MODEL-DRIVEN COMPOSITION AND PERFORMANCE EVALUATION OF PATTERN-BASED SYSTEMS

By

Arundhati Kogekar

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May 2007

Nashville, Tennessee

Approved:

Professor Aniruddha Gokhale

Professor Lawrence W. Dowdy

Professor Swapna Gokhale

DEDICATION

This thesis is dedicated to my parents, Radhika and Vijay Kogekar.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Aniruddha Gokhale, for providing valuable guidance during the course of my Master's research. I am grateful to my thesis committee members Dr. Lawrence Dowdy and Dr. Swapna Gokhale for their insightful comments. I would also like to express my gratitude to Dr. Douglas Schmidt for his constructive feedback and suggestions. I appreciate all the help provided to me by my fellow graduate students at Vanderbilt University and at the Institute of Software Integrated Systems.

Last but not the least, I would like to acknowledge the support and encouragement provided by my family during my Master's studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Real-time, performance-critical distributed systems are used in many domains, such as telecommunications, power grid and enterprise systems among others. These systems have diverse and stringent Quality of Service (QoS) requirements such as scalability, fault tolerance and reliability. The design of such contemporary large-scale systems is based on elegant patterns as well as pattern languages [6]. Such patterns-based systems are designed and implemented by composing together different pattern-based functional building blocks. Patterns [6] represent solutions to a common set of problems arising in a particular context. A pattern therefore is a body of expert knowledge on best practices, designs and strategies that has been documented in a standardized manner and that can therefore be reused in similar situations. In the context of large-scale distributed systems, patterns represent solutions for common distributed and network programing tasks such as event handling, memory management, service access and configuration, concurrency and synchronization [17]. Figure I.1 [5] shows a pattern-based distributed middleware architecture composed of building blocks representing these patterns.

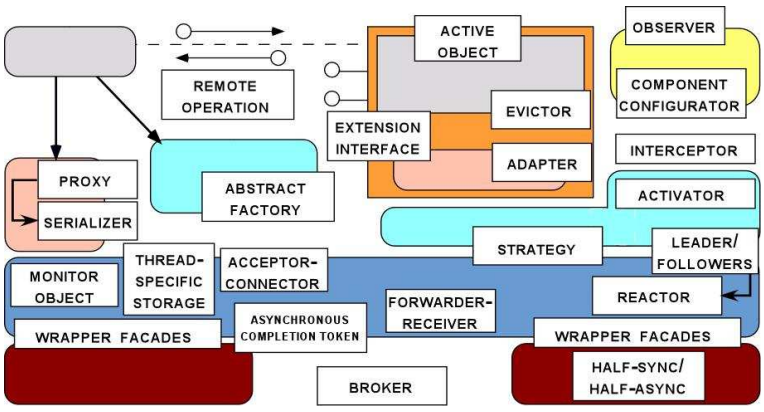


Figure I.1: Patterns in Middleware

Visual modeling languages make it easer for system architects to intuitively design complex systems. Using domain-specific visual modeling languages for designing a system raises the level of abstraction, which makes design reuse possible. Model-driven techniques make the task of designing the system easier by disentangling various orthogonal design-time concerns such as system composition, configuration and behavior, while still ensuring that the designer is aware of the effect of change within each concern on the end system. This thesis describes the Pattern-Oriented Software Architecture Modeling Language (POSAML), which enables the system architect to compose and configure his/her complex system using pattern-based building blocks within the Generic Modeling Environment (GME) [19].

Usually, the building blocks of large-scale system come equipped with a number of customizable configurations. It is then the task of the system architect to select the right building blocks or components, evaluate the performance of these configurations and select the one that is best suited for that particular domain. While the design of such systems is in itself a hard task, a major challenge faced by the designer is a lack of understanding of how different system configurations affect the QoS of the end system. In traditional systems development, the architect often has to wait very late into the system life-cycle, for example, until runtime, to validate the system and its configuration, which is both costly and time-consuming. Here again, model-driven technologies provide the capability to conduct "What-If" design time analysis of the system. Incorporating performance analysis of the system at design time itself provides greater flexibility in changing system configurations if they are found to be sub-optimal. If the configuration that provides the most optimum performance is known at design time, code can be written only for that configuration. Similarly, the sooner the design flaws are detected in the system life-cycle, the easier and cheaper it is to correct them. This highlights the benefit of using design-time performance

analysis, such as simulations, to predict the system performance at design time, when there is still time and ample opportunity to change the design without wasting additional resources. POSAML therefore provides the capability of modeling simulation and benchmarking parameters along with the actual system model. In this manner, performance evaluation artifacts can be auto-generated from the model. These can then be plugged into existing libraries so that the system can be evaluated at design time itself.

The emphasis on design-time performance analysis of distributed systems now maps to the performance evaluation of the patterns-based building blocks using simulation mechanisms. The simulation of a pattern-based building block, however, presents its own set of challenges. Each block interacts in different ways with other blocks. For example, the reactor pattern has to deal with numerous simultaneous events. The event-handling mechanism in the reactor allows concurrency by enabling event handling in multiple threads concurrently. This necessitates the use of a powerful discrete event-based simulator which would be able to simulate the simultaneous occurrence and handling of events. In addition, the simulator must be able to incorporate any modifications to the model easily, i.e, the simulation of the combination of two or more building blocks should not require extensive rewriting of existing simulation code.

Generally, when evaluating a system using simulations, the effect of each parameter is evaluated by running simulations multiple times. Each time, it is tedious to rewrite the parameter file required for the simulations. For a system with multiple input parameters and multiple metrics, manual effort expended in rewriting these files becomes considerable. This effort can be saved if simulation artifacts are generated from the model itself. This capability is also provided by POSAML.

Performance evaluation at run-time is also important to validate that the end

system, and not just the system design, does indeed meet its Quality of Service (QoS) goals. As in the case of simulations, benchmarks are run multiple times for multiple system configurations. It is tedious and error-prone to specify these benchmarks by hand. Auto-generating these parameter files would therefore save considerable time and effort. In addition, specifying benchmark characteristics at design time along with the design gives the system designer control in the run-time evaluation of the system. It also provides an intuitive interface where benchmark characteristics can be associated with specific blocks of the system. This makes it easier for a developer to understand and implement the benchmarks. The capability of modeling benchmark characteristics and generating benchmark parameter files from the models is also provided by POSAML.

Thesis Organization The thesis is organized in the following manner:

Chapter II of the thesis discusses related work in the area of modeling and performance evaluation of large-scale systems. Chapter III provides an overview of POSAML and its different aspects. Chapter IV describes the Structural View of POSAML in detail. This is illustrated by describing how the Reactor and Active Object patterns are modeled in POSAML. The use of POSAML in performance evaluation by simulation and benchmarking is elaborated in Chapter V. The simulation set-up, results and analysis of the Reactor pattern using the OMNeT++ simulator are described in Chapter VI. Chapter VII elaborates on future work and provides the conclusion for the thesis.

CHAPTER II

RELATED WORK

With the growing complexity of component-based systems, composing system-level performance and dependability attributes using component attributes and system architecture is gaining attention. Crnkovic *et al.* [4] classify the quality attributes according to the possibility of predicting the attributes of the compositions based on the attributes of the components and the influence of other factors within the architecture and the environment. However, they do not propose any methods for composing the system-level attributes.

At the model and program transformation level, the work by Shen and Petriu [18] investigated the use of aspect-oriented modeling techniques to address performance concerns that are weaved into a primary UML model of functional behavior. It has been observed that an improved separation of the performance description from the core behavior enables various design alternatives to be considered more readily (i.e., after separation, a specific performance concern can be represented as a variability measure that can be modified to examine the overall systemic effect). The performance concerns are specified in the UML profile for Schedulability, Performance, and Time (SPT) with underlying analysis performed by a Layered Queuing Network (LQN) solver.

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

There have been efforts to evaluate the performance of middleware patterns analytically by various researchers [7, 16]. 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* [9] 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.

An approach for generating simulation programs from UML diagrams is explained by Arief and Speirs in [2]. The authors describe a Java-based UML tool which can be used to generate XML simulation artifacts from UML class and sequence diagrams.

Extensive research has been done in the area of regression benchmarking [3] by Kalibera, *et al* [10]. The authors describe a tool suite for the regression benchmarking of Mono, an open-source middleware platform. Another hand-crafted benchmarking suite is the OpenCORBA Benchmarking initiative by Tuma, *et al* [20]. However, like other handcrafted techniques, its implementation takes a lot of tedious effort. In addition, it presents certain difficulties such as combining various performance factors, incorporating network conditions and background load, and ensuring the portability of results. A model-driven approach toward benchmarking middleware is CCMPerf [14], which overcomes these limitations. Additionally, it provides the right level of visual abstraction necessary to design and set up an experiment, as well as auto-generating low-level, error-prone code.

CHAPTER III

POSAML - A VISUAL MODELING LANGUAGE FOR SYSTEM COMPOSITION AND EVALUATION

The Pattern Oriented Software Architecture Modeling Language (POSAML) is a domain-specific visual modeling language which enables system architects to compose and configure complex distributed systems. Model-based solutions based on visual aids can help resolve the variability in complex systems such as distributed middleware as well as provide automated QoS validation. POSAML provides QoS validation by virtue of plugging in different model interpreters and enabling system architects to run simulations based on their designs.

POSAML provides the following "*views*" or "*aspects*" to model the system:

III.1 Structural View

The design of complex, hierarchical systems often consists of assembling individual but compatible building blocks. These building blocks most often are patterns-based. A software pattern [6] codifies recurring solutions to a particular problem occurring in different contexts, which is embodied as a reusable software building block. The systems developer chooses a block based on various factors including the context in which the application will be deployed, the concurrency and distribution requirements of the application, the end-to-end latency, timeliness requirements for real-time systems, or throughput for other enterprise systems (e.g., telecommunications call processing).

