Several functional and non functional concerns must be addressed simultaneously when designing these applications. Software for such applications often must deal with significant *variability* caused by the need to address multiple simultaneous challenges, including: (1) integrating several individual subsystems using different technologies and platforms, (2) supporting a diverse set of *software attributes* including functional properties, such as the algorithms used to achieve the goals, and QoS properties, such as predictable latency/jitter, throughput guarantees, scalability, 24x7 availability, dependability, and security that must be satisfied simultaneously, (3) adaptation and constant fine tuning of the system required due to changing customer needs and operating conditions, and (4) ensuring that investments in these applications are safeguarded from technology obsolescence.

Aspect- and Feature-Oriented Design methodologies [3,8,15] have shown promise in terms of managing the complexity of large-scale software design through explicit *separation of concerns* (SoC) [19]. Most earlier work on SoC has been applied at the programming language level. In this paper, we focus on designing and implementing visual domain-specific modeling languages (DSMLs) [13] that enable the application of feature and aspect oriented principles to address the sources of variability in large-scale distributed service-oriented systems described above. In particular we focus on DSML design issues that illustrate the separation of concerns in the middleware that hosts application compo-

⋆ This work was supported by NSF CSR SMA grant
∗

nents. These concerns include the basic building blocks used for middleware composition, their configurations, and their performance characteristics. Our DSML is called POSAML (Patterns-oriented Software Architecture Modeling Language), which incorporates the patterns [25] for building network-centric software systems.

The rest of the paper is organized as follows: Section 2 describes the variability challenges in service oriented middleware and describes the need for visual modeling languages to address these challenges; Section 3 describes the pattern language modeling details of the POSAML language; Section 4 illustrates the separation of concerns capabilities of POSAML; Section 5 compares our work to existing research; and finally Section 6 describes lessons learned and future work.

2 Variability Challenges in Service Oriented Middleware

As noted earlier distributed service-oriented applications must be hosted on possibly heterogeneous middleware platforms in the networked environment. Moreover, each middleware stack on which the individual components of the distributed application are hosted may need to be fine tuned in different ways to cater to the several different quality of service (QoS) requirements of the overall application. These requirements are one of several sources of variability in the design of these large-scale applications. Others include the deployment decisions, resource allocations, workload characterizations among others. In this paper we focus on the variability in the middleware itself and how our solution addresses these challenges.

In the remainder of this section we describe the sources of variability in middleware and how visual domain-specific modeling languages provide the solutions to address these challenges.

2.1 Challenge 1: Middleware Composition Variability

When designing service-oriented systems, developers must decide the composition and configuration of middleware that hosts the application components. Middleware composition includes assembling individual building blocks of middleware that are compatible with each other to form middleware stacks that are suited to host application components. The building blocks of middleware are most often patterns-based. A software pattern [9] codifies recurring solutions to a particular problem occurring in different contexts, which is embodied as a reusable software building block. The choice of patterns-based block is driven by various factors, including the context in which the service-oriented system will be deployed, the concurrency and distribution requirements of the application, and other concerns, such as end-to-end latency and timeliness requirements for real-time service-oriented systems, or throughput for other service-oriented systems (such as telecommunications call processing). We refer to this incurred design space variability as the middleware compositional variability.

Figure 1 illustrates a family of interacting patterns forming a pattern language [2] for middleware designed to support service-oriented systems. When designing service-oriented applications, its developers will be required to decide the right set of patterns to use in the middleware compositions, such that the desired end to end QoS and functional requirements are met.

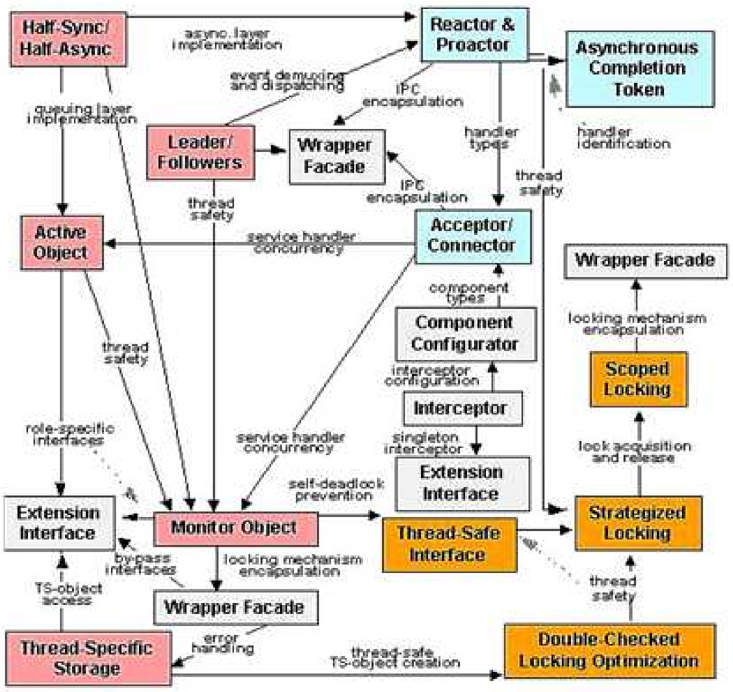

Fig. 1. Middleware Patterns and Pattern Languages

Patterns such as the *Active Object* (not shown) and *Leader-Follower* [25], provide alternate solutions to concurrency with each having its own advantages and disadvantages. For example, although the Active Object pattern is simple to implement, it incurs an additional performance penalty due to thread context switching and message queueing. Conversely, the Leader-Follower does not incur these drawbacks, but its implementation is more complicated to implement and analyze, so it may cause race conditions if not implemented properly.

Other patterns like the *Reactor* and *Proactor* [25] provide solutions to synchronous and asynchronous event demultiplexing and dispatching, respectively. Both of these patterns provide the building block of the asynchronous part of the *half-sync/half-async* pattern shown in Figure 1. Patterns, such as the *Asynchronous Completion Token* pattern can be used only in those cases where the underlying platform can support asynchronous messaging. Developers of service-oriented systems need to make the right choices of patterns to use in their solutions, such that the end-to-end QoS and functional requirements of the service-oriented systems are met.

The middleware can be customized by composing either the Reactor or Proactor pattern with the Leader-Follower or Active Object pattern using the *Strategy* [9] pattern. Similarly, a reactor could be composed with the *Connector* (client-side) or *Acceptor* (server-side) pattern. Developers of service-oriented systems need to make such choices of patterns and their configurations to use in their solutions, so that the end-to-end performance and functional requirements of the service-oriented systems are met, *i.e.*, they are responsible for handling the composition dimension of middleware variability.

2.2 Challenge 2: Configuration Variability

Middleware developers provide numerous configuration options to customize the behavior of individual building blocks, such as the Reactor and Acceptor. This flexibility incurs significant variability in design choices and hence affects the performance of building blocks. Since the impact is on a per building block basis – as opposed to a composition – we refer to this as the configuration dimension of variability.

Figure 2 shows an example of this type of variability involving an event demultiplexing and dispatching mechanism codified by the Reactor pattern [25]. In this pattern, a service-oriented application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening for incoming events. On occurrence of an event, the demultiplexer dispatches the event by making a callback to the right application-supplied event handler.

