Abstract

Standardized middleware forming general-purpose application hosting platforms provide several different building blocks with numerous functionalities and capabilities. These blocks can be assembled and configured in various combinations to provide specialized middleware stacks to suit diverse quality of service requirements for assorted application contexts. Choosing the right set of building blocks and their configurations for the needs of a specific application becomes the responsibility of the application developers and integrators. This selection process, however, tends to use ad hoc techniques based on trial-and-error and runtime testing of the configured platform. Such an approach adversely impacts both time-to-market and opportunities for reuse, and cannot scale given the variability introduced by the different building blocks. A desirable solution to this problem involves design-time performance evaluation of the composition of building blocks. This paper demonstrates how generative programming combined with model-driven development can play an important role in addressing this challenge. We show how our approach provides an automated and scalable solution to this problem by offering advantages over manual provisioning of middleware.

Categories and Subject Descriptors D.2.4 (Software) | [S/W Engineering]: (S/W Verification)

General Terms Validation

Keywords performance models, simulations, model-driven generative technologies

1. Introduction

Society is increasingly reliant on a wide array of services in different domains (e.g., electric power grid, mobile communications, patient records management, entertainment) provided by distributed networked systems. Rapid advances in networking, hardware and software technologies, and increased competition, is requiring service providers to rapidly introduce newer services to the market. Service providers, however, now have to deal with two major forces. On the one hand they must reduce the time to market while on the other hand they must ensure that the services provide high performance and are dependable.

Recent trends in software design indicate that the use of prefabricated artifacts for software development is on the rise. A paradigm called Software Factories [8] 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 commercial off-the-shelf (COTS) software components that one can acquire from different vendors and assemble them to deploy large-scale applications.

A key enabler supporting this vision has been standardized component middleware [23], such as J2EE [30], .NET [16] and the CORBA Component Model (CCM) [19], which comprises software layers that provide platform-independent execution semantics and reusable services that coordinate how application components are composed and interoperate.

In order to support a wide range of applications in different domains, standardized general-purpose middleware provide a number of highly configurable building block frameworks that can be composed with each other to form specialized middleware hosting platforms for the applications. Each building block provides unique capabilities (e.g., concurrency, messaging, connection management, prioritized request handling) and are designed based on proven software design patterns [5]. Middleware building blocks thus provide the means to rapidly compose, configure, assemble and deploy the platforms to host the applications that service providers want to rapidly bring to market.

Choosing the right set of building blocks to use and their configurations is the responsibility of the application developers and integrators. They are required to ascertain whether the choice of building blocks, and their configurations and compositions will provide the required levels of quality of service (QoS), which includes performance and dependability, to the applications. This process often ends up being a trial-and-error exercise with validation of the desired QoS left to rigorous runtime empirical benchmarking and testing. This ad hoc iterative process is detrimental towards meeting the rapid time-to-market goals. Moreover, such techniques provide no incentives and opportunities for reuse in future products with similar QoS needs.

A desired solution to this challenge should substantially alleviate and potentially eliminate the need for this ad hoc, iterative process while promoting reuse. One such approach hinges on carrying out the validation of the observed QoS of building block compositions of the hosting platforms earlier in the application development lifecycle [34] (i.e., at design time). Such an analysis, though not as precise as runtime benchmarking, would still provide sufficiently accurate estimates and bounds that can help systems integrators make appropriate choices in provisioning the hosting platforms.

Design-time QoS analysis is almost invariably based on a model that represents application behavior. For example, in order to enable model-based QoS analysis of event-driven applications, it is necessary to build a model of the underlying event demultiplex-
ing framework that is ubiquitous in such applications. In the past, design-time QoS analysis has been conducted for a specific application. In some cases, analysis methodology for generic architecture styles that are applicable at the level of the application architecture have been developed [35].

The greatest benefit of this approach is realized when such models capture the structure and behavior of the highly configurable middleware building blocks. An engineer can use the application QoS requirements to guide the provisioning of middleware hosting platforms. Support for automated QoS analysis offers systems integrators instant feedback on the design choices they have made. Model-driven engineering (MDE) [13, 24] based on the use of domain-specific languages and generative programming plays a vital role in this approach.

This paper describes such an approach describing the domain-specific modeling languages we have developed to capture middleware building block structure and behavior. The structural aspects of middleware building blocks are captured in a language called POSAML (Patterns Oriented Software Architecture Modeling Language), which models software patterns [5, 26] to capture the structure and variabilities in the middleware building blocks. The behavioral aspects of the building blocks are captured as stochastic reward nets (SRNs) [17] in our SRNML (Stochastic Reward Net Modeling Language), which enables us to use Petri Net-based solvers to analyze our design choices. The generative aspects of our approach enable the automation of the middleware composition and configuration, and its design-time QoS validation process.

