STUDY ON MODELS OF PERFORMANCE EVALUATION OF SOFTWARE ARCHITECTURES

G.Kasi Reddy #1 and Dr.D.Sravan Kumar *2

#1 Research Scholar CSE JNTU Hyderabad A.P. India
gkasireddy@rediffmail.com

*2 Principal and Professor CSE KITE WCPES Hyderabad A.P. India
dasojusravan@gmail.com

ABSTRACT

The fundamental structure of a software system is referred to as the software architecture. Software architectural evaluation becomes a familiar practice in software engineering community for developing quality software. There have been several methods and techniques to evaluate software architectures with respect to the desired quality attributes such as maintainability, usability and performance. Researchers have identified that the quality attributes of a software system are restricted by the architecture. Therefore, it is important to evaluate the quality properties of a system specially performance during architectural design. This paper presents a study on two methods i.e Performance Evaluation of Software Architectures using Prototyping and Performance evaluation of software architectures with queuing network models which are used for evaluating the performance of a software architecture.

Keywords: Unified Modeling Language, Queuing Networks, Software Performance Evaluation, Distributed System, Software architecture and architecture prototyping.

1. INTRODUCTION

Software Architectures (SA) describe software system structures at a high level of abstraction [3, 4]. They represent at an early stage of development the phase in which basic choices of components and interactions among components are made. Ideally, those choices reflect any constraints imposed by the hardware architecture upon with the system is to run. Designers are often faced with the problem of choosing among different functionally equivalent software architectures. This choice is driven by non-functional factors such as performance, reliability, and topological/economical constraints. Among these, performance is one of the most influential factors to be addressed.

The size and complexity of software systems are constantly increasing. During recent years, software engineering research has identified that the quality properties of software systems, e.g., performance and maintenance, often are constrained by their architecture [1]. Before committing to a particular software
architecture, it is important to make sure that it handles all the requirements that are put upon it, and that it does this reasonably well.

One important quality attribute to evaluate during architectural design is performance. Many times performance problems are not detected until system integration test, and thus are very costly to correct [3]. Some even argue that a design change is at least ten times more expensive after the code has been written than during architectural design. Therefore, it is important to evaluate the performance of a system as early as possible in the system development process, i.e., during the architectural design phase.

In this paper two different approaches i.e Performance Evaluation of Software Architectures using Prototyping and Performance evaluation of software architectures with queuing network models were studied.

2. SOFTWARE ARCHITECTURE

Software systems are constructed with a requirement specification as a base. The requirements in the requirement specification can be categorized into functional requirements and non-functional requirements, also called quality requirements. The design of software systems has traditionally been centred around the functional requirements. Although software engineering practice was forced to incorporate the quality requirements as well, software engineering research focused on the system functionality.

During recent years, the domain of software architecture [1, 4, 5] has emerged as an important area of research in software engineering. This is in response to the recognition that the architecture of a software system often constrains the quality attributes. Thus, architectures have theoretical and practical limits for quality attributes that may cause the quality requirements not to be fulfilled. If no analysis is done during architectural design, the design may be implemented with the intention to measure the quality attributes and optimize the system at a later state. However, the architecture of a software system is fundamental to its structure and cannot be changed without affecting virtually all components and, consequently, considerable effort.

Software architecture can be divided into three problem areas, i.e., designing, describing, and evaluating a software architecture. In this paper we focus on evaluating software architectures, and in particular evaluating their performance. Four approaches to architecture evaluation can be identified, i.e., scenarios, simulation, mathematical modelling, and experience-based reasoning. Smith [3] discusses an approach to modelling system performance mathematically, although one may require simulation in certain cases. In this paper we give a study on on the construction of an executable prototype of the architecture and an approach for software performance modeling based on UML as the software description notation, and Queuing Networks (Kleinrock, 1975) as the performance model.

2.1 QN Model Overview

QN Model motivates two things. First, the high level of abstraction of the model makes it easy to define a direct correspondence between software components and performance model elements, so that one can easily derive the QN from UML diagram specification. This allows to provide a direct feedback of performance results at the software design level. The second motivation is that it aims to identify a simple product-form QN (Kleinrock, 1975) in order to obtain performance measures by computationally efficient algorithms (Reiser and Lavenberg, 1980).
Performance evaluation of software architectures with queuing network models considers the software performance evaluation cycle described by the following steps:

1. Definition of the performance requirements of the software system.
2. UML specification of the software system.
3. Transformation of the software specification into a performance model based on QN, using a suitable transformation algorithm.
4. Analysis of the QN and derivation of performance indices.
5. Feedback of performance results on the software model elements.
6. Analysis of the software performance results and possible iteration from step 2 (e.g., if the performance requirements are not met or different architectural implementations have to be examined).