The Reactor pattern could be configured in many different ways depending on the event demultiplexing capabilities provided by an 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 event handler could be managed by the same thread of control that was listening for events giving rise to a single-threaded Reactor implementation. Alternately,

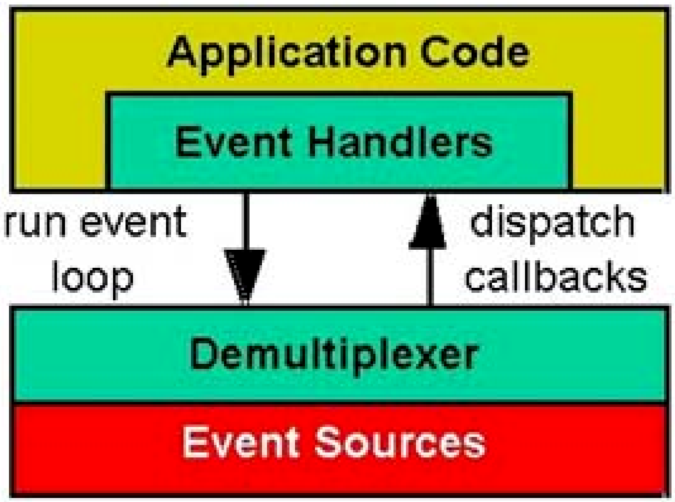

Fig. 2. Event Demultiplexers in Middleware

the event could be delegated to a pool of threads to handle the events giving rise to a thread-pool Reactor (other variants are also possible). These different configuration choices are dictated by the functional and performance needs of service-oriented systems, which means that service-oriented developers are responsible for handling this horizontal dimension of variability.

2.3 Impact of Variability on Systems Analyses

Sections 2.1 and 2.2 describe the composition and configuration variability showcasing the challenges that need to be overcome in selecting the right set of building blocks for middleware. Resolving these challenges is important due to the significant variability [6] and its impact on performance that is manifested by the need to compose service-oriented systems by selecting the right set of individual building blocks to assemble. Moreover, the fact that each building block can be customized in different ways adds to the variability in performance characteristics of the composed system.

In addressing these sources of variability, there is a need to estimate expected end-to-end performance provided by the customized middleware very early in the development lifecycle. Otherwise service-oriented developers will be forced to rely on rigorous testing later in the lifecycle to determine if the design choices were appropriate. Unfortunately, these decisions occur very late in the lifecycle of service-oriented systems, which adversely impacts costs and schedules. A mechanism is needed that will enable performance estimation of composable systems at design time. It is well understood that any changes at design-time are much easier to incorporate into the system. Moreover, the higher-levels of abstraction and formalism provided by the design-phase makes it possible to formally verify the systems for correctness.

2.4 Solution Approach: Model Driven Software Development

Model-driven Software Development (MDSD) [13, 17] has gained prominence in assisting service-oriented developers to make the right choices in configuring and composing large systems. Model-based performance analysis can be used to analyze the performance of patterns-based service-oriented systems. In order to facilitate the use of model-based performance analysis, however, it is necessary for MDSD tools to provide analytical models of the various possible patterns. These models can be used to estimate the performance of each pattern when used in different configurations and contexts. Subsequently, these models can be used in a plug-and-play fashion to form larger models corresponding to composition of patterns in the service-oriented system.

The domain of interest of a DSML can range from something very specific, such as the elements of a radar system, or can be as broad as the set of all the component-based middleware applications built using platforms such as EJB, or CCM. The key idea behind DSMLs is their ability to capture the elements of the domain as *first-class objects*.

A DSML can be viewed as a five-tuple [13] consisting of:

(1) **Concrete Syntax (C)**, which defines the specific notation (textual or graphical) used to express domain elements,
(2) **Abstract Syntax (A)**, which defines the concepts, relationships and integrity constraints available in the language,
(3) **Semantic Domain (S)**, which defines the formalism used to map the semantics of the models to a particular domain,
(4) **Syntactic Mapping ($\mathbf{M}_C$: A→C)**, which assigns syntactic constructs (*e.g.*, graphical and/or textual) to elements of the abstract syntax, and
(5) **Semantic Mapping ($\mathbf{M}_S$: A→S)**, which relates the syntactic concepts to those of the semantic domain.

Thus the abstract syntax determines all the (syntactically) correct "sentences", *i.e.*, models, in the language. A DSML is also known as a "metamodel", since a DSML is (itself) a model that defines all the possible models that can be built using it. To support the development of DSMLs, we have developed a "meta-programmable" modeling environment called the Generic Modeling Environment (GME) described in Sidebar 1.

Sidebar 1: Generic Modeling Environment

Generic Modeling Environment (GME) is an open-source, visual, configurable design environment for creating DSMLs and program synthesis environments, available for download from `escher.isis.vanderbilt.edu/downloads?tool=GME`. A unique feature of GME is that it is a *meta-programmable* environment. By *meta-programmable*, we refer to the fact that GME is not only used to build DSMLs, but also to build models that conform to a DSML. In fact, the environment used to build DSMLs in GME is itself built using another DSML (also known as the *meta-metamodel*) called "MetaGME". GME provides the following elements to define a DSML:

- **Project**, which is the top-level container in a DSML,
- **Folders**, which are used to group collections of similar elements together,
- **Atoms**, which are the indivisible elements of a DSML, and used to represent the leaf-level elements in a DSML,
- **Models**, which are the compound objects in a DSML, and are used to contain different types of elements like References, Sets, Atoms, Connections *et al.* (the elements that are contained by a Model are known as *parts*),
- **Aspects**, which are primarily used to provide a different viewpoint of the same Model (every part of a Model is associated with an Aspect),
- **Connections**, which are used to represent relationships between the elements of the domain,
- **References**, which are used to refer to other elements in different portions of a DSML hierarchy (unlike connections which can be used to connect elements within a Model),
- **Sets**, which are containers whose contained elements are defined within the same aspect and have the same container as the owner.

3 Addressing Middleware Variability via Modeling Pattern Languages

Architectural patterns present in middleware systems are discussed extensively in the book "Pattern Oriented Software Architecture: Patterns for Networked and Concurrent Systems" [21]. Our pattern modeling language, based on patterns described in this book, is called the Pattern Oriented Software Architecture Modeling Language (POSAML). It enables modeling of the patterns given in [21]. Besides pattern modeling, POSAML also incorporates feature modeling and benchmark modeling. Thus POSAML consists of the following three aspects:

(1) Pattern Aspect: The pattern aspect is where a system modeler can compose and model the various patterns in the system. Figure 3 shows an example where the designer has modeled the Reactor and Acceptor-Connector patterns. 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 figure. Sections 3.1 and 3.2 talk in depth about pattern modeling of the Reactor and Acceptor-Connector patterns respectively.

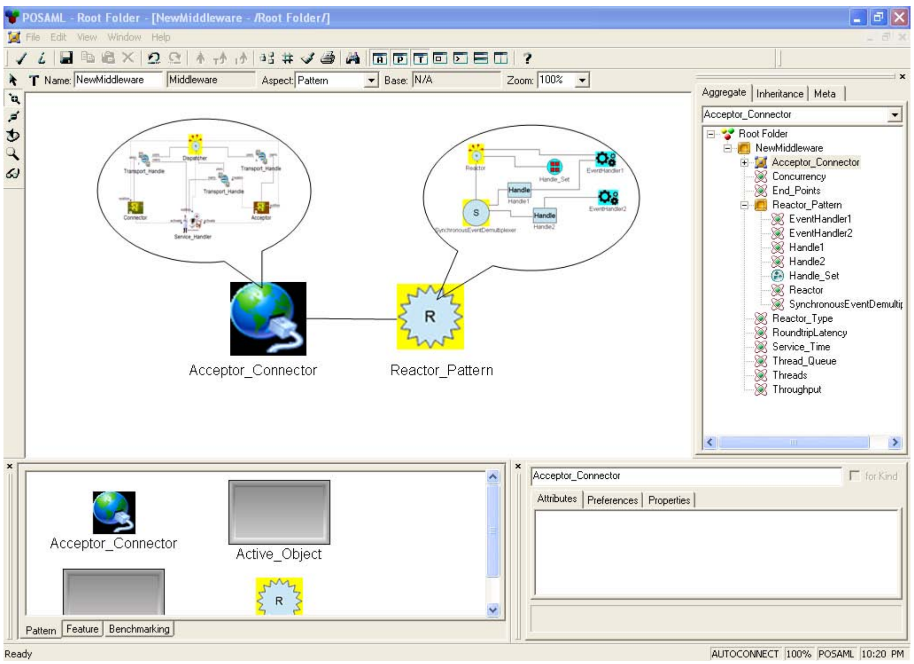

Fig. 3. Overview of POSAML

From the figure, it can be seen that POSAML follows a hierarchical structure. At the top-most level one can model inter-pattern relationships and constraints. At the lower level, a designer can go “inside” each pattern to model the participants of the pattern and the intra-pattern relationships between them.