The rest of the paper is organized as follows: Section 2 discusses the accidental complexities in manually developing and conducting the QoS validation process, which are resolved using MDE-based generative techniques; Section 3 describes our MDE approach describing the domain-specific modeling languages and generative aspects of our research; Section 4 illustrates the use of our MDE generative approach for QoS validation of a representative event-driven system; Section 5 discusses our work comparing it to similar research described in the literature; and finally, Section 6 provides concluding remarks outlining the lessons learned from this study, as well as directions for future research.

2. Challenges in Design-time QoS Analysis

This section describes the process of conducting design-time QoS analysis of the design choices made by the system integrator for provisioning the middleware. Many different design-time QoS analysis approaches exist, including various analytical and simulation approaches [21]. Our design-time validation approach uses an analytical method involving constructing Stochastic Reward Net (SRN) [17] models of middleware building blocks and their compositions, and solving these models for the expected workloads and service times.

Our objective in this section is to underscore the fact that manually developing and conducting design-time analysis has its limitations stemming from scalability issues and having to deal with the variability in the middleware building blocks. Our approach to resolving these challenges is based on model-driven engineering (MDE) [13, 24] comprising the use of domain-specific modeling languages (DSMLs) and generative programming [4].

We highlight these challenges by illustrating the steps in developing a SRN model for a well known middleware building block that provides synchronous event handling capabilities. Such a building block invariably incorporates the Reactor pattern [26].

Initially we describe the characteristics of the reactor pattern and the relevant performance measures. A SRN model of the reactor pattern is then presented along with a discussion of how the performance measures can be obtained by assigning reward rates at the net level. We conclude the section with a discussion of the scalability challenges involved in constructing the performance models.

2.1 Modeling Considerations for the Reactor-based Building Block

The QoS analysis models for building blocks must consider both the structural and the behavioral aspects. Figure 1 depicts a typical event demultiplexing and dispatching structure of the reactor pattern [26]. The application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening to incoming events. On the occurrence of an event, the demultiplexer dispatches the event by making a callback to the correct application-supplied event handler. This is the idea behind the reactor pattern, which provides synchronous event demultiplexing and dispatching capabilities.

Figure 2 illustrates the reactor pattern behavior. We categorize these behavioral dynamics into two phases described below, which we leverage in our SRN model.

1. **Registration phase**: In this phase all the event handlers register with the reactor associating themselves with a particular event type they are interested in. Event types usually supported by a reactor are input, output, timeout and exceptions. The reactor will maintain a set of handles corresponding to each handler registered with it.
2. Snapshot phase: Once the event handlers have completed their registration, the main thread of control is passed to the reactor, which in turn listens for events to occur. A snapshot is then an instance in time wherein a reactor determines all the event handlers that are enabled at that instant. For all the event handlers that are enabled in a given snapshot, the reactor proceeds to service each event by invoking the associated event handler. There could be different strategies to handle these events. For example, a reactor could handle all the enabled events sequentially in a single thread or could hand it over to worker threads in a thread pool. After all the events are processed, the reactor proceeds to take the next snapshot of the system.

2.2 SRN Model of the Reactor Pattern

Having considered the structural and the behavioral aspects of a building block, the next step is to construct the model. As mentioned earlier, we use stochastic reward nets (SRNs) [17] as the analytical techniques used for design-time QoS analysis. SRNs represent a powerful modeling technique that is concise in its specification and whose form is closer to a designer’s intuition about what a model should look like. Since a SRN specification is closer to a designer’s intuition of system behavior, it is also easier to transfer the results obtained from solving the models and interpret them in terms of the entities that exist in the system being modeled. An overview of SRNs can be found in [20]. Stochastic reward nets have been extensively used for performance, reliability and performability analysis of a variety of systems [10–12, 17, 22, 29].

In our SRN model of the reactor, we consider a single-threaded, select-based implementation of the reactor pattern with the following characteristics:

- The reactor receives two types of input events with one event handler for each type of event registered with the reactor.
- Each event type has a separate queue, which holds the incoming events of that type. The buffer capacity for the queue of type #1 events is denoted $N_1$ and of type #2 events is denoted $N_2$.
- Event arrivals for both types of events follow a Poisson distribution with rates $\lambda_1$ and $\lambda_2$, while the service times of the events are exponentially distributed with rates $\mu_1$ and $\mu_2$.
- In a given snapshot, if the event handles corresponding to both the event types are enabled, then they are serviced in no particular order. In other words, the order in which the events are handled is non-deterministic.