Figure III.1 illustrates a family of interacting patterns forming a pattern language [1] for middleware designed to support such applications. The middleware can

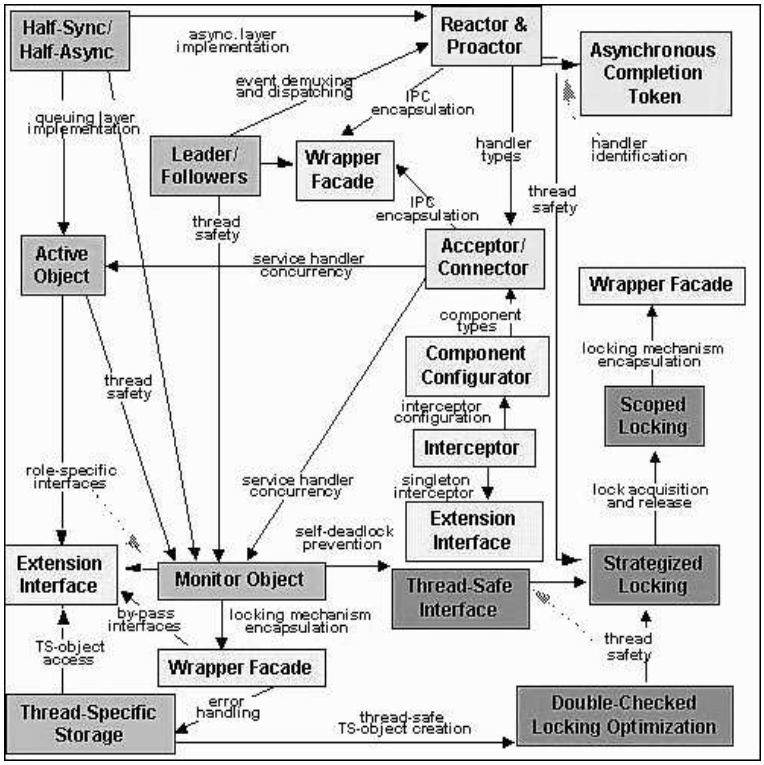


Figure III.1: Middleware Patterns and Pattern Languages

be customized by composing compatible patterns. For example, event demultiplexing and dispatching via the Reactor or Proactor pattern can be composed with the concurrent event handling provided by the Leader-Follower or Active Object pattern. 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.

The *Structural View* of POSAML, also known as the *Pattern Aspect*, is where a system modeler can compose and model the various patterns in the system. 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.

To illustrate this point, Chapter IV describes in detail how one can model compositions of patterns such as Reactor and Active Object in the structural view of POSAML.

III.2 Feature View

Complex systems are equipped with numerous configuration options to customize the system behavior of so that the system meets the QoS demands. This flexibility further exacerbates the already incurred variability in design choices that the systems developer is required to make. 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. Judging the best configuration manually from a myriad of choices is very difficult. The visual modeling capabilities of the *Feature View* of POSAML provide a solution to this problem. The feature view of POSAML allows model users to use domain-specific artifacts to model a system in contrast to using low-level platform-specific artifacts. Once the feature modeling is done, the next step is for the tool to transform pattern specific features into a configuration file which can be used by the end system. Various constraints are in place to minimize the risk of choosing wrong connections and options. Some of these constraints are checked using Object Constraint Language (OCL) and some of them are checked at the time of generating a configuration file corresponding to these features. The Feature View of POSAML is explained in detail in previous work [11].

III.3 Simulation View

While it is important to be able to compose and configure the system at design-time, it is equally important to validate those compositions and configurations so that they provide the best QoS for the system. Design-time validation ensures that design errors do not propagate further, thus saving considerable time and effort. POSAML provides the *Simulation View* for design-time validation. In this view, the system architect can model simulation parameters of interest to either the composed system or to individual building blocks. The simulation interpreter goes through this model and generates simulation artifacts such as simulation initialization files. These can be plugged into existing simulation libraries to run simulations for various configurations and compare the performance of each configuration.

III.4 Benchmarking View

POSAML also provides capabilities to design benchmarks to evaluate the final system at run-time. The system designer can model certain benchmarking characteristics (such as the metric to measure and the input parameters) in the *Benchmarking View* of POSAML. These benchmarking characteristics can be exported using a "benchmarking interpreter" to provide inputs to an existing benchmarking library. This capability ensures that the system designer, who is also a domain expert, has a say in deciding which benchmarks to run when the system has finally been implemented. The Benchmarking Aspect and its associated interpreter are further described in Section V.2.

CHAPTER IV

STRUCTURAL MODELING USING POSAML

As described earlier, large-scale systems are composed of pattern-based building blocks. It is easier to design a system by assembling these reusable blocks. POSAML allows a system architect to model this in its *Structural View*, also known as the *Pattern Aspect*. In this view, the architect can compose his/her system by selecting and connecting the appropriate building blocks.

Figure IV.1 illustrates the high-level POSAML meta-model defined in GME.

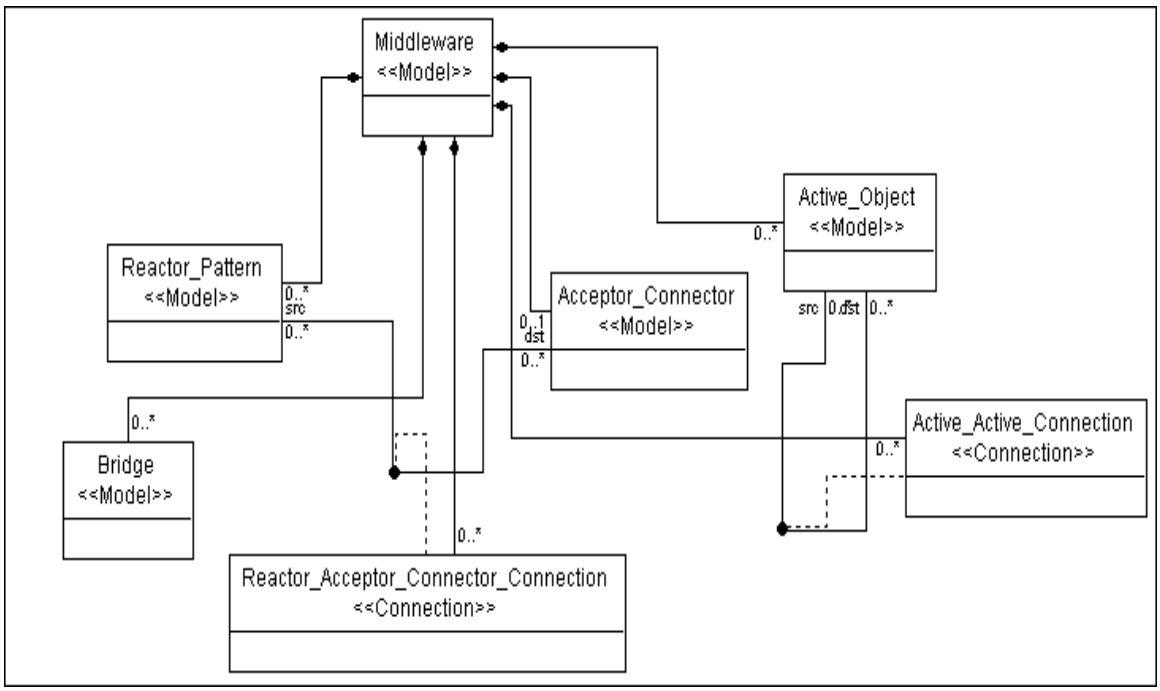


Figure IV.1: Top-Level Meta-Model of Structural View

This meta-model enables a system architect to model various individual patterns as well as their composition in the structural view of POSAML. While the figure shows a "Middleware Model" as being composed of patterns, POSAML can be used for modeling any kind of hierarchical, pattern-based large-scale system. For instance,

the designer can model the individual Reactor, Acceptor-Connector, Bridge or Active Object patterns, as well as a composition of the Reactor-Acceptor-Connector patterns, or a combination of multiple active object patterns. To model individual patterns, this high-level meta-model is connected to individual pattern meta-models shown in Figure IV.4 and Figure IV.7.

Figure IV.2 shows an example where the designer has modeled the Reactor and Acceptor-Connector pattern-based blocks using POSAML. In a client-server appli-