(2) Feature Aspect: The system designer can set various features of each pattern in the feature aspect of POSAML. For example, the designer can specify the “End-points” feature for the Acceptor-Connector Pattern. The features are written to a configuration file by the Feature interpreter. This configuration file can be used to change the configuration of the middleware system. Section 4.2 discusses the Feature Aspect of POSAML.

(3) Benchmarking Aspect: The designer can select which benchmarking parameters to set for the performance analysis of the modeled system. These parameters are then written to an XML file by the benchmarking inter-

preter. This file can be used by an existing benchmarking library. Section 4.1 describes the Benchmarking Aspect of POSAML.

Figure 4 shows the meta-model for the top-most view of pattern modeling. This meta-model consists of the individual pattern models, and specifies how they can be connected to each other. In addition to this, this meta-model also defines the three aspects of POSAML, namely Pattern, Feature and Benchmarking.

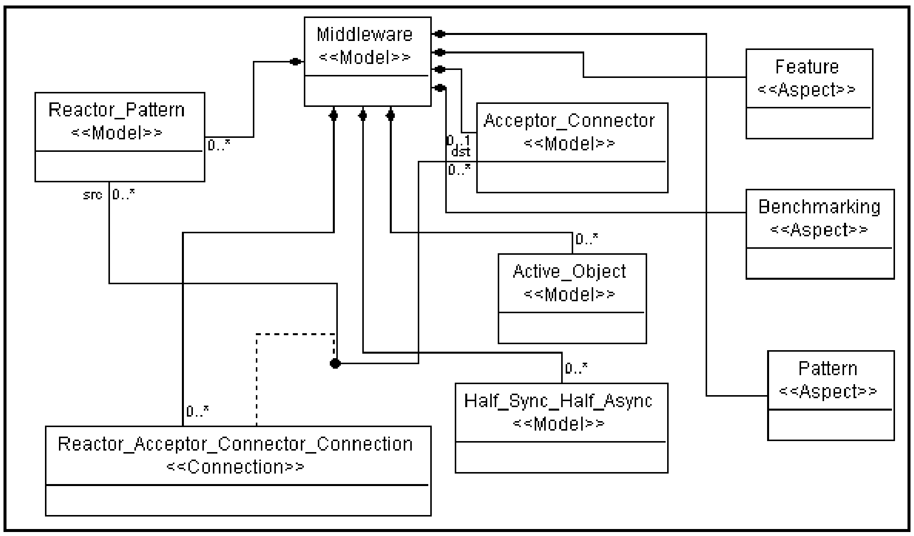

Fig. 4. Top-level Metamodel of Middleware

In the next two subsections, we illustrate the use of POSAML to model two patterns, namely the Reactor and the Acceptor-Connector.

3.1 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 a real-time, event-driven and performance-critical environment. The Reactor architectural pattern [21] 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 5 [21] shows the structural composition of the reactor pattern.

The Reactor meta-model that we have developed for POSAML is given in Figure 6. This meta-model is based on the structural representation of the Reactor pattern that is shown in Figure 5. In addition to this, the meta-model includes certain cardinality and relationship constraints (for example, a reactor

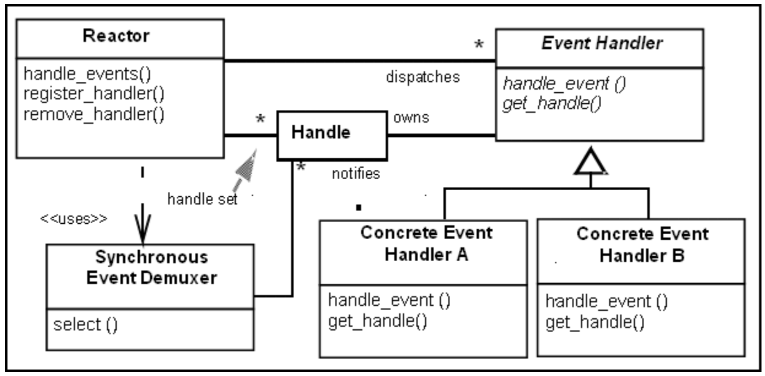

Fig. 5. Structural Represenation of the Reactor Pattern

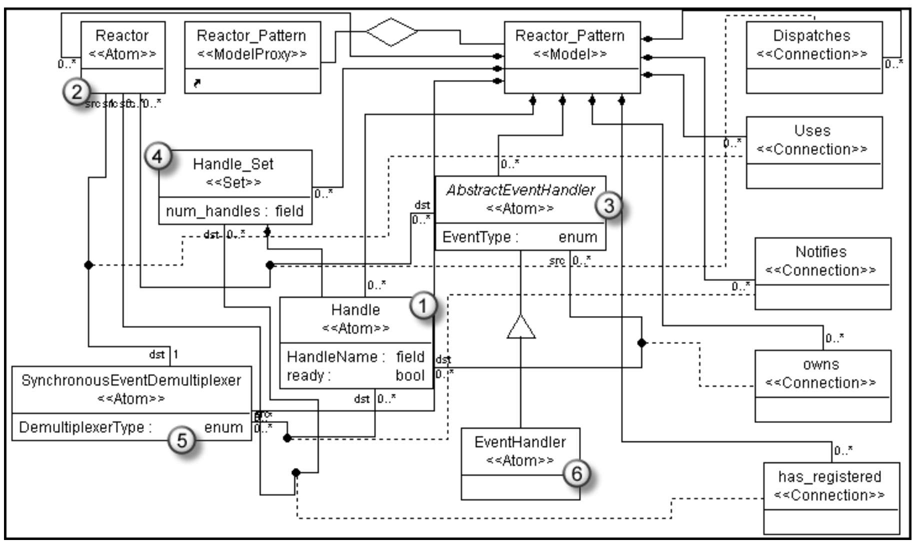

Fig. 6. Metamodel for the Reactor Pattern

can be connected to one and only one synchronous event demultiplexer). These constraints ensure that the modeler does not build an incorrect model.

The Reactor pattern consists of the following participants:

(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. Figure 5 shows three types of concrete event handlers: the Acceptor, the Connector and the Service Handler.

Figure 7 [21] illustrates the dynamics of the Reactor pattern.

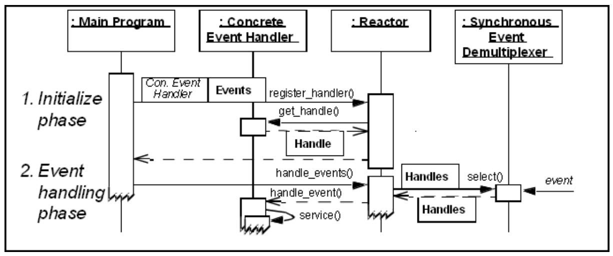

Fig. 7. Dynamics of the Reactor Pattern

In the initialization phase, the application registers its event handlers, and the handles that they accept, with the reactor. The reactor collects all the registered handles into a handle set. In the event handling phase, whenever an event occurs, the synchronous event demultiplexer channels it to the reactor. Based on the handle for that event, the reactor dispatches the corresponding registered event handler, which then processes the event. Additional information about the Reactor architectural pattern can be found in [21].