We now describe the steps in constructing the SRN model of our representative middleware building block.

**Description of the net:** Figure 3 shows the SRN model for the Reactor pattern with the characteristics described in Section 2.1. Table 1 summarizes the enabling/guard functions for the transitions in the net. The net on the left-hand side models the arrivals, queuing and service of the two types of events. Transitions $A_1$ and $A_2$ represent the arrival of the events of type #1 and #2, respectively. Places $B_1$ and $B_2$ represent the queue for the two types of events. Transitions $Sn_1$ and $Sn_2$ are intermediate transitions that are enabled when a snapshot is taken. Places $S_1$ and $S_2$ represent the enabled handles of the two types of events, whereas transitions $Sr_1$ and $Sr_2$ represent the execution of the enabled event handlers of the two types of events. An inhibitor arc from place $B_1$ to transition $A_1$ with multiplicity $N_1$ prevents the firing of transition $A_1$ when there are $N_1$ tokens in place $B_1$. The presence of $N_1$ tokens in place $B_1$ indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place $B_2$ to transition $A_2$ achieves the same purpose for type #2 events. The inhibitor arc from place $S_1$ to transition $Sr_2$ prevents the firing of transition $Sr_2$ when there is a token in place $S_1$. This models the prioritized service for an event of type #1 over event of type #2 in a given snapshot.

The net on the right of Figure 3 models the process of taking successive snapshots and prioritized service of the event handle corresponding to type #1 events in each snapshot. Transition $Sn_1$ is enabled when there is a token in place $StSnSht$, at least one token in place $B_1$, and no tokens in place $S_1$. Similarly, transition $Sr_2$ is enabled when there is a token in place $StSnSht$, at least one token in place $B_2$, and no tokens in place $S_2$. Transition $T_{EndSnSht}$ is enabled when there is a token in either one of the places $S_1$ and $S_2$, and the firing of this transition deposits a token in place $SnShtInProg$.

The presence of a token in the place $SnShtInProg$ indicates that the event handles that were enabled in the current snapshot are being serviced. After these event handles complete execution, the current snapshot is complete and it is time to take another snapshot. This is accomplished by enabling the transition $T_{EndSnSht}$. Transition $T_{EndSnSht}$ is enabled when there are no tokens in both places $S_1$ and $S_2$. Firing of the transition $T_{EndSnSht}$ deposits a token in place $StSnSht$, indicating that the service of the enabled handles in the present snapshot is complete, which marks the initiation of the next snapshot.

**Assignment of reward rates:** The performance measures can be computed by assigning reward rates at the net level as summarized in Table 2. The throughputs of events of type #1 and #2 denoted $T_1$ and $T_2$ respectively are given by the rate at which transitions $Sr_1$ and $Sr_2$ fire. The queue lengths of the events denoted $Q_1$ and $Q_2$ are given by the number of tokens in places $B_1$ and $B_2$ respectively. The loss probability of type #1 events denoted $L_1$ is given by the probability of $N_1$ tokens in place $B_1$. Similarly, the loss probability of type #2 events denoted $L_2$ is given by the probability of $N_2$ tokens in place $B_2$. The reward rates to obtain the optimistic and pessimistic response times were derived [31,32], using the tagged customer approach [15].

**Table 2. Reward assignments to obtain performance measures**

<table>
<thead>
<tr>
<th>Performance metric</th>
<th>Reward rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>return rate($Sr_1$)</td>
</tr>
<tr>
<td>$T_2$</td>
<td>return rate($Sr_2$)</td>
</tr>
<tr>
<td>$Q_1$</td>
<td>return (# $B_1$)</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>return (# $B_2$)</td>
</tr>
<tr>
<td>$L_1$</td>
<td>return (# $B_1 :: N_1?1: 0$)</td>
</tr>
<tr>
<td>$L_2$</td>
<td>return (# $B_2 :: N_2?1: 0$)</td>
</tr>
</tbody>
</table>

2.3 Dealing with Middleware Variability

The previous section describes the manual process of constructing a SRN model for the reactor-based building block and assigning reward rates to obtain the different performance measures. As mentioned previously, this process requires substantial effort. Moreover, the effort expended serves to solve a specific problem at hand with almost no opportunity for reuse. This approach does not scale when QoS validation models have to account for entire middleware stacks composed out of different building blocks. Middleware frameworks demonstrate substantial variability that make the problem of QoS analysis challenging. The challenges in the following sub-sections must be addressed in order to automate and scale the design-time QoS analysis process.
2.3.1 Per Building Block Variability