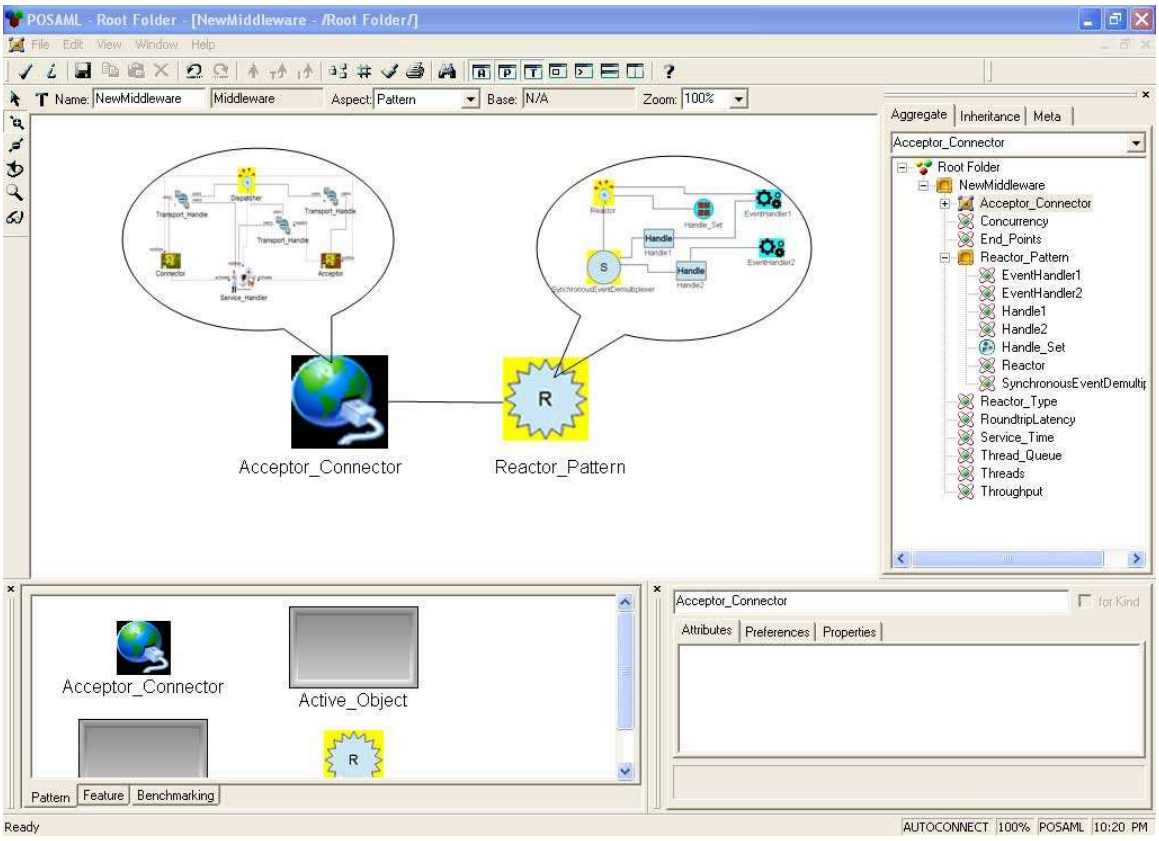


Figure IV.2: Overview of POSAML

cation, the Reactor would exemplify event handling within the server, while the Acceptor-Connector would demonstrate the communication mechanisms between the client and server. In addition to this high-level view, the user can click on any one of the patterns and model its internals, as shown in Figure IV.2. This highlights the

fact that POSAML is well-suited to modeling hierarchical pattern-based systems. The next two sections describe how the Reactor and Active Object patterns are modeled in POSAML.

IV.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 systems designed for use in real-time, event-driven and performance-critical environments. The Reactor [17] 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 IV.3 illustrates the structure of the Reactor Pattern in UML notation.

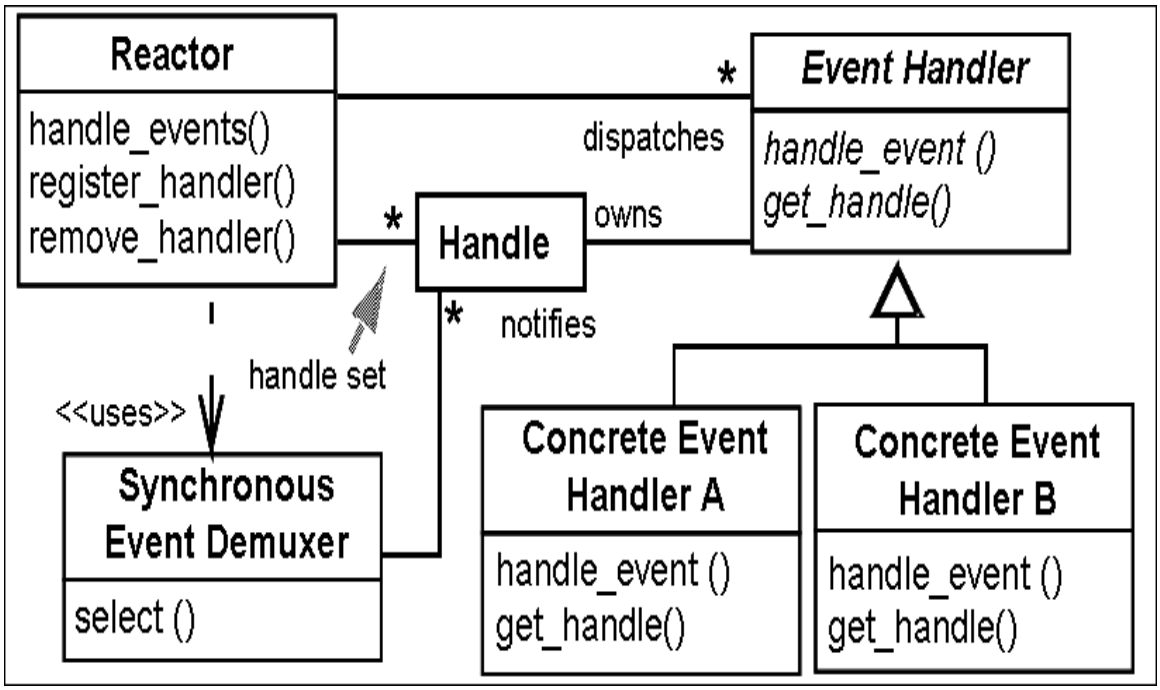


Figure IV.3: UML Diagram of the Reactor Pattern

Corresponding to the UML figure, Figure IV.4 illustrates the meta-model of the Reactor building block in the domain-specific POSAML.

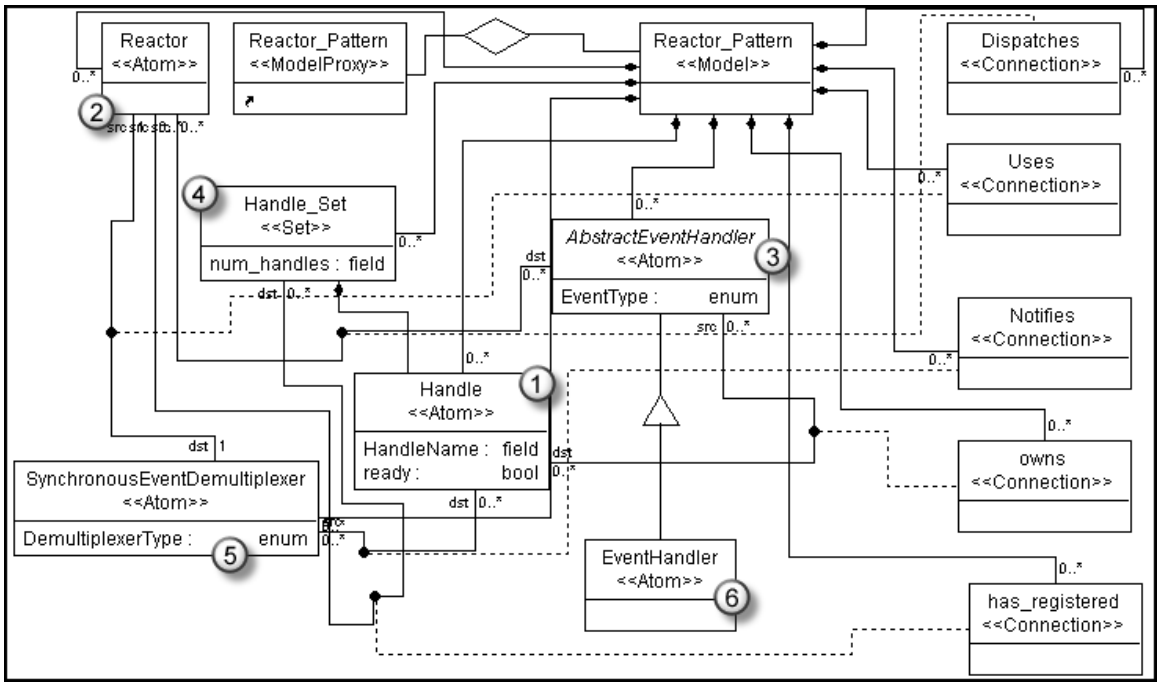


Figure IV.4: Meta-model of the Reactor Pattern

This meta-model enables the designer to model the following participants and their relationships in POSAML:

1. *Handle*: The handle uniquely identifies event sources such as network connections or open files. Whenever an event is generated by an event source, it is queued up on the handle for that source and marked as "ready."

2. *Reactor*: The reactor is the dispatching mechanism of the Reactor pattern. In response to an event, it dispatches the corresponding event handler for that event.

3. *Event Handler*: The event handlers are the entities which actually process the event. These are registered with the reactor and are dispatched by the reactor when the event for which they are registered occurs.

4. *Handle Set*: The registered handles form a set called the "Handle Set."

5. *Synchronous Event Demultiplexer*: This entity is actually implemented as a function call, such as `select()` or `WaitForMultipleObjects()` (in case of Windows-based systems). It waits for one or more indication events to occur, and then propagates these events to the reactor.

6. *Concrete Event Handlers*: The concrete event handlers specialize the generalized Event Handler. They are responsible for processing specific types of events, such as input data or timeouts.

A sample Reactor model corresponding to the UML diagram, modeled in the Structural View of POSAML, is shown in Figure IV.5.

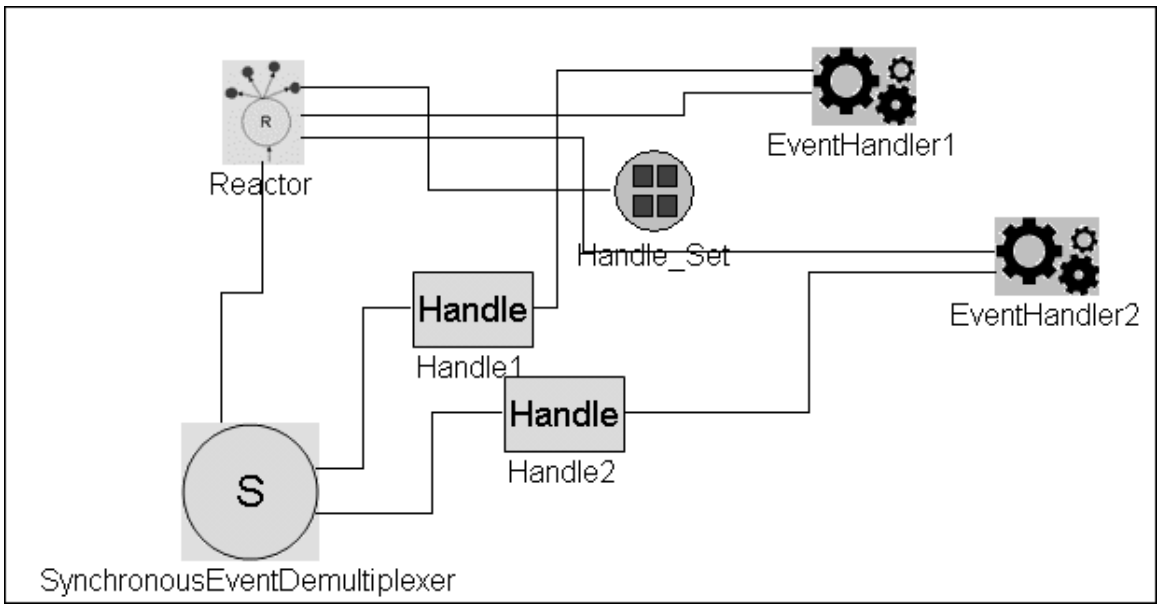


Figure IV.5: Model of the Reactor Pattern

In Figure IV.5 the designer has modeled two event handlers, corresponding to a handle set consisting of two handles. Both the 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.