A system developer can model the Reactor pattern using POSAML as shown in Figure 8.

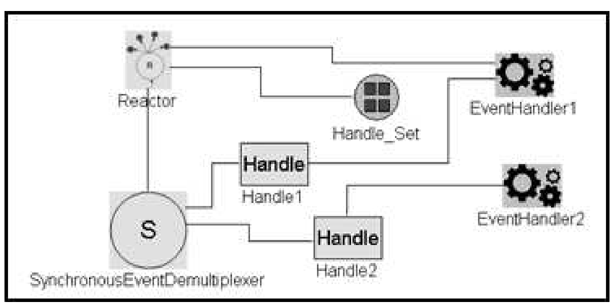

Fig. 8. Model of the Reactor Pattern

It can be seen here that the system developer has modeled the reactor pattern as having two event handlers. The service time for these handlers can be fixed in the benchmarking aspect (refer Section 4.1). Due to the constraints specified in the meta-model (refer Figure 6), the modeler cannot connect the various objects in the model arbitrarily. For instance, the modeler would not be allowed to connect the event handlers to the synchronous event demultiplexer;

this would raise an error. In this way, through the use of constraints, POSAML only allows the system designer to model his/her patterns in a correct manner.

3.2 Modeling of Acceptor and Connector Pattern

It is a design pattern that decouples connection establishment and service initialization in a distributed system from subsequent processing performed by the two end points of a service once they are connected and initialized [23]. This decoupling is achieved by acceptors, connectors and service handlers. Connection oriented protocols provide reliable delivery of data between two or more end points of communication [24]. Establishing connection between endpoints involves two roles:

(1) Passive role: This type of role initializes endpoints of communication at a particular address and waits passively for the other endpoint(s)to connect with it.
(2) Active role: This type of role actively initiates a connection to one or more endpoints that are playing the passive role.

It is important to know the communication role in terms of server and client. Mostly client is the one who plays an active role in connecting with an passive server but reverse is also possible. So, in Acceptor-Connector pattern, acceptor provides us with this passive role and connector plays an active role.

3.2.1 Participants:

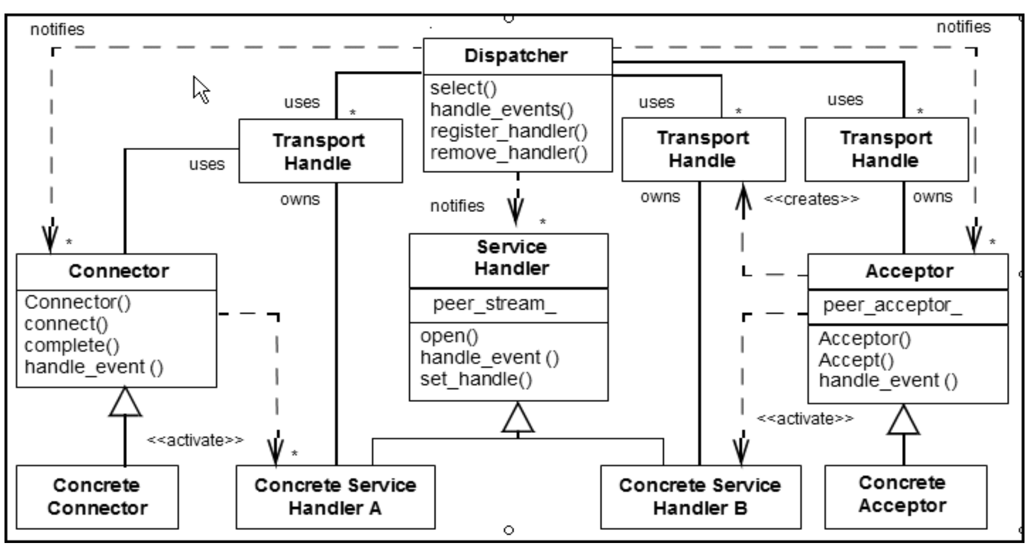

Fig. 9. UML diagram of Acceptor-Connector Pattern

The structure and participants of Acceptor-Connector pattern [23] is illustrated in Figure 3.2.1. This UML diagram which shows the relationship between different participants of this pattern. There are six key participants in this pattern. These participants are described as follows:

(1) Acceptor: It is a factory that implements passive strategy to establish a connection and initialize the associated Service Handler. It creates a passive mode endpoint transport factory (transport handle) that has necessary endpoints needed by the Service Handlers.
(2) Connector: It 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. It has synchronous (using Reactor pattern) and asynchronous mode (using Proactor pattern) of connections
(3) Dispatcher: It is in charge of registering, removing and dispatching the events that occurs on the registered handles by calling back application-specific Event Handlers. Dispatcher is for management of registered Event Handlers. In case of Acceptor, Dispatcher demultiplexes connection indication events received on transport handles. Multiple Acceptors can be registered within Dispatcher. And for Connector, the dispatcher demultiplexes completion events that arrive in response to connections.
(4) Service Handler: It is an abstract class that is inherited from Event Handler. It implements an application service, typically 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. The Service Handler uses the Handle component referencing to its peer for data-mode exchange with its connected peer Service Handler. So once the Service Handler is completely initialized by Acceptor and Connector factory it typically does not interact with these components any further.
(5) Transport endpoints: It is a factory that listens for connection requests to arrive, accepts those connection requests and creates transport handles that encapsulate the newly connected transport endpoints. Using these end points data can be exchanged by reading or writing to their associated transport handles. A transport handle encapsulates transport endpoint.

3.2.2 Acceptor-Connector Metamodel

In order to understand the detailed relationship between various participants of Acceptor-Connector pattern, we will discuss different phases that are there in the Acceptor and Connector. Connector pattern contains options to have synchronous or asynchronous type of connection but in POSAML language as of now we are supporting only synchronous mode of connection.

Now we will talk about the collaboration between server and client among different participants of Acceptor and Connector patterns at a very low level of abstraction. Both Acceptor and Connector pattern metamodel is part of single paradigm sheet called Acceptor-Connector. In order to understand Acceptor-Connector metamodel and model properly we have simplified it by dividing metamodel into two parts. These two parts are shown in fig 13 and 11 and are based on the UML diagram of Acceptor-Connector pattern shown in fig 3.2.1. In fig 11 we can see that the root element of this metamodel is Acceptor-Connector proxy model (1) and it is a proxy of Acceptor-Connector model, which is a part of our basic paradigm meta model called middleware shown in fig 4. Acceptor-Connector model proxy is in equivalence relation with local Acceptor-Connector model (1). Dispatcher atom(2) and Reactor atom proxy are equivalent and these are basically proxies of Reactor atom of Reactor pattern shown in fig 6. And this Dispatcher atom acts as event handler for both abstract Acceptor atom and Connector atom. Now we will discuss metamodel of Connector and Acceptor separately:

Collaboration of Connector: The behavior and flow of connector can be seen in figure 10. And, the collaboration between the participants in the synchronous Connector scenario can be divided into following stages:

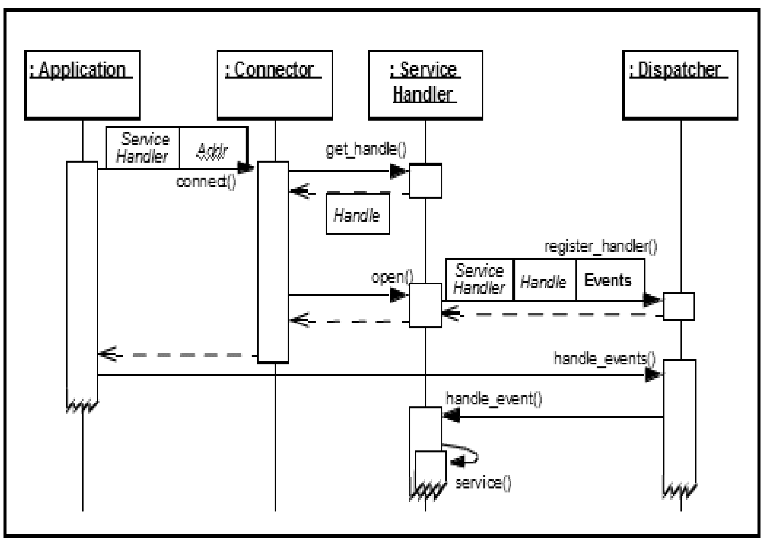

Fig. 10. Connector Dynamics

(1) Connection & Service initialization phase: In figure 11 a client creates a Dispatcher atom(2) that handles the events associated with that client. Then it creates one concrete Connector atom(3) derived from Abstract_Connector atom(3). It also activates concrete Service_Handler atom(4) derived from Abstract_Service_Handler atom(4) (that will be in charge of requesting the server peer service handler) for each type of service needed using Activate_Connector connection. For this, then the Connector atom generates an event and sends it on the Transport_Handle_Connector atom(5) of the server Acceptor of fig 13 in charge of the desired service. In return, the Connector atom gets a connection using End_Point atom(6) that

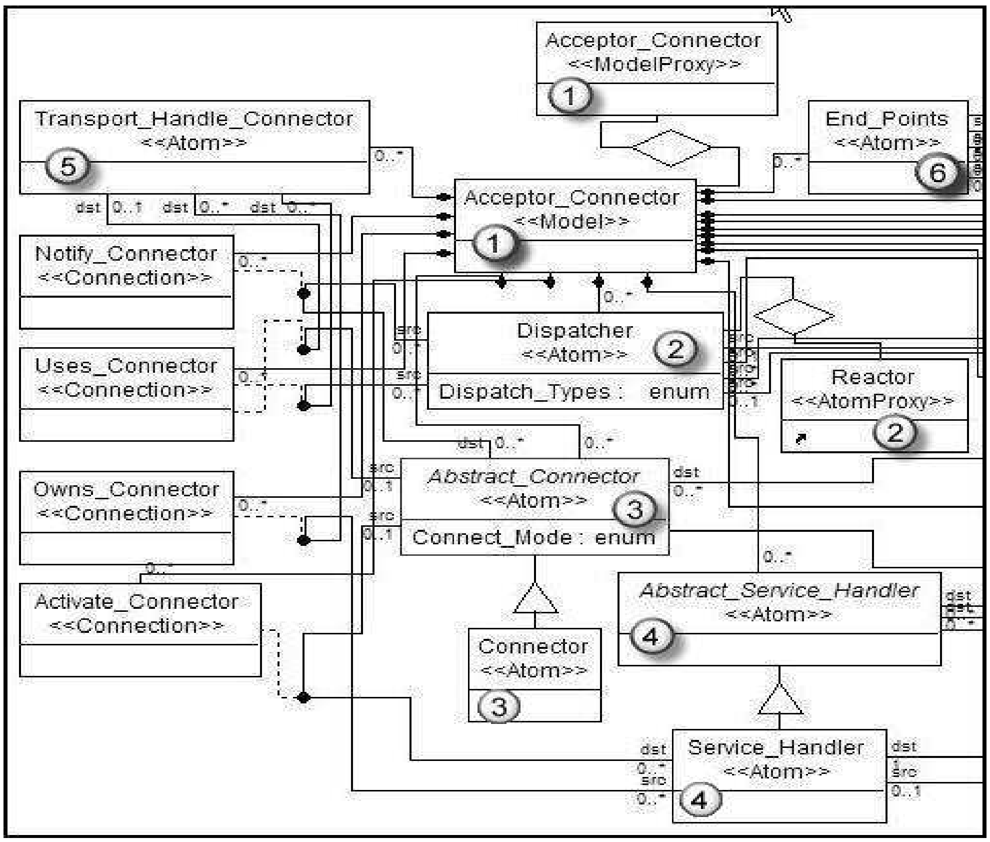

Fig. 11. Connector part of metamodel

corresponds to the Transport_Handle_Connector atom(5) of the peer Service_Handler atom(4). This End_Points atom(6) is again equivalent to End_Points atom proxy got from feature metamodel 4.2. This atom has three attributes like host name, listening port and the protocol. It can then register itself to the Dispatcher atom(2).

(2) Service processing phase: When the Dispatcher atom(2) detects an event on the Transport_Handle_Connector atom(5) of the client Service_Handler atom(4), it notifies the handle event method of this Service_Handler atom(4) using Notify_Connector connection . When the Dispatcher atom(2) detects a ready event on the client Service_Handler atom (4), it calls its handle event method. The Service_Handler atom (4) then gets the results from server. If a client Service_Handler atom(4) wants to close the service, it generates a close event on its server Service_Handler atom(4) Handle. Then it is removed from the list of Dispatcher atom (2) by calling the remove handler method.

Collaboration of Acceptor: The behavior and flow of acceptor can be seen in figure 12. Acceptor component provides means passive connection establishment. The collaboration between Acceptor and service handler participants and are divided into three phases:

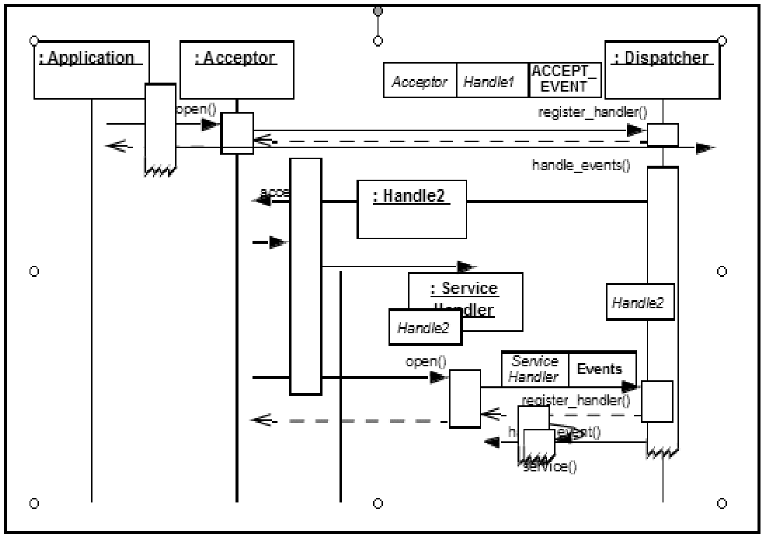

Fig. 12. Acceptor Dynamics

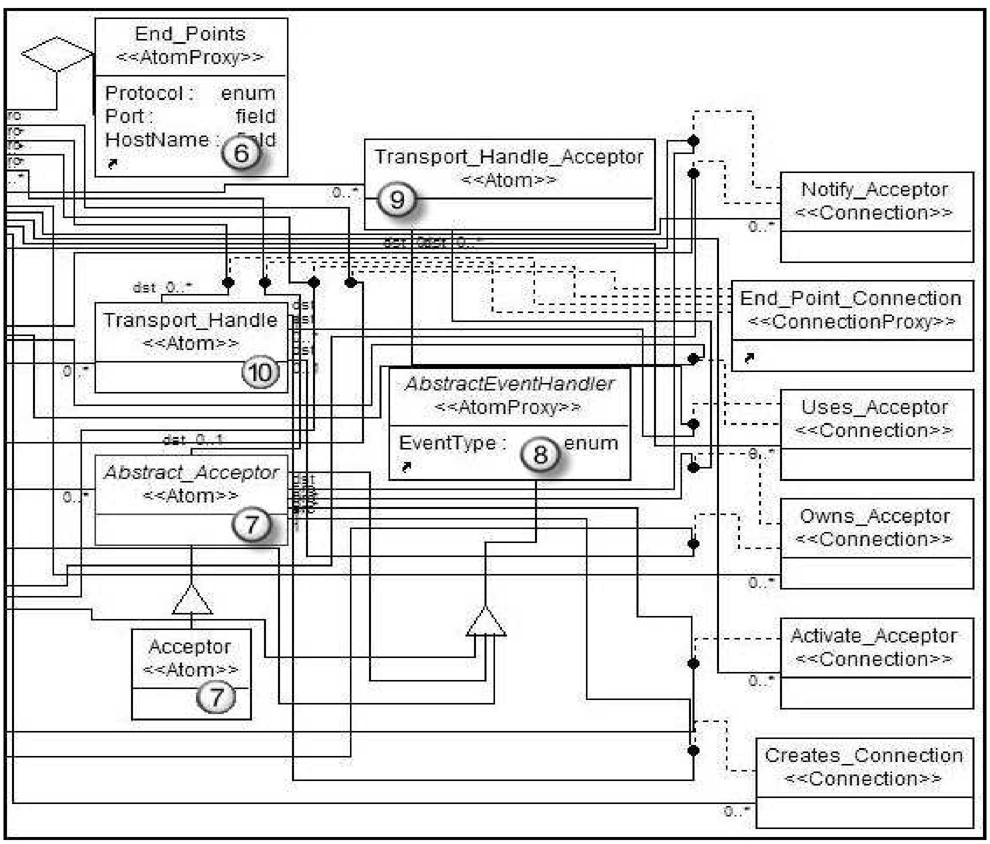

Fig. 13. Acceptor part of metamodel

(1) Initialization phase: In figure 13, Acceptor atom(7) is derived from Abstract_Acceptor atom(7) and it acts as server and initializes a connection passively. When an application calls the open method on Acceptor atom. This method creates a passive mode Transport_Handler_Acceptor(9) atom which encapsulates End_Points atom(6). It binds to end points i.e., IP

address, TCP Port and Protocol and then like a server it listens to connection requests from peer Connectors. These End_Points are used by Transport_Handlers(10) using End_Point_Connection connection proxy got from feature view of POSAML. Transport_Handler_Acceptor is owned by Acceptor by using Owns_Acceptor connection. After this the open method registers the Acceptor atom(7) object itself to Dispatcher atom(2). Dispatcher atom(2) is equivalent to Reactor proxy atom. This Reactor proxy atom and AbstractEvenHandler proxy atom(8) are defined in Reactor pattern in section 3.1 as Reactor atom(2) and AbstractEventHandler atom(3). After this the Dispatcher atom(2) can notify using Notify_Acceptor connection to Acceptor atom(7) whenever a connection event is coming from a client on the Transport_Handle_Acceptor atom(9).

(2) Service handler initialization phase: Dispatcher atom(2) detects some event on the Transport_Handle_Acceptor atom(9) of some Acceptor. It then calls the handle event method of this Acceptor atom(7) for him to handle this particular event. Acceptor atom(7) uses passive mode transport endpoint to create a new Transport_Handle atom (10) using Create_Connection connection. It next activates a new Service_Handler atom using Activate_Acceptor connection that will be in charge of processing the requests of the client. This Service_Handler(4) stores these new Transport_Handles atom(10) information and registers itself to the Dispatcher atom(2).

(3) Service processing phase: This stage is about the service use and closure. After above two stages, Dispatcher atom(2) will directly initiate server client communication using Service_Handler atom(4). In this stage exchange of data between peers can happen directly using already connected Transport_Handles (using transport endpoints). And after the whole process is done, Service_Handler(4) atom will call the remove handler method of the Dispatcher atom(2) to make it stop listening on its Transport_Handle for the events he had registered before.

A modeler can model Acceptor-Connector pattern in POSAML as shown in Figure 14. Various constraints are there to minimize the risk of choosing wrong combination of elements in the pattern. Only correct combination of connections and features are allowed to add for a particular pattern. For example only End Point feature can be added to Acceptor-Connector pattern. Some of these constraints comes in picture when we try to do model transformation using interpreters i.e., checking of connections between different objects of model.It is assumed that modeler will have basic knowledge of patterns i.e., flow between different objects. Modeler can have more than one acceptor in his model.

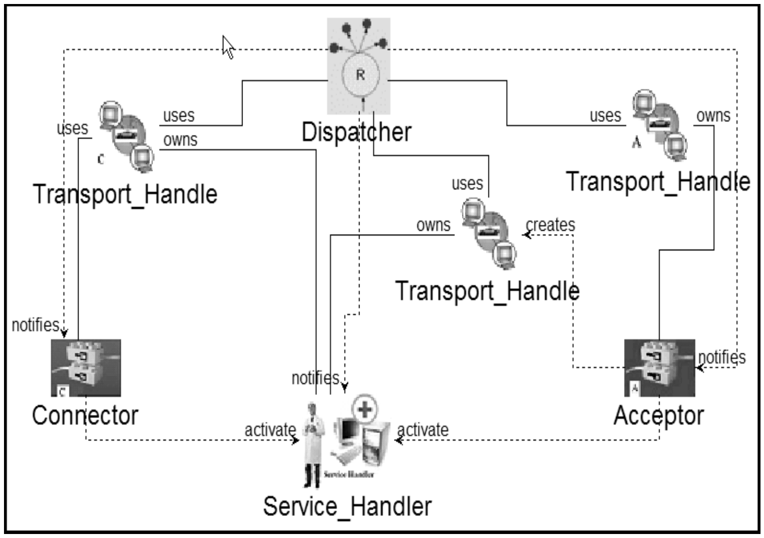

Fig. 14. Model of Acceptor-Connector

4 Modeling Separation of Concerns

4.1 Benchmarking

To enable the performance analysis of composed system, the modeling language provides a method to model benchmarking characteristics for the system. This can be done in the "Benchmarking Aspect". Figure 15 shows the meta-model for the Benchmarking paradigm.

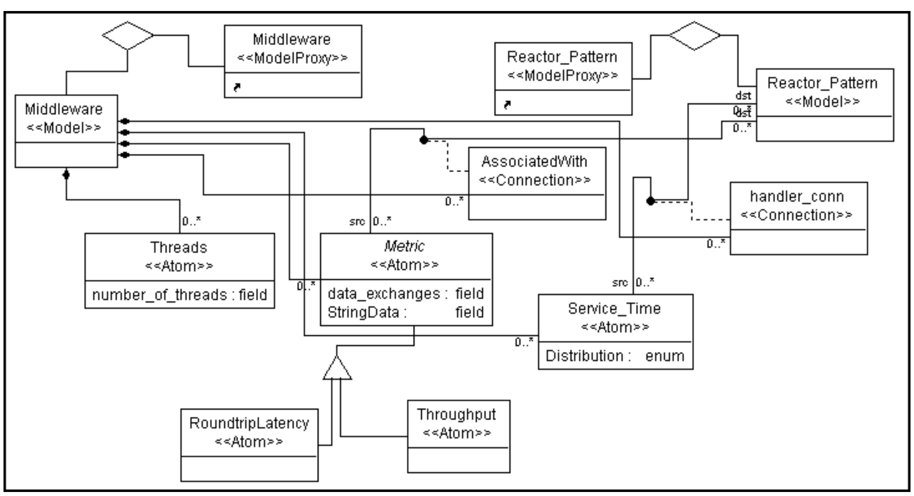

Fig. 15. The Benchmarking Meta-Model

The meta-model enables the system developer 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 handle time for each event handler. A sample model which can be constructed using this paradigm is shown in Figure 16.