Middleware developers provide numerous configuration options to customize the behavior of individual building blocks. This flexibility incurs significant variability in design choices and hence affects the performance delivered by each building block. For example, many different variations of the reactor pattern are possible depending on the configuration parameters used. These variations stem from the different event demultiplexing and event handling strategies used in a reactor. For example, in network-centric applications, networking events can be demultiplexed using operating system calls, such as `select` or `poll`. For graphical user interfaces (GUIs), these events could be due primarily to mouse clicks and can be handled by GUI frameworks like Qt or Tk. On the other hand, the event handling mechanisms could involve a single thread of control that demultiplexes and handles an event, or each event could be handled concurrently using worker threads in a thread pool or by thread on demand.

Other variations stem from the number of event types handled, the buffer space available for queuing events, input workloads and event service rates. To enable design-time QoS validation for an application employing a variant of the reactor pattern, the SRN model of the variant needs to be constructed manually. This process is cumbersome, tedious and time-consuming.

2.3.2 Compositional Variability

The variability challenges outlined above are further complicated when systems are composed of several middleware building blocks. Developers must decide the set of building blocks to use in the system composition and also ensure that the customizations to individual building blocks are compatible with each other in the vertical composition. This approach ensures that the desired end-to-end performance and functional requirements are met. The choice of blocks selected are driven by various factors, including the context in which the 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 systems, or throughput for enterprise systems.

Figure 3. SRN model for the Reactor pattern

<table>
<thead>
<tr>
<th>Transition</th>
<th>Guard function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sn1</td>
<td>((#StSnpShot == 1) &amp;&amp; (#B1 &gt;= 1) &amp;&amp; (#S1 == 0)) ? 1 : 0</td>
</tr>
<tr>
<td>Sn2</td>
<td>((#StSnpShot == 1) &amp;&amp; (#B2 &gt;= 1) &amp;&amp; (#S2 == 0)) ? 1 : 0</td>
</tr>
<tr>
<td>T_SrvSnpSht</td>
<td>((#S1 == 1)</td>
</tr>
<tr>
<td>T_EndSnpSht</td>
<td>((#S1 == 0) &amp;&amp; (#S2 == 0)) ? 1 : 0</td>
</tr>
</tbody>
</table>

Table 1. Enabling/Guard functions

Figure 4 illustrates a family of interacting patterns forming a pattern language [2] for middleware frameworks. Patterns such as the Active Object (not shown) and Leader-Follower [26], provide alternate approaches to concurrency with each solution 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.

Figure 4. Middleware Patterns and Pattern Languages
3. Automating the QoS Modeling and Analysis Process

The previous sections described how a SRN model for QoS analysis of a building block used in middleware provisioning can be developed. The process described until now focuses on manually developing these models and the associated input scripts used by the model solvers, such as Stochastic Petri Net Package (SPNP) [9]. As the middleware provisioning complexity grows, however, it becomes tedious and error prone to develop these models manually since the accidental complexities involved in modeling the system performance characteristics and the input script sizes for the model solver grow substantially.

As explained earlier there is substantial variability manifested in middleware frameworks. Even if tools are developed to alleviate these challenges, such tools should make it possible for a software architect, who is a non-expert in the tools and techniques of performance analysis, to be able to use the model. To enable these dual objectives, it is necessary to encapsulate the QoS analysis modeling approach described in the earlier sections into user-friendly tools that automate the QoS analysis task and enable model scalability to large systems.

In this section, we describe a model-driven engineering (MDE) [18, 24, 33] approach, which allows the user to automate the QoS analysis process and scale the smaller base models to larger systems. Our approach provides capabilities to capture both the structural and behavioral aspects of the middleware building blocks, which can then be composed with each other. These capabilities are provided in the form of domain-specific modeling languages (DSMLs). The generative capabilities of these DSMLs enable synthesis of artifacts, such as configuration parameters for the middleware and scripts for the SRN solvers. These capabilities are described in the remainder of this section.

3.1 Modeling Languages for QoS Analysis

In this section we describe the ideas based on a model-driven [18] generative programming [4] framework we are developing to address the aforementioned challenges. Our modeling framework comprises modeling languages we have developed using the Generic Modeling Environment (GME) [14]; GME is a tool that enables domain experts to develop visual modeling languages and generative tools associated with those languages. The modeling languages in GME are represented as metamodels. A metamodel in GME depicts a class diagram using UML-like constructs showcasing the elements of the modeling language and how they are associated with each other. The GME environment can then be used by application developers to model patterns that conform to the syntax and semantics of the modeling language captured in the metamodels.