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

IV.2 Modeling the Active Object Pattern

The Active Object pattern is used to decouple the execution of a method from its invocation [15]. This enhances concurrency and ensures that the response time for a client request in a client-server system is reduced. Figure IV.6 shows a UML diagram of the Active Object pattern.

Corresponding to the UML figure, Figure IV.7 illustrates the meta-model of the domain-specific POSAML.

The Active Object Pattern consists of the following participants:

1. *Client*: The client invokes a method on the Active Object.

2. *Proxy*: A proxy is an entity which provides interfaces that clients can invoke on the Active Object. When a client invokes a method defined by the proxy, the proxy forms a Method Request and inserts it into the Scheduler's Message Queue. A proxy executes a method in the client's thread of control.

3. *Method Request*: The method request is used to pass information such as the

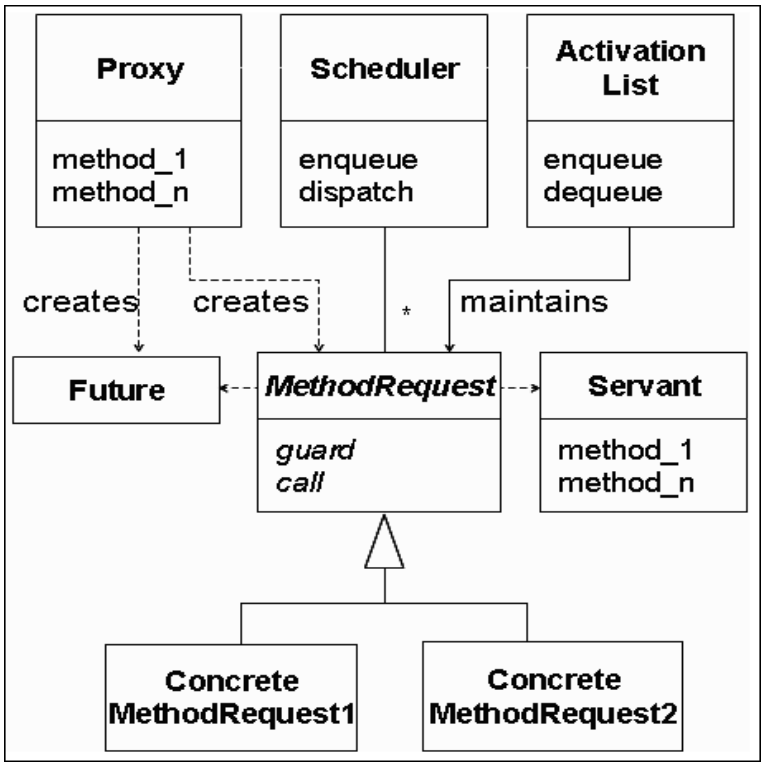


Figure IV.6: UML Diagram of the Active Object Pattern

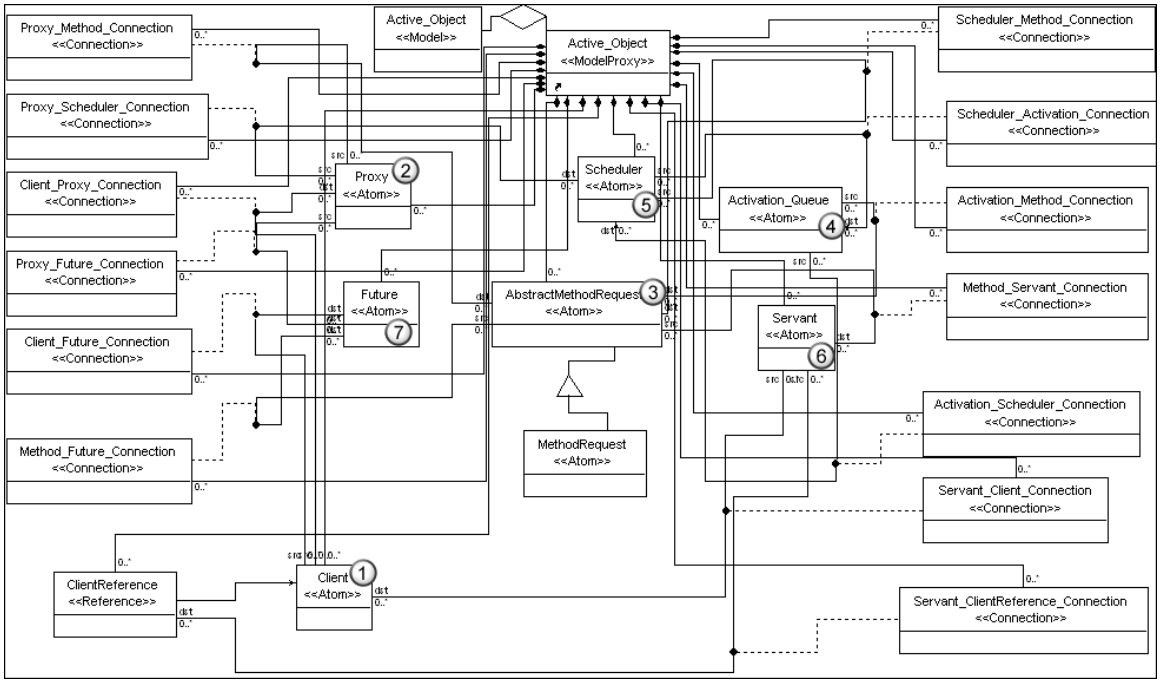


Figure IV.7: Meta-model of the Active Object Pattern

parameters of the method invocation to the Scheduler. In POSAML, an Abstract Method Request can be specialized by different concrete method requests which conform to the same interface.

4. *Activation Queue*: An activation queue contains the pending method requests which have been sent to the scheduler by the proxy. This is the entity which decouples the client thread from the servant thread so that both can execute concurrently.

5. *Scheduler*: A scheduler runs in its own thread. It decides which method requests to service, based on synchronization constraints. When a method request is to be serviced, the scheduler executes the servant that implements that method.

6. *Servant*: A servant actually implements the methods that are defined by the proxy and called by the client. The scheduler dispatches the servant when that particular method request is to be serviced. Servants therefore execute in the scheduler's thread of control.

7. *Future*: A future is a mechanism by which a client can receive return values back from the servant. The future is basically a place where the servant can store its results. The client can then access this future to retrieve the results.

The Active Object meta-model enables a system designer to model the participants of the Active Object Pattern and their inter-relationships. Figure IV.8 describes how the Producer-Consumer problem using the Active Object Pattern can be modeled in POSAML.

At the higher level, we have a combination of two producers and one consumer. Each of these can be modeled individually as Active Object Patterns in the Structural View of POSAML and configured in the Feature View. Future work in this regard

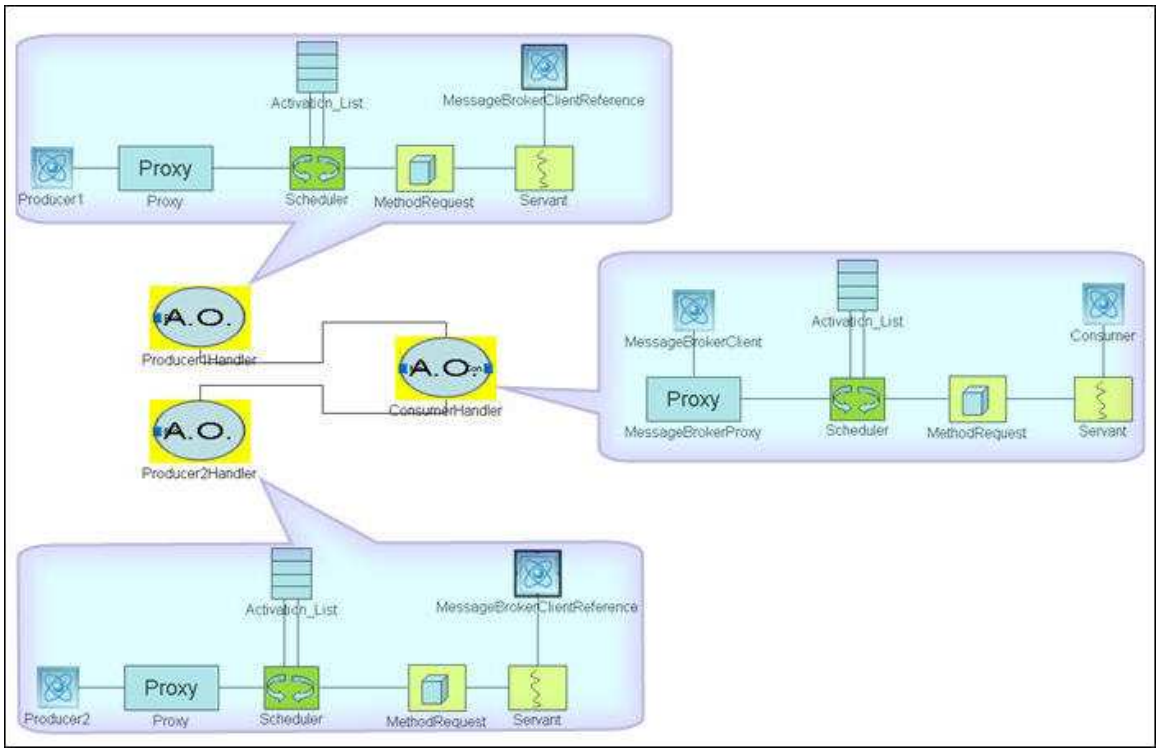


Figure IV.8: Model of a Producer-Consumer Problem Using POSAML

lies in allowing a modeler to model variants of the pattern, such as an integrated scheduler or a distributed active object [15].