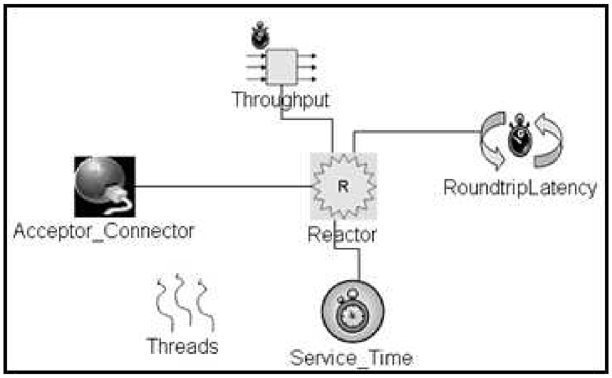

Fig. 16. Benchmarking Aspect

In this case the developer has modeled two patterns, the Reactor and the Acceptor-Connector, and the benchmarking characteristics to analyze the performance of the Reactor pattern. Therefore the latency and throughput metrics are attached to the Reactor pattern. In addition, the developer has modeled the number of client threads and the service time for each event handler in the Reactor Pattern.

4.1.1 The Interpreter for the Benchmarking Aspect

The benchmarking aspect enables a user to model the benchmarking characteristics of the system. Using the benchmarking interpreter, the developer can generate benchmarking parameters for an existing benchmarking library. These parameters can be 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 by an existing library.

```
- <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>
```

Our future work in this regard lies in auto-generating benchmarking code, instead of just generating benchmarking parameters for existing benchmarking libraries. Another approach which we are pursuing is the auto-generation of simulation models of the patterns using POSAML. This would be helpful in the design-time analysis of middleware systems.

4.2 Feature Modeling

A Feature model [7]is 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, feature modeling aspect allows model users to use domain-specific artifacts to model a system in contrast to using low-level platform-specific artifacts. The feature modeling in our modeling paradigm will provide structural representation of different possible middleware pattern properties. In our case the feature modeling comprise modeling several non functional and QoS requirements, such as the choice of network transport, listening end points, multithreaded requirements, periodicity of request represented as higher-level artifacts. This will enable system integrators to select various strategies, resources, settings and factories that can be parameterized according to user needs by driving it from a very high level feature modeling. Traditional middleware architectures suffer from insufficient model level reusablity. Feature model aspect of POSAML brings a way to set behavioural aspect of middleware.

Developing a feature model out of metamodel involves defining valid entity and relationships in the schema. All these features are pattern specific. For example Concurreny, 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.

This part of the section talks about the GME meta model of feature aspect of POSAML shown in figure 4.2. Some of the features designed in the meta model are as following:

(1) **Concurrency:** It is very important for different middleware to manages concurrendy and allow long running operations to execute simultaneously without impeding the progress of other operations. Default server concurrency strategy provided by ACE/TAO supports two types of concurrency [22]. It specifies the concurrency strategy an ORB uses. Concurrency has Concurrency_Option as attribute and this attribute has two strategies as menu item. These concurrency strategies are:
  (a) Reactive : This is a default concurrency type. It registers the connections transport end points with Reactor. When events occur, it

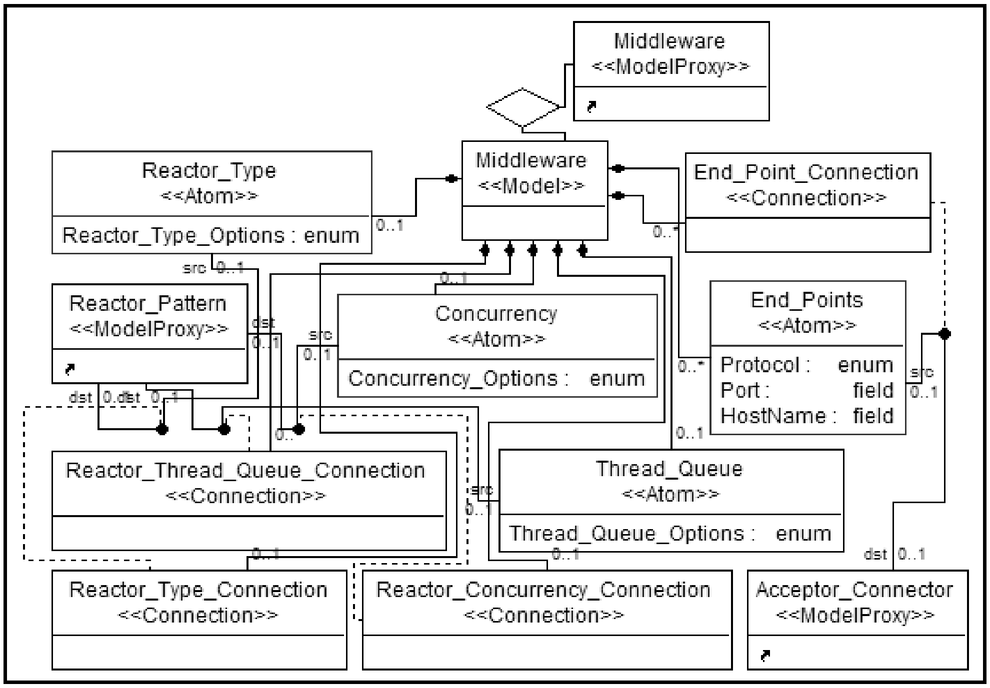

Fig. 17. POSAML Metamodel: Feature View

starts dispatching these events to the reactive connections in order. In this mechanism ORB handles request reactively i.e., the ORB runs in one thread and service multiple requests/connections simultaneously using the `ACE_Reactor`, which uses select or a similar event demultiplexing mechanism supported by the platform. Its connections are single-threaded and that is it is scalable but is not a very good concurrency mechanism.

(b) Thread Per Connection: The ORB handles new connections by spawning a new thread whose job is to service requests coming from the connection.

(2) **Reactor Type:** This feature is used to specify the kind of reactor the ORB uses. It is a advanced resource factory option and if this type of factory is loaded, any default directives of original resource factory will have not effect. Reactor_Type has Reactor_Type_Option as attribute and this attribute has five strategies as different reactor types: Thread pool (tp): It uses the `ACE_TP_Reactor`, a select based thread pool reactor which is the default. Multi threaded (select_mt): It uses the multi-thread select based reactor. Single threaded (select_st): It uses single-thread select based reactor. Wait for multiple objects (wfmo): It uses WFMO reactor and can be used only for windows. Message Wait for multiple objects (msg_wfmo): It uses Msg WFMO reactor. It is used only for windows.

(3) **Thread Queue:**: This feature helps to specify what type of thread queue to use. Thread Queue has a attribute called Thread_Queue_Options. This attribute has two options i.e first-in-first-out (LIFO) and last-in-first-out (LIFO). This feature applies only to the `ACE_TP_Reactor`, i.e., when

reactor type is thread pool. And it specifies the order, LIFO, the default, or FIFO, in which waiting threads are selected to run by the `ACE_Select_Reactor_Token`. Thread Queue is also a part of advance resource factory.