We have developed two modeling languages that provide the visual syntactic and semantic elements required to model the systems compositions and their performance models.¹ The first language is called POSAML (Patterns-Oriented Software Architecture Modeling Language), which models the patterns-based building blocks of contemporary middleware frameworks and represents the structural and compositional capabilities of our framework. Details of individual patterns are described in the POSA [26] pattern language. The POSA pattern language is a vocabulary describing a set of related patterns used to develop network services.

The other modeling language is called SRNML (Stochastic Reward Net Modeling Language), which enables a user to model the behavior of individual patterns-based building blocks as a SRN. SRNML provides all the syntactic and semantic elements of SRNs, which enables a developer to model the behavior of a patterns-based building block.

3.2 Structural Modeling using POSAML

Architectural patterns present in middleware systems are discussed extensively in the POSA design patterns book [26]. Our pattern modeling language, based on patterns described in this book, is called the Pattern Oriented Software Architecture Modeling Language (POSAML). This DSML allows composition of patterns-based building blocks to create specialized middleware stacks for applications. The variability challenges in the middleware stacks and their QoS validation is addressed via three synergistic aspects (i.e., artifacts) of the POSAML language:

1. Pattern Aspect: The pattern aspect is where a system modeler can compose and model the various patterns in the middleware.
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. The Feature Aspect is described in depth in later sections.
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 interpreter. This file can be used by an existing benchmarking library.

Figure 5 shows the metamodel for the top-most view of pattern modeling. This metamodel consists of the individual pattern models, and specifies how they can be connected to each other. In addition to this, this metamodel also defines the three aspects of POSAML, namely Pattern, Feature and Benchmarking.

![Figure 5. Top-level metamodel of Middleware in POSAML](image)

3.2.1 Building Block Modeling in POSAML

Figure 6 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. 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.

Built-in constraint checking in POSAML enables correct by construction structure of system compositions since any erroneous associations between different patterns are flagged as an error at modeling time. Moreover, the feature modeling view and benchmarking view of the language also flags errors made by a model engineer. For example, if a model engineer chooses a single threaded

---

¹ Formal descriptions of these modeling languages is beyond the scope of this paper.
reactor demultiplexing strategy in the feature view, but provides a thread pool configuration in the benchmarking view, then the constraint checker flags this condition as an error.

3.2.2 Addressing Middleware Variability via Feature Modeling in POSAML

A feature model [4] is an abstraction of a family of systems in a particular domain capturing commonalities and variabilities among the members of the family. In POSAML, the 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 provides a structural representation of different possible middleware pattern properties. In our case, feature modeling comprises modeling several non-functional and QoS requirements, such as the choice of network transport, listening end points, multi-threaded requirements and periodicity of requests represented as higher-level artifacts. This enables 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 model.

Developing a feature model as an instance of a metamodel involves defining valid entities and connections among those entities. All of the features are pattern specific. For example Concurrenc,

Inhibitor Arc

Immediate Transition

Inhibitor Arc

Immediate Transition

Place

Immediate Transition

Transition

Input Arc

Output Arc

Reactor Pattern

Thread Queue

AcceptoConnector

Concurrenc

End_Points

Thread Queue

Reactor Pattern

Figure 6. Overview of POSAML

Reactor Type (such as thread Pool, multi-threaded select or wait-for-multiple-objects) and the type of Thread Queue with respect to the Reactor Pattern. The End-Point feature is used to specify the port and IP address that the Acceptor-Connector Pattern can use. After the feature modeling part is completed, the next step is to construct a model interpreter to transform pattern-specific features into a configuration file that is used by the middleware. If features are not selected from the model, default values of these features will be picked. Various constraints are in place to minimize the risk of choosing wrong connections and options. Some of these constraints are checked using OCL (e.g., checking for non-null references or proxies), but others are checked at the time of generating a configuration file corresponding to these features.

3.3 Behavioral Modeling using SRNML

A second language we have developed is the Stochastic Reward Net Modeling Language (SRNML). Figure 8 illustrates the meta model of SRNML. SRNML provides all the artifacts necessary to model a building block as a SRN. This includes elements, such as immediate transitions, inhibitor arcs, tokens, places and guard functions. SRNML is also developed using GME. Details of SRN models and its syntactic/semantic elements have been covered in Section 2.2. The generative capabilities associated with SRNML synthesize artifacts necessary to solve the SRNs and analyze the QoS of the building blocks.