CHAPTER V

PERFORMANCE EVALUATION USING POSAML

V.1 Simulation Aspect of POSAML

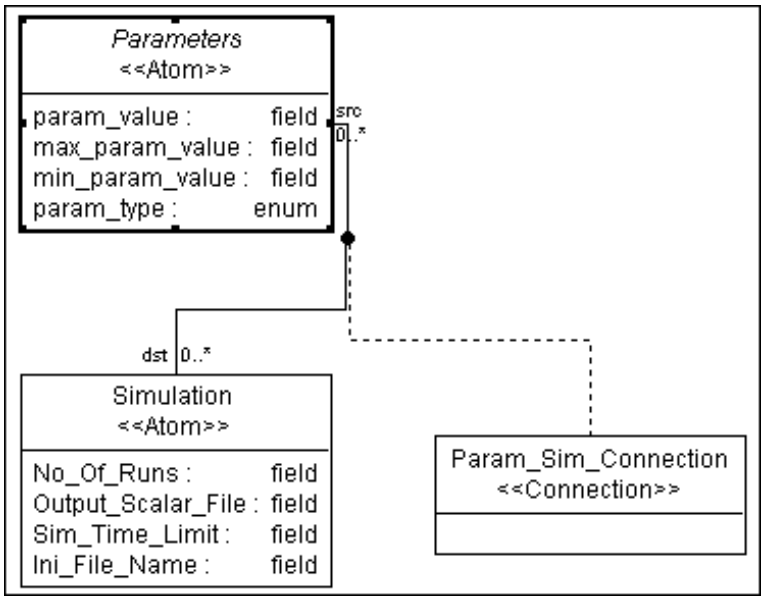
Modeling Simulation Parameters in POSAML The *Simulation Aspect* of POSAML enables a system designer to evaluate the system design. Every pattern has a simulation aspect associated with it. Figure V.1(a) illustrates the common simulation meta-model used across patterns, while Figure V.1(b) is a snippet of the reactor meta-model illustrating the simulation meta-model specific to the Reactor pattern.

The designer models simulation parameters for a specific pattern in the simulation aspect. The following values can be specified for each simulation parameter, as shown in Figure V.1(a):

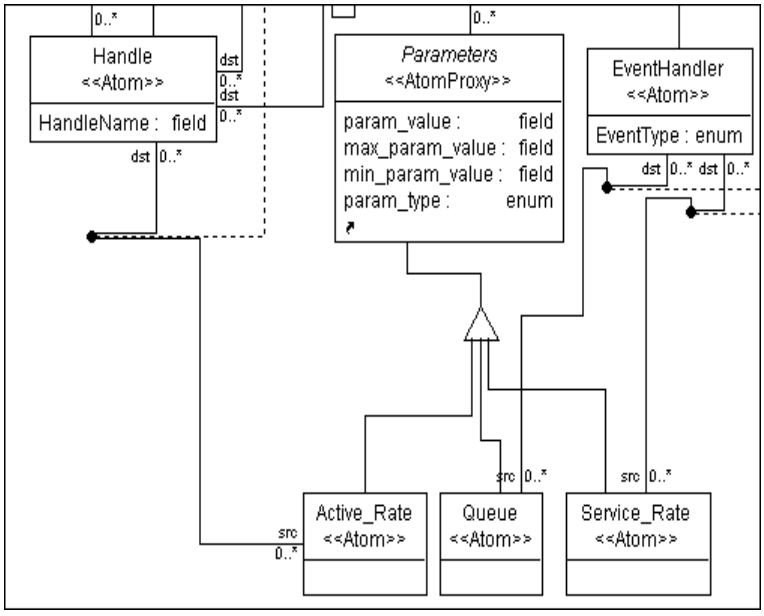
- Parameter Type: This specifies whether a parameter remains constant throughout the simulation or is varied over each simulation run.
- Parameter Value: If a parameter is constant, the parameter values is specified which stays constant throughout the simulation.
- Minimum and Maximum Parameter Values: If a parameter is variable across runs, then the minimum and maximum values specify the range of values that a parameter can take across runs.

A sample POSAML simulation model for the Reactor pattern is shown in Figure V.2.

The meta-model shown in Figure V.1(a) enables a modeler to specify the following top-level simulation options in the *Simulation* block in Figure V.2:



(a) Common Simulation Meta-Model



(b) Reactor-Specific Simulation Model

Figure V.1: Simulation Meta-Model

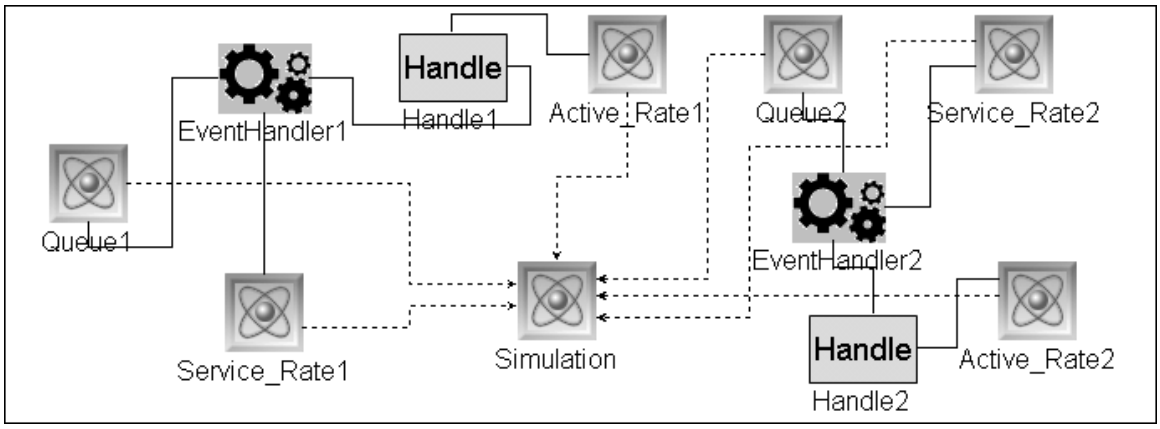


Figure V.2: Simulation Model for Reactor Pattern in POSAML

- Number of runs
- The simulation output file name
- The generated simulation parameter file name
- The simulation time limit for each run

From Figure V.1(b) and Figure V.2 we can see that the following simulation parameters specific to the Reactor pattern can be modeled:

- Queue: This parameter specifies the queue size for a particular event handler. By running simulations for various values of the queue size, the designer can determine the configuration with the most optimum queue size.
- Service Rate: This parameter specifies the service rate for a particular event handler. This parameter is important for the study of the effect of various types of event handlers on the system.
- Active Rate: This parameter specifies the rate at which a handle becomes active. This essentially describes the arrival pattern of events into the system.

Generative Capabilities of POSAML After a modeler has modeled the simulation parameters, the *Simulation Interpreter* is run. This interpreter is specific to the back-end simulator. Typically, the topology of the simulation is specified in a separate file from the simulation parameters. The simulation interpreter currently generates simulation parameterization files based on the model. A sample parameterization file for the OMNeT++ simulator, generated by the interpreter from the POSAML models given in Figure V.2, is given below:

```
[General]
preload-ned-files=*.ned
network=reactor_block
sim-time-limit=10000s

[Parameters]
reactor_block.num_handlers=2;
reactor_block.generator[1].lambda=0.4;
reactor_block.handler[0].mu=2;
reactor_block.handler[1].mu=2;
reactor_block.handler[0].queue_size=5;
reactor_block.handler[1].queue_size=5;

[Run 0]
reactor_block.generator[0].lambda=0.4;
[Run 1]
reactor_block.generator[0].lambda=0.6;
[Run 2]
reactor_block.generator[0].lambda=0.8;
```

```
[Run 3]
reactor_block.generator[0].lambda=1.0;
[Run 4]
reactor_block.generator[0].lambda=1.2;
[Run 5]
reactor_block.generator[0].lambda=1.4;
[Run 6]
reactor_block.generator[0].lambda=1.6;
[Run 7]
reactor_block.generator[0].lambda=1.8;
[Run 8]
reactor_block.generator[0].lambda=2.0;
```

This parameterization file is generated for an OMNeT++ simulation of the reactor pattern, as discussed in Chapter VI. The "simulation_time_limit" and the "number_of_runs" are generated from the *Simulation* block in the model shown in Figure V.2. The constant parameters modeled in Figure V.2 are generated under the "Parameters" section of the file. In the "Runs" section, each of the variable parameters are incrementally varied from *min_value* to *max_value* (as specified in the model) across *number_of_runs*. The interpreter maps the *Active_Rate*(i.e., the rate at which a handle becomes active) to "lambda", i.e., the rate at which a generator generates events. Similarly, *Service_Rate* from the model is mapped to "mu" for each event handler in the parameterization file. The "num_of_handlers" are generated from the number of event handlers modeled in the Reactor pattern in the Structural View of POSAML, as described in Section IV.1.

In a simulation library, writing the topology files is generally done only once. Different runs of the simulation are carried out by changing the simulation parameters. Therefore, auto-generating the parameterization file saves relatively more human effort than auto-generating the topology file itself. Hence the current work focuses mainly on auto-generating the simulation parameterization file. However, future work in this regard lies in auto-generating the entire topology of the simulation from the POSAML models.