(4) **End points:** This feature instructs a server ORB to listen for requests on the interface specified by endpoint. ACE/TAO endpoints are specified using a URL style format. The various attributes of endpoints that can be modeled by our modeling language are listening ports(TCP port number), Host address or name and Protocol (TCP, UDP or Shared Memory).

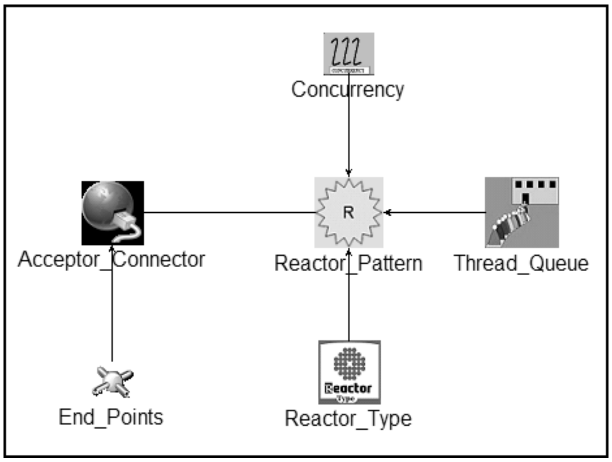

Fig. 18. POSAML Model: Feature View

A middleware developer uses feature aspect of POSAML modeling language as a visual tool to select different pattern specific features of middleware. In figure 4.2 we can see the how modeler can model various features in this modeling language. Modler can model zero or more features using this tool. Once the feature modeling part is done, then the next step is the tool to transform pattern specific features into configuration file using interpreters. If features are not selected from the model, default values of these features will be picked. In order to minimize the risk of choosing wrong connections and options, various constraints are there. Some of these constraints are checked using OCL constraint language i.e,. checking for non null references or proxies and some of them are checked at the time of The selected feature set with different options are exported into files using configurator interpreter and are used to configure middleware system . We will talk about configurator interpreter in section 4.3.

4.3 Configurator Interpreter

The dynamic semantics of POSAML are implemented via GME interpreters. In this modeling language we have two interpreters as of now and these interpreters traverse the graphical hierarchy of model which are modeled by application modelers. One of the interpreters called "Configurator Interpreter" generates two artifacts which are required to configure middleware. We have build POSAML platform independent but this interpreter is ACE/TAO specific and for other platforms One of the artifacts is the service configurator file (svc.conf) and the other one is a script file. Other interpreter called Benchmark interpreter will be discussed in other section 4.1. Service Configurator file is used to set Qos related configuration policies by middleware for different applications. It has different options that control the behavior of strategies and resources used by middleware framework. This file allows application to configure service objects statically and dynamically configuring middleware. Options [22]in svc.conf file can represents either the components provided by middleware framework or customized components developed by users. If this configurator file is not available, ACE/TAO middleware framework selects all default configurations.

Following are the different factories that can be configured:

(1) Advance Resource Factories: This factory controls the creation of configurable resources used by ACE/TAO's ORB core. The resource factory is responsible for constructing and providing access to various resources used by the ORB irrespective of whether they perform client or server roles.
(2) Server Strategy factory: This factory creates various strategies of special utility to the ORB that is useful for controlling the behavior of servers. This factory is responsible for creating strategies useful for server objects like the concurrency strategy and the request demultiplexing strategies used by the POA.
(3) Client Strategy factory: This factory creates various strategies of special utility to the ORB, useful for controlling the behavior of clients. This factory is responsible for creating strategies useful for clients such as request multiplexing strategies, wait strategies, connect strategies.

Svc.conf will contain strategies like this:

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

Script file has inputs to run any application like Naming service or benchmarking evaluation tool with proper end points like listening ports, protocol and host name. These endpoints are also used by Acceptor-Connector pattern for different transport handles.

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

5 Related Work

With the growing complexity of component-based systems, composing system-level performance and dependability attributes using the component attributes and system architecture is gaining attention. Crnkovic *et al.* [**?**] 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 such as the architecture and the environment. However, they do not propose any methods for composing the system-level attributes.

At the model-driven development and program transformation level, the work by Shen and Petriu [**?**] 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 considerd more readily (i.e., after separation, a specific peformance 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 Queueing Network (LQN) solver.

A disadvantage of the approach is that UML forces a specific modeling language. The SPT profiles 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. Our work falls in the category of developing domain-specific models.

There have been efforts to evaluate the performance of middleware patterns analytically by various researchers [10], [20]. 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* [12] have worked on the performance evaluation of the CORBA method invocation and threading models. However, they have not focused on the pattern-based approach toward the performance analysis of middleware. Model-driven techniques are increasingly being used for middleware development, but converting static pattern-based middleware models into simulation models for the purpose of performance evaluation has not yet been a focus of research in the research community.

There are various specialization techniques described in literature, which can be leveraged to specialize middleware. For example, Feature-Oriented Programming (FOP) [4] is an appropriate technique to design and implement program families, and which uses incremental and stepwise refinement approaches [4,26]. FOP aims to cope with the increasing complexity and increasing lack of reusability and customizability of contemporary software systems. Aspect-Oriented Programming (AOP) [14] is another related programming paradigm and has similar goals: It focuses primarily on separating and encapsulating crosscutting concerns to increase maintainability, understandability, and customizability. However, it does not focus explicitly on incremental designs or program families. Aspect-Oriented Programming can change an existing functionality without refactoring of code, addresses concerns with minimum coupling, makes it reusable and implements no hierarchy refinements. These features of AOP can lead to error free and efficient code. It can prevent code clutter, tangling and scattering and makes it easy to add new functionality by creating new aspects. New features or behavior can be added at any stage of development thus relieving the developer of committing to under/over design. So an unknown functionality which cannot be predicted ahead of time is not a problem.

These characteristics of AOP can be leveraged to create an implementation that is easier to design, understand, and maintain resulting in higher productivity, improved quality, and better ability to implement newer features. We therefore leverage these capabilities of AOP as a middleware specialization technique for product lines.

6 Conclusions

References

[1] Extensible Markup Language (XML) 1.0 (Second Edition). `www.w3c.org/XML`, Oct. 2000.

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

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

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE*, May 2003.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[6] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

[7] K. Czarnecki and U. Eisenecker. Generative programming: Methods, tools, and applications. Addison-Wesley, 2000.

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

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

[10] S. Gokhale, A. Gokhale, and J. Gray. Model-driven performance analysis methodology for distributed performance sensitive software systems, 2005.

[11] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[12] M. Harkema, B. M. M. Gijsen, R. D. van der Mei, and Y. Hoekstra. Middleware performance: A quantitaive modeling approach. In *International Symposium on Performance Evaluation of Computer and Communication Systems (SPECTS)*, 2004.

[13] 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.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, and Cristina. Aspect-oriented programming. In *In proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[15] 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.

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

[17] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

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

[19] 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.

[20] 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.

[21] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Addison Wesley, 2000.

[22] D. C. Schmidt. Options for tao components. In *The TAO Documentation*.

[23] D. C. Schmidt. Acceptor-connector: An object creational pattern for connecting and initializing communication services. In *Pattern Languages of Program Design*, 1995.

[24] D. C. Schmidt. Design patterns for active and passive establishment of network connections. Aug 1995.

[25] 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.

[26] O. Spinczyk and D. Lohmann. Aspect-oriented programming with c++ and aspectc++. In *AOSD*, 2005.

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

[28] W3C. Simple Object Access Protocol (SOAP) 1.1. `www.w3c.org/TR/SOAP`, May 2000.