Figure 7. POSAML Model: Feature View

In Figure 7 we observe that the modeler has specified the concurrency strategy (such as reactive or thread-per-connection), the

Figure 8. SRN Metamodel

Figure 9 illustrates the SRN model developed using SRNML and represents exactly the same model that was developed manually and described in Section 2.2.

3.4 Model Scalability via Model Replicators

Although scalability of the models is not the focus of this paper, we describe our approach for completeness sake. Model scalability is addressed using a model replicator approach [7] supported by aspect modeling, replication and weaving tools, such as C-SAW (Constraint-Specification Aspect Weaver) [6], which are integrated with GME and can work with any metamodels developed in GME. C-SAW provides support for modularizing crosscutting modeling concerns as well as scaling models in the GME. The weaver operates on the internal representation of a model (similar to an abstract
Section 3 introduced the modeling languages we have developed to support modeling of the composed middleware systems. This section describes the synthesis capabilities of the DSMLs that allow generation of metadata artifacts required to run the analytical model solvers, simulators and benchmarking tools. In this section we use the manually crafted SRN model of the reactor pattern described in Section 2.2 to describe the generative capabilities.

4. Generative Capabilities of POSAML and SRNML

Section 3 introduced the modeling languages we have developed to create QoS analysis models of the composed middleware systems. This section describes the synthesis capabilities of the DSMLs that allow generation of metadata artifacts required to run the analytical model solvers, simulators and benchmarking tools. In this section we use the manually crafted SRN model of the reactor pattern described in Section 2.2 to describe the generative capabilities.

4.1 Generative Capabilities of POSAML

This section summarizes the generative capabilities of POSAML. Presently, there are two model interpreters available that are used to generate configuration and script files for various POSAML configurations. These are:

1. Configurator Interpreter: This interpreter generates two artifacts which are required to configure middleware. POSAML itself is platform-independent. However, each interpreter is specific to a middleware framework (e.g., the ACE/TAO middleware [25] framework). We envision that the configuration files for each middleware technology could be generated using an interpreter specific to that technology.

One of the artifacts generated by this interpreter is the service configurator file (svc.conf), which 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 an application to configure service objects statically while dynamically configuring middleware options. Options in the svc.conf file can represent either the components provided by the middleware framework or customized components developed by users. If this configurator file is not available, default configurations are selected by the middleware framework.

The following represent different factories that can be configured:

(a) 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, regardless of whether they perform client or server roles.

(b) Server Strategy factory: This factory creates various strategies of special utility to the ORB that is useful for controlling the behavior of servers.

(c) Client Strategy factory: This factory creates various strategies of special utility to the ORB, useful for controlling the behavior of clients.

Excerpts of the svc.conf comprising different strategies are shown below. Different middleware configurations can be seamlessly synthesized in this manner. The generated configuration directives are guaranteed to be compatible for the compositions of the building blocks since any incompatibilities in the compositions will be caught by the DSML’s constraint checking capabilities. Moreover, the generative framework can synthesize such configuration files for different middleware frameworks by plugging in a separate model interpreter.

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

A script file is also generated by the configurator interpreter to run any application, such as a Naming service or benchmarking evaluation tool with proper end points like listening ports, protocol and host name. These endpoints are also used by the Acceptor-Connector pattern for different transport handles. Excerpts from an example of the generated script file is given below. The script file can be arbitrarily complex depending on the models of the configuration aspect. Moreover, it may be tailored for the middleware platform of choice by plugging in the right model interpreter.

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

2. Benchmarking Interpreter: 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 Features Aspect. This interpreter stores information in an XML file that is used by an existing benchmarking library, which is parametrized. Excerpts of the generated XML file are shown below. This XML file can be arbitrarily complex depending on the building block compositions and their configurations. The generative capabilities...
This section illustrates the generative capabilities of SRNML and how we used it for QoS analysis of the scenario illustrated in Section 2.2. The following listing illustrates a snippet of the model generated artifacts that drive the SPNP analytical model solver [9]. In this case it is a C-like program used by the SPNP solver. Similar artifacts can be generated by our modeling tools that drive simulations and empirical benchmarks.