V.2 Benchmarking Aspect of POSAML

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* [13]. These characteristics can be the metrics to measure, the workload such as the number of threads and the time required by an event handler in the Reactor Pattern to handle a request. Figure V.3 illustrates the Benchmarking meta-model of POSAML.

A sample model which can be constructed using this paradigm is shown in Figure V.4.

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 as either a uniform or an exponential distribution. These benchmarking characteristics can be exported using a "benchmarking interpreter" to provide inputs to an existing benchmarking library.

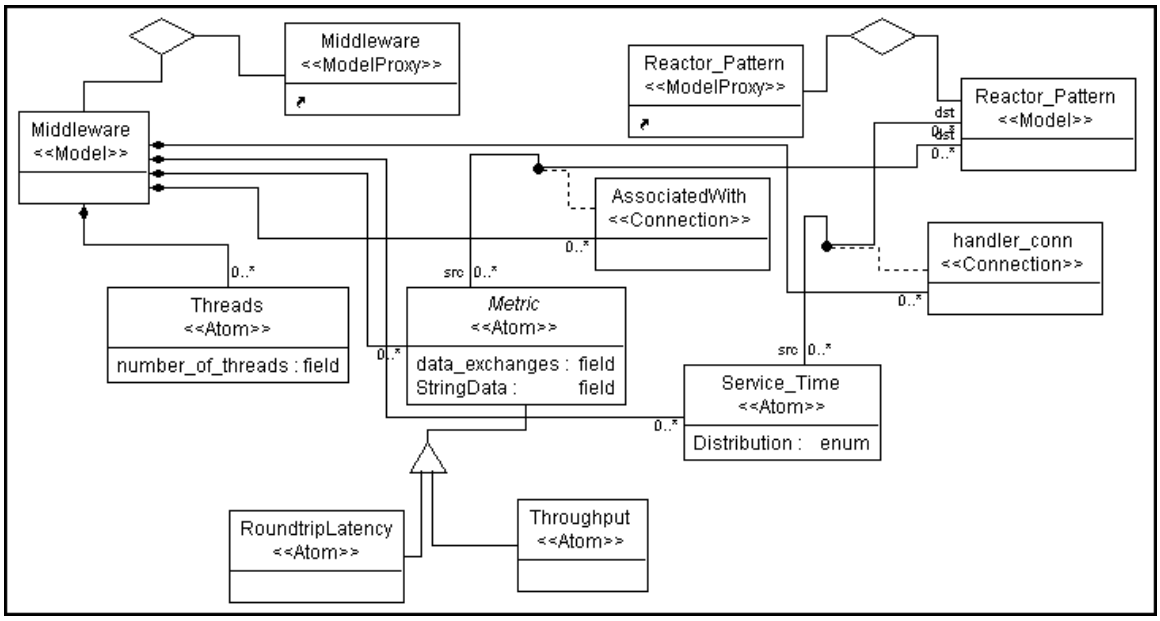


Figure V.3: Benchmarking Meta-Model

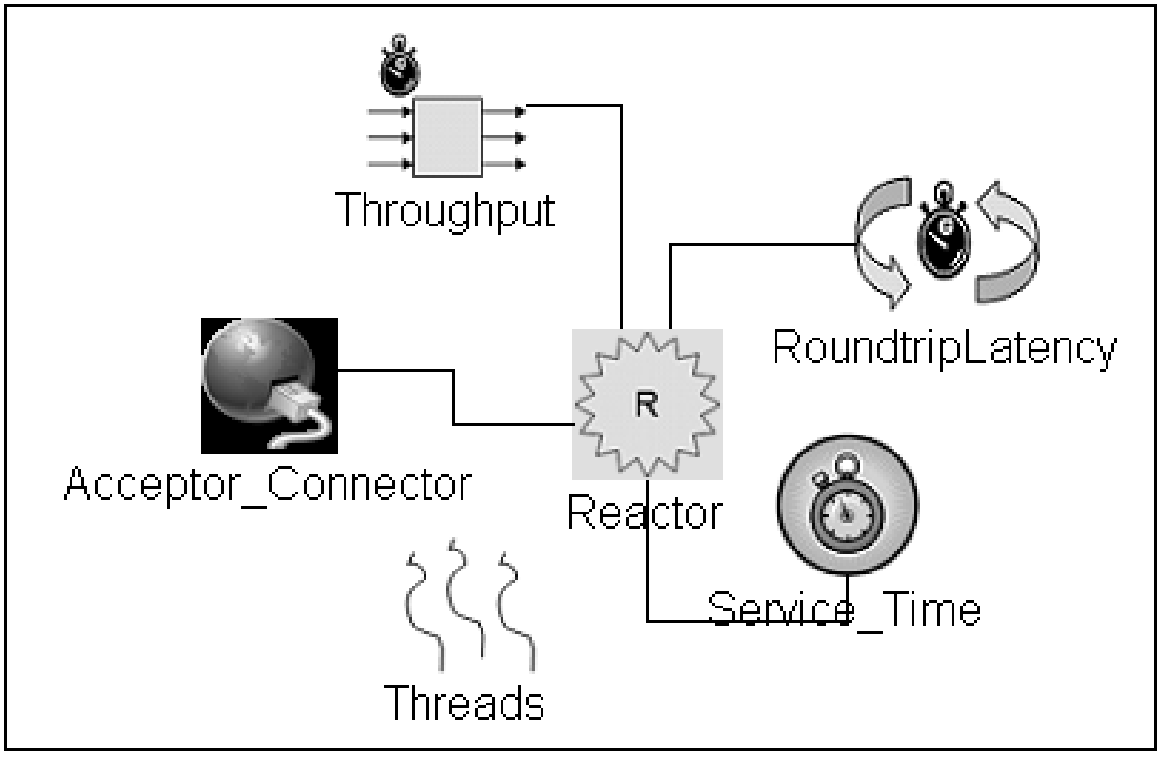


Figure V.4: Benchmarking Aspect

Generating Benchmarking Artifacts Using the Benchmarking Model 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). The benchmarking interpreter has to traverse along all three aspects of POSAML. It gathers pattern information from the Pattern Aspect, benchmarking information such as metrics from the Benchmarking Aspect and feature information such as type of Reactor or Acceptor end-points from the Feature Aspect. This interpreter stores this information in an XML file that is used by an existing benchmarking 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>
```

Future work in this regard lies in auto-generating benchmarking code, in addition to generating benchmarking parameters for existing benchmarking libraries.

CHAPTER VI

SIMULATION MODEL OF REACTOR PATTERN

This chapter describes the simulation and analysis of the Reactor building block found in most large-scale, distributed systems. After careful study OMNeT++ (`www.omnetpp.org`) was chosen as the back-end simulator because of its ease of use, flexible and modular architecture, parametric approach and open-source code base. OMNeT++ also has an advantage over other existing simulators in that it easily allows for the simulation of virtually any modular, event-driven system, and not just communication-network oriented systems.

OMNeT++ [22] follows a hierarchical architecture. At the lowest level of the hierarchy are simple modules which encapsulate behavior. These simple modules are represented by C++ classes. A compound module may be composed of simple as well as other compound modules. Modules communicate with each other via message-passing. An event is said to have occurred whenever a module sends/receives a message. A module may have parameters whose values are specified externally in an initialization file. These parameters can be varied in different simulation runs. In the context of middleware, these parameters can be used to simulate and analyze the effect of different middleware configuration options. Additional information about OMNeT++ can be found in the OMNeT++ User Manual [21].

VI.1 The Simulation Model

The simulation model [12] for the Reactor pattern is based on the structure of the Reactor as shown in Figure IV.3. The topology of the model is shown in Figure VI.1. This topology is specified in the .NED file of OMNeT++.

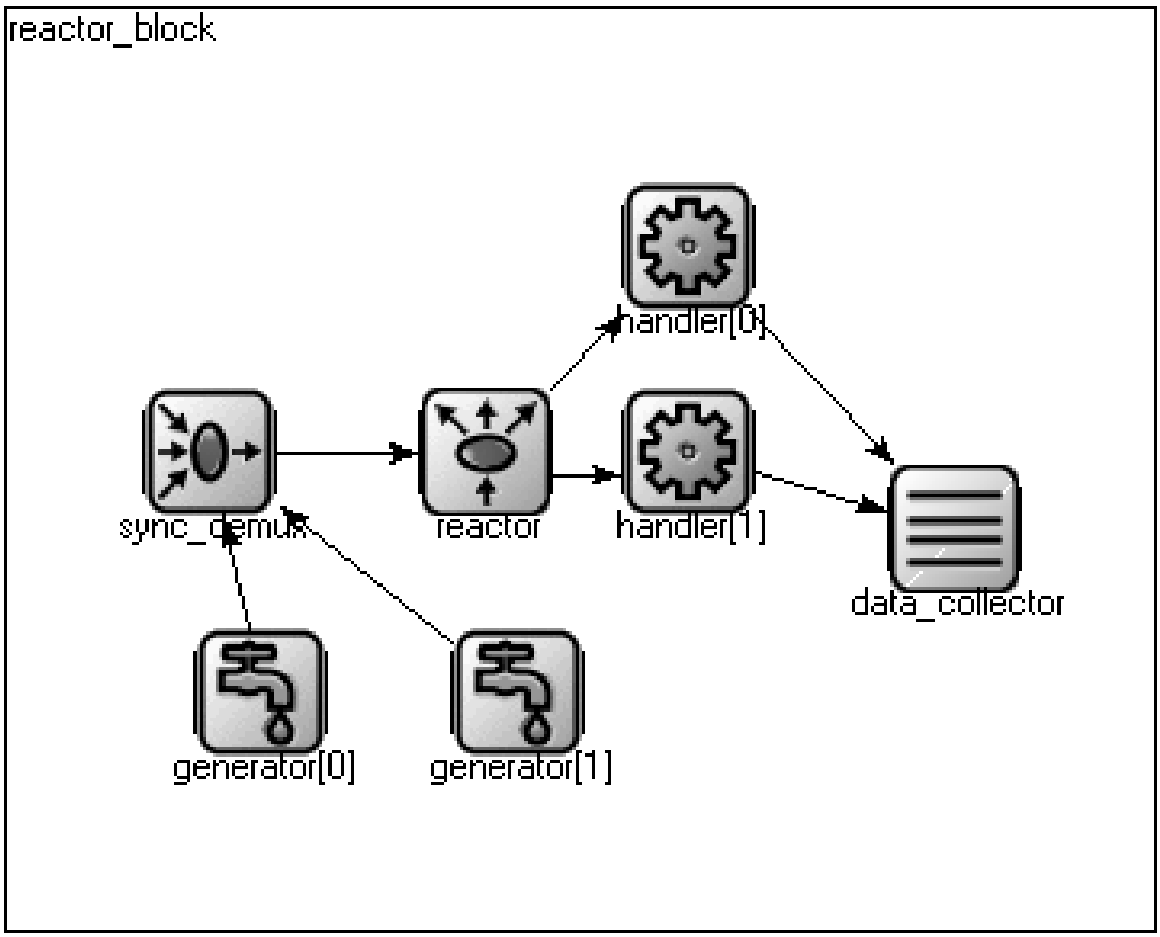


Figure VI.1: Simulation Model of Reactor in OMNeT++

The simulation model consists of the following blocks:

- *Event Generators*: Event generators are event sources, which generate events at a Poisson distribution rate λ. The number of event generators and their rates of event generation are parameterized values.

- *Synchronous Event Demultiplexer*: The synchronous event demultiplexer receives the events generated by the Generators. Depending on which generator generated the event, the synchronous event demultiplexer attaches an *Event_Type* value to the event and subsequently propagates the event to the Reactor.

- *Reactor*: Depending on the *Event_Type*, the reactor dispatches and activates the appropriate Event Handler by sending an event to that handler.

- *Event Handlers*: Each Event Handler has an exponentially distributed service time with rate μ. Each Event Handler also has a bounded queue associated with it with a maximum size of N. Upon receiving the dispatch event from the Reactor, the event is immediately handled if the queue is empty and no other event is being handled. If an event is currently being handled and the queue is not full, the incoming event is queued. If the queue is full, then the event is dropped. After an event has been handled, the event handler propagates it to the data collector. The event-handling process is simulated by scheduling the event to be propagated after a delay of *Service_Time* seconds. The number of event handlers as well as the service rate μ of each handler is a parameterized value and can be changed for each simulation run.

- *Data Collector*: The data collector acts as an event sink. It receives events sent by the Event Handlers. The data collector also calculates the throughput value and loss probability for each Event type.

The generators generate events at a Poisson distribution because the generated events represent the arrival pattern of events into the system, which is most commonly taken to be Poisson. Similarly, the service times of Event Handlers are exponentially distributed according to the most common service pattern. We have modeled

a bounded buffer for Event Handlers as most of the real-time, event-driven systems do not have the memory resources required for an infinite buffer.

VI.2 Statistics Collection

The following metrics [8] are measured during the simulation process:

- *Throughput(T)*: The throughput for each event type i is calculated by the data collector as the number of events of that type received by the collector divided by the simulation time at the end of the simulation run. The throughput metric is important for real-time event processing and distributed applications, such as on-line stock trading services.

- *Queue Length (Q)*: The queue length for each event type is recorded each time an event arrives for the event handler for that type. The queue length metric is significant for resource-constrained systems, such as RFID chips, that need to know the optimum buffer size to allocate for buffering events.

- *Loss Probability (L)*: The loss probability for an event type i is calculated by the data collector as the number of events sent by the event handler divided by the total number of events arriving in the event handler. This metric is significant for hard real-time systems where the loss of a control event would significantly affect the performance and even correctness of the system.

VI.3 Use of POSAML

While developing the simulation model manually, it was observed that the scalability of the model stood out as an important issue. As the number of event handlers and event generators increases, it becomes quite difficult to construct a correct simulation

model by hand. Similarly, if we add a few more patterns such as Acceptor-Connector to the simulation model, it will be extremely difficult to manage the entire model manually. The use of model-driven generative techniques for generating simulation models automatically are of great help in this regard . These techniques factor out some of the common tasks in simulation (such as adding new connections upon addition of a new handler). They also guard against any errors introduced by changes to the model.

The *Simulation Aspect* of POSAML has been used to achieve scalability in simulations. As described in V.1, a simulation initialization file for OMNeT++ simulations is generated from the Simulation Model of the Reactor Pattern. The input parameters are specified in this file and are read at runtime by the *OMNeT++* simulation environment for each set of simulation runs. This file therefore drives the simulations in an existing simulation library written for the Reactor Pattern.

VI.4 Simulation Results and Analysis

This section describes the results of simulating the reactor pattern in OMNeT++ by varying different parameters. The number of event generators, as well as the number of event handlers, is set to two. Table VI.1 lists out the input parameters, the performance metrics and their notations.

The initial values of the input parameters are shown in Table VI.2.

Effect of Arrival Rate For the first set of simulation runs, the effect of the arrival rate λ_0 on the throughput, mean queue length and probability of event loss was measured. As noted in the sample *omnetpp.ini* file, λ_0 was varied from 0.4 to 2.0 in steps of 0.2, while the other input parameters were kept constant at the values given in Table VI.2. The results are shown in Figure VI.2. It can be seen that as

(a) Parameters

Parameter	**Type 0**	**Type 1**
Arrival Rate	λ_0	λ_1
Service Rate	μ_0	μ_1
Maximum Buffer Length	N_0	N_1

(b) Metrics

Metric	**Type 0**	**Type 1**
Mean Queue Length	Q_0	Q_1
Throughput	T_0	T_1
Loss Probability	L_0	L_1

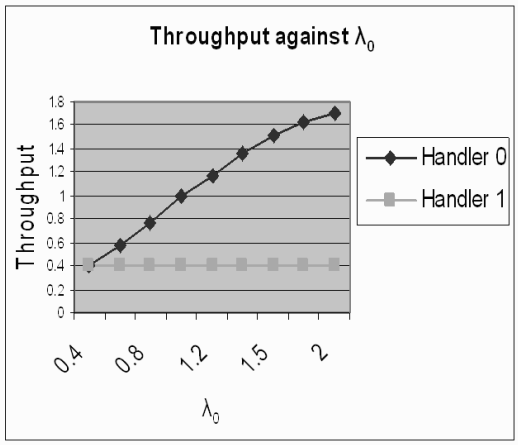
Table VI.1: Notations

Parameter	**Initial Value**
λ_0	0.4/s
λ_1	0.4/s
μ_0	2.0/s
μ_1	2.0/s
N_0	5
N_1	5

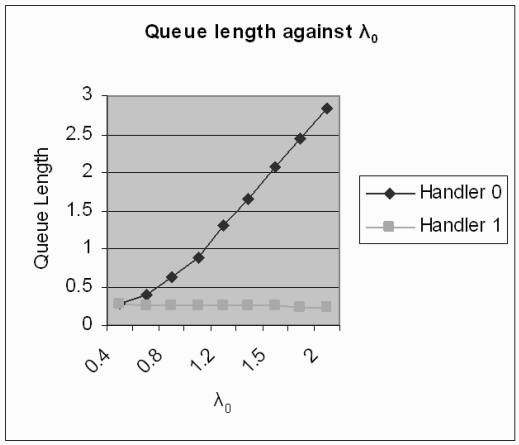
Table VI.2: Initial Set-Up

the arrival rate for Event Type 0 increases, the throughput for Type 0 also increases. The throughput for Type 1 remains constant, since arrival and processing of Type 0 is independent from Type 1.

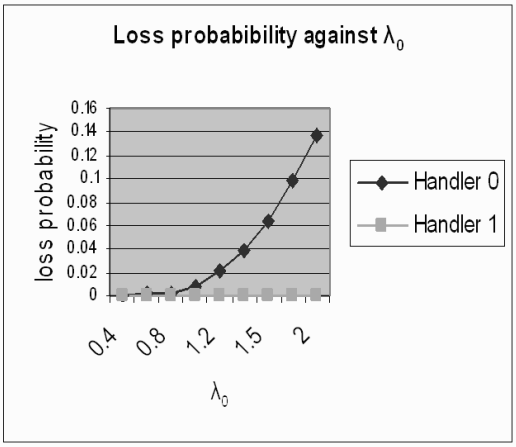
It can also be seen that as the arrival rate increases, the loss probabibility of Type 0 events increases, i.e, more Type 0 events are likely to be dropped. This can also be correlated to the increase in the mean queue length of Type 0 events.



(a) Throughput



(b) Mean Queue Length



(c) Loss Probability

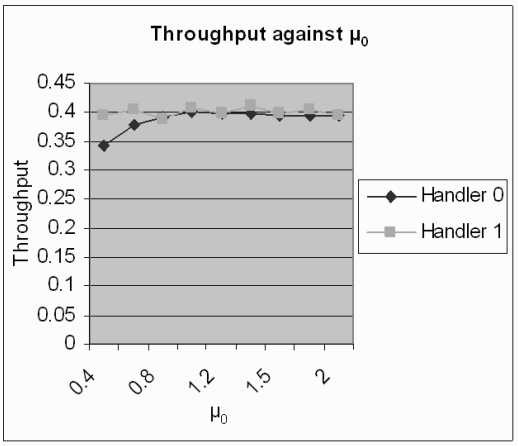
Figure VI.2: Effect of Arrival Rate

Effect of Service Time For the second set of simulation runs, the effect of the service rate μ_0 on the throughput, mean queue length and probability of event loss was measured. This time μ_0 was varied from 0.4/s to 2.0/s in steps of 0.2, while other input parameters were kept constant at the values given in Table VI.2. The results are shown in Figure VI.3. It can be seen that as μ_0 increases (i.e the time required by Handler 0 to process the events decreases) the throughput for Type 0 increases. The throughput for Type 1 remains constant, since arrival and processing of Type 0 is independent from that of Type 1.

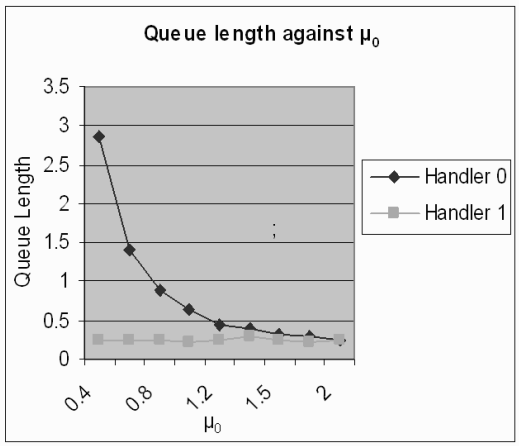
It can also be seen that the loss probability of Type 0 events decreases as service time decreases, since the number of queued events decrease with decrease in service time. This can also be deduced by the decrease in Mean Queue Length as seen in Figure VI.3(b). It can be seen from Figure VI.3(c) that the probability of loss increases rapidly when μ_0 drops below 0.8/s. This would be useful information for a system developer who needs to know the maximum allowable service time for a given loss probability.

It should be noted that the simulation model of the Event Handler does not take into account *how* the handler actually handles the event. The simulation therefore does not consider the effects of implementation artifacts such as the internal data structures used. For the purposes of simulation, the handler is considered to be a *black box*.

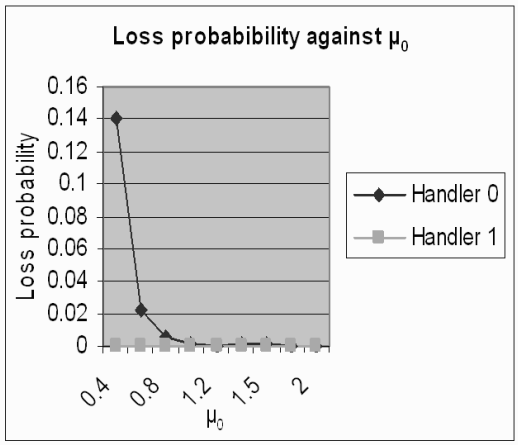
Effect of Maximum Buffer Size For the third set of simulation runs, the effect of the maximum buffer sizes N_0 and N_1 on the throughput, mean queue length and probability of event loss was measured. Other input parameters were kept constant at the values shown in Table VI.2. N_0 and N_1 were both kept at 1 for the first run. The results are illustrated in Figure VI.4. It can be seen that for a maximum buffer size of 1, all three metrics, i.e. throughput, mean queue length and loss probability



(a) Throughput



(b) Mean Queue Length

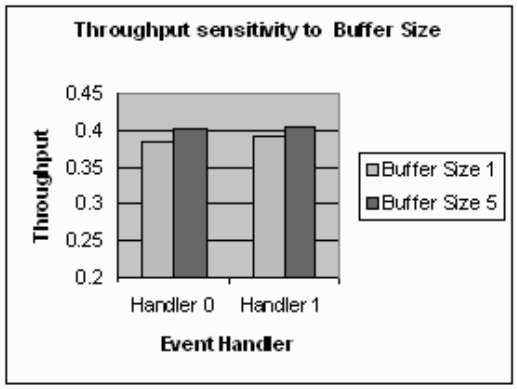


(c) Loss Probability

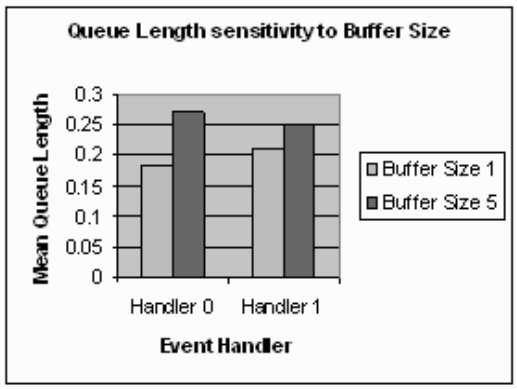
Figure VI.3: Effect of Service Time