In the code snippet shown below the keyword place is used to represent a place, trans is used to denote a timed transition and rateval used to denote the rate of a time transition. inn is used to denote an immediate transition and probval and priority denotes its probability and priority respectively, and guard denotes the conditions under which the transition fires. For example, the guard function for transition Sn1 is given by function sn1 which evaluates to true when the condition presented in Table 1 is satisfied and false otherwise. The keywords iarc, oarc and mharc represent input, output and inhibitor arcs respectively. The entire set of input, output and inhibitor arcs are not shown here due to space limitations.

```c
void net () {
  /* Places */
  place("B1"); place("B2");
  place("S1"); place("S2");
  place("StSnp"); init("StSnp",1);
  place("SnpInProg");

  /* Timed transitions */
  trans("A1"); rateval("A1",lam1);
  trans("A2"); rateval("A2",lam2);
  trans("Sr1"); rateval("Sr1",mul1);
  trans("Sr2"); rateval("Sr2",mul2);

  /* Immediate transitions */
  inn("Sn1"); probval("Sn1",1.0);
  priority("Sn1",0.0); guard("Sn1",sn1);
  inn("Sn2"); probval("Sn2",1.0);
  priority("Sn2",0.0); guard("Sn2",sn2);
  inn("StSnp"); probval("StSnp",1.0);
  priority("StSnp",0.0); guard("StSnp",gd);
  inn("TESnp"); probval("TESnp",1.000);
  guard("TESnp",gd1); priority("TESnp",120);

  /* Input arcs */
  iarc("Sn1","B1");

  /* Output arcs */
  oarc("A1","B1");
  ...

  /* Inhibitor arcs */
  mharc("A1","B1",1);

  ...
}
```

The total number of lines in the generated SRN code depends on the number of event types handled by the reactor. For example, for two event types the number of lines of code is 231, for three event types the number of lines of code is 278 and for four types of events the number of lines of code is 390. It can be observed that the number of lines of code increases with the number of event types.

A substantial advantage provided by the generative capabilities is the near total elimination of accidental complexities. For example, a single change to the SRNML model (such as, scaling the reactor to include one more type of event), will need changes to several places in the code used to solve the model. One such change is the need to add the places, transitions and arcs. This may be somewhat easier to address even manually. The difficulty arises when one needs to manage the interactions with the other parts of the model, which if not done right can lead to numerous errors including potential for infinite loops, which end up becoming a non-trivial debugging effort. All these complexities are largely eliminated by virtue of the generative capabilities of SRNML.

In the design stage, it is important to assess the impact of configuration options and parameters on the performance metrics. One of the configuration options is the buffer space available to hold the incoming events of each type. This choice will have a direct impact on all the performance metrics. Most importantly, from the user’s perspective, the buffer space will influence the probability of denying the service requests.

The impact of the buffer capacities on the performance measures can be analyzed. The values of the remaining parameters (except for the buffer capacities) are reported in Table 3. We consider two values of buffer capacities $N_1$ and $N_2$. In the first case, the buffer capacity is set to 1 for both types of events, whereas in the second case the buffer capacity of both types of events is set to 5. The performance metrics for both of these cases are summarized in Table 4. Because the values of the parameters of the service requests from organization #1 ($\lambda_1, \mu_1$ and $N_1$) are the same as the values of the parameters for the service requests from organization #2 ($\lambda_2, \mu_2$, and $N_2$), the throughputs, queue lengths, and the loss probabilities are the same for both types of service requests for each one of the buffer capacities as indicated in Table 4. It can be observed that the loss probabilities are significantly higher when the buffer capacity is 1 compared to the case when the buffer space is 5. Also, due to the higher loss probability, the throughput is slightly lower when the buffer capacity is 1 than when the buffer capacity is 5. The performance metrics obtained from the SRN model were validated by the metrics obtained from simulation using CSIM [27].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Event #1</th>
<th>Event #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival rate</td>
<td>$\lambda_1 = 0.400/\text{sec.}$</td>
<td>$\lambda_2 = 0.400/\text{sec.}$</td>
</tr>
<tr>
<td>Service rate</td>
<td>$\mu_1 = 2.000/\text{sec.}$</td>
<td>$\mu_2 = 2.000/\text{sec.}$</td>
</tr>
</tbody>
</table>

It is rarely the case that the values of the arrival and service rates can be estimated with accuracy in the design phase. The SRN model can also be used to analyze the sensitivity of the performance measures to the parameter values. The results of sensitivity analysis are not included in this paper due to space limitations.
5. Related Work

Performance and dependability analysis of some middleware services and patterns has been addressed by some researchers. Ramani et al. [22] develop a SRN model for the performance analysis of CORBA event service, which is a pattern that provides publish/subscribe service. Aldred et al. [1] develop Colored Petri Net (CPN) models for different types of coupling between the application components and with the underlying middleware. They also define the composition rules for composing the CPN models if multiple types of coupling is used simultaneously in an application. A dominant aspect of these works are related to application-specific performance modeling. In contrast we are concerned with determining how the underlying middleware that is composed for the systems they host will perform.

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. [3] classify the quality attributes according to the possibility of predicting the attributes of the compositions based on the attributes of the components and the influence of other factors 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 [28] investigated the use of aspect-oriented model transformation techniques to address performance concerns that are woven into a primary UML model of functional behavior. It has been observed that an improved separation of the performance description from the core behavior enables various design alternatives to be considered more readily (i.e., after separation, a specific performance concern can be represented as a variability measure that can be modified to examine the overall systemic effect). The performance concerns are specified in the UML profile for Schedulability, Performance, and Time (SPT) with underlying analysis performed by a Layered Queuing Network (LQN) solver.

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.

6. Concluding Remarks

Time-to-market pressures and economic reasons mandate that the next generation of distributed networked services be developed via composition and assembly of off-the-shelf reusable components. Even the middleware frameworks that host these applications have to be provisioned by selecting, composing and configuring the right set of building blocks. Service providers are now required to reason about the performance and dependability of such composed systems earlier in the systems development lifecycle so that flaws can be rectified earlier. However, due to the substantial variability that exists within every building block and in their compositions, a selection approach that uses ad-hoc manual methods to analyze and select optimal configurations becomes infeasible. This paper described a framework for design-time QoS analysis and validation of services that are composed, configured and deployed using patterns-based middleware building blocks.

With the help of two modeling languages, namely POSAML and SRNML, structural as well as behavioral modeling of middleware systems can be carried out. POSAML can be used to perform design-time middleware block selection, configuration and composition. SRNML can be used to automatically synthesize complex SRN models, simulations and empirical benchmarks for the composed systems.

Lessons Learned and Future Work

Model-driven Engineering comprising the use of DSMs and generative programming provides an effective solution to address the challenges facing next generation, large-scale software systems. QoS validation of these systems both at design- and run-time are essential to ensure that when these systems are deployed, they deliver the promised QoS guarantees to the users. Our experience developing and using the MDE framework described in this paper suggests the following benefits:

- MDE approaches alleviate several tedious and error-prone tasks, and promote reusability of the artifacts that are developed.
- DSMs within MDE frameworks provide an intuitive mechanism for users to represent their systems.
- Generative capabilities of MDE frameworks alleviate the need for system developers to be aware of different backend artifacts, such as performance analysis engines and the real platforms on which the systems are hosted. Such tasks as providing the right inputs for the analysis engines or configurations of the platforms can be synthesized by the generative capabilities.
- Variability in the systems - in our case the middleware building blocks and their compositions - can be handled by introducing feature modeling and aspect weaving.

Some shortcomings and potential solutions to our MDE approach are described below:

- The POSAML models of building blocks capture the patterns described in the POSA pattern language. Similarly, the SRNML models capture the dynamics of a pattern. Patterns describe the essence of a solution but do not capture the platform-specific and implementation-specific details. Thus, our POSAML and SRNML DSMs do not capture these traits and hence the QoS validation and analysis is approximate. It is possible to introduce additional feature modeling capabilities that handle platform-specific details, which can then provide more accurate analysis.

- Our MDE framework addresses provisioning of individual middleware stacks for applications. For distributed applications, there may be a need to coordinate the middleware provisioning on individual nodes of a distributed system for achieving distributed QoS. This is a much harder problem, which is currently not addressed in our MDE approach and will serve as the basis for our future work.

- Applications themselves are increasingly being developed using component-based engineering. MDE approaches to enable...
application composition and deployment are emerging. Our MDE framework complements such frameworks. There is a need, however, to integrate such frameworks to realize very powerful tool suites.

• Although our current focus is on applying analytical methods, our vision calls for an integrated framework to drive QoS analysis using analytical models, simulations and empirical benchmarking of large systems. Our goal is to address different dimensions of QoS including performance, reliability and security.

• Ultimately our goal is to provide additional generative capabilities such as auto-generation of parameterized middleware code from POSAAML models for contemporary middleware platforms.

• Scaling the models to larger systems can be achieved using techniques, such as model replicators [7]. We have conducted preliminary work in model scaling for individual building blocks. Our future work will consider applying these scalability techniques in the context of middleware building block compositions.

Acknowledgments

This research was supported by a grant from the National Science Foundation.

References