were sub-optimal. In the second run, N_0 and N_1 were 5. The change in the results was remarkable, especially for the loss probability, as seen in Figure VI.4(c). The loss probability was almost zero for buffer size of 5. This indicates that the system was able to sustain the given arrival pattern. It also serves as a useful indicator to resource provisioners about the capacity of the system to handle higher event arrival rates for given buffer constraints.

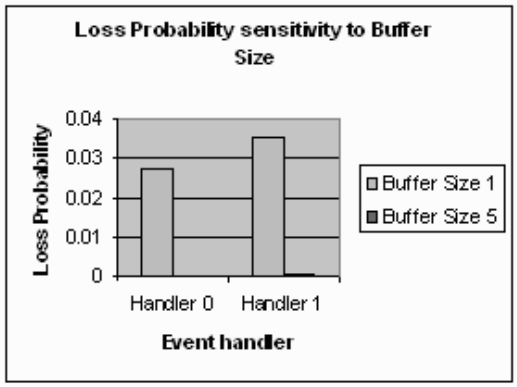
The throughput also increased, since considerably less number of events were being dropped. As expected, the mean queue length also increased with increase in maximum buffer size.



(a) Throughput



(b) Mean Queue Length



(c) Loss Probability

Figure VI.4: Effect of Maximum Buffer Size

These results provide some insightful information about the design of the system.

For example, from Figure VI.2 it can be seen that with increase in the arrival rate there is increase in the throughput, but the loss probability is increased as well. Therefore for higher arrival rates the designer could provide multi-level queues in his/her design to minimize the probability of loss. Also, by studying the effect of a combination of parameters such as maximum buffer size and the service rate, the designer could decide the most optimum configuration for the system.

CHAPTER VII

CONCLUSION

Large-scale, distributed systems present several challenges with respect to the accidental complexities associated with provisioning (i.e., configuration and QoS validation). In current practice, provisioning of such systems is performed through low-level, non intuitive and non reusable means. The manual nature of these techniques makes them error prone and tedious, and prohibits a system provisioner from rapidly exploring various design alternatives. POSAML addresses these challenges by providing a visual interface for designing and evaluating complex systems. POSAML allows various provisioning scenarios to be explored in a rapid manner that is platform-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.

The simulation model of the reactor pattern represents the first step in the bottom-up approach toward design-time analysis of pattern-based large-scale systems. Auto-generation of simulation files from POSAML models bridges the gap between model-driven structural design and design-time performance evaluation. Using POSAML and associated interpreters, changes in system structure or configuration are automatically reflected in the simulation files at the "click of a button". The simulation of the Reactor pattern also provides some insight into the event-handling behavior of middleware and other complex systems. This experience should prove useful in

the simulation and analysis of other building blocks as well as that of the composed system. Future work in this area will focus on building simulation libraries for other patterns as well as further exploring how to simulate combinations of patterns.

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

BIBLIOGRAPHY

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, NY, 1977.

[2] L.B. Arief and N.A. Speirs. A UML Tool for an Automatic Generation of Simulation Programs. In *Proceedings of the Second International Workshop on Software and Performance*, pages 71–76, Ottawa, Canada, September 2000. ACM.

[3] L. Bulej and P. Tuma. Current Trends in Middleware Benchmarking. In *Week of Doctoral Students 2003 conference*, pages 232 – 237, 2003.

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

[5] Douglas C. Schmidt and Frank Buschmann. Patterns, Frameworks, and Middleware: Their Synergistic Relationships. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003. IEEE/ACM.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

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

[8] Donald Gross. *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics. Wiley-Interscience, 3 edition, 1998.

[9] 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.

[10] T. Kalibera, L. Bulej, and P. Tuma. Quality Assurance in Performance: Evaluating Mono Benchmark Results. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 271–288. Springer, 2005.

[11] Dimple Kaul, Arundhati Kogekar, Aniruddha Gokhale, Jeff Gray, and Swapna Gokhale. Managing Variability in Middleware Provisioning Using Visual Modeling Languages. In *Proceedings of the Hawaii International Conference on System Sciences HICSS-40 (2007), Visual Interactions in Software Artifacts Minitrack, Software Technology Track*, Big Island, Hawaii, Jan 2007.

[12] Arundhati Kogekar and Aniruddha Gokhale. Performance Evaluation of the Reactor Pattern Using the OMNeT++ Simulator. In *Proceedings of the 44th Annual Southeast Conference*, Melbourne, FL, April 2006. ACM.

[13] Arundhati Kogekar, Dimple Kaul, Aniruddha Gokhale, Paul Vandal, Upsorn Praphamontripong, Swapna Gokhale, Jing Zhang, Yuehua Lin, and Jeff Gray. Model-driven Generative Techniques for Scalable Performabality Analysis of Distributed Systems. In *Proceedings of the NSF NGS Workshop, International Conference on Parallel and Distributed Processing Symposium (IPDPS) 2006*, Rhodes Island, Greece, April 2006. IEEE.

[14] A. S. Krishna, E. Turkay, A. Gokhale, and D. C. Schmidt. CCMPerf: A Benchmarking Tool for CORBA Component Model. Available from: `http://citeseer.ist.psu.edu/639589.html`.

[15] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Proceedings of the* 2^{nd} *Annual Conference on the Pattern Languages of Programs*, pages 1–7, Monticello, Illinois, September 1995.

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

[17] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.

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

[19] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[20] P. Tuma and A. Buble. Open CORBA Benchmarking, 2001. Available from: `http://citeseer.ist.psu.edu/tuma01open.html`.

[21] A. Varga. *The OMNeT++ User Manual*, 1997. Available from: `http://www.omnetpp.org/doc/manual/usman.html`.

[22] Andrs Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.