Starting from the UML software model annotated with performance-oriented annotations based on the UML Performance Profile (Object Management Group (OMG), 2002a), this approach considers UML Use Case, Activity and Deployment diagrams. It defines an algorithm for translating the annotated software model into a QN based performance model. The performance model is solved using an appropriate solution technique and performance results provide feedback at the software specification level by the annotations. Hence the software performance analysis cycle can be iterated to meet given performance requirements or to compare different software design alternatives.

Figure 1: Software Performance Steps and UML-QNE Structure

3. THE PROTOTYPE-BASED EVALUATION APPROACH

In the core of the prototype-based evaluation approach is the architecture prototype that approximates the behavior of the completed software system. This section gives a short introduction to the steps involved in performing a simulation based architecture evaluation, then describes the changes that are made to that approach, and finally describe the resulting prototype-based evaluation approach.

3.1 Simulation based architecture evaluation

A simulation-based evaluation is performed in five steps [2]:

1. Define and implement context.
2. Implement architectural components.
3. Implement profile.
4. Simulate system and initiate profile.
5. Predict quality attribute.
Define and implement context. In this first step, two things are done. First, the environment that the simulated architecture is going to interact with is defined. Second, the abstraction level that the simulation environment is to be implemented at is defined (high abstraction gives less detailed data, low abstraction gives accurate data but increases model complexity).

Implement architectural components. In this step, the components that make up the architecture are implemented. The component definitions and how the components interact with each other can be taken directly from the architecture documentation.

Implement profile. A profile is a collection of scenarios that are designed to test a specific quality attribute. The scenarios are similar to use-cases in that they describe a typical sequence of events. These sequences are implemented using the architectural components that are going to be evaluated. This results in a model of the system components and their behavior. How a profile is implemented depends on which quality attribute we are trying to assess as well as the abstraction level that is necessary for getting relevant data.

Simulate system and initiate profile. The simulation model is executed. During the execution, data is gathered and stored for analysis. The type of data that is gathered depends on which quality attribute that we want to evaluate.

Predict quality attribute. The final step is to analyze the collected data and try to predict how well the architecture fulfills the quality requirements that we are trying to evaluate. This step is preferably automated since a simulation run usually results in a large amount of raw data.

3.2 Adaptations to the evaluation method

The workflow from the simulation-based evaluation had to be adapted to incorporate steps that this model wanted to perform in this prototype-based evaluation. The main changes that this approach made were to introduce an evaluation support framework and put more emphasis on iteration in the evaluation process.

3.2.1 Evaluation support framework

This approach added the step of creating an evaluation support framework for use during the evaluation. A layered view of where the support framework is placed is shown in Figure 2. It choose to create the evaluation support framework for two reasons.

First, it makes us less dependent on the architecture component that this approach want to evaluate. The framework decouples the architecture component that it is evaluating from the architecture model that is used to generate input to the component. This increases the reusability of the architecture model as it only depends on the API provided by the framework and not directly on the architecture component.

Second, all logging can be performed by the framework, resulting in that neither the architecture model nor the architecture component that are evaluated need to care about the logging. This leads to both that the logging is done in a consistent way independent of the underlying architecture component, and that no change has to be made to the architecture component when it is fitted to the framework. All that is needed is a wrapper class that translates between the calls from the framework and the architecture component.
During the development and execution of the prototype it found that it became necessary to perform the development of both the architecture model and the evaluation support framework in an iterative way. This needed to reiterate steps two to five in order to make adjustments to the way data was logged, and also to the behavior of the architecture model that was used. The need to make these changes was identified first after an initial execution of the simulation and analysis of the generated data. The positive thing with adding an iteration is that the initial results can be reviewed by experts (if such are available) that can determine if the results are sensible or not, and if changes to the model are necessary.

3.3. Prototype based architecture evaluation

In order to perform a prototype based evaluation there are some conditions that has to be fulfilled.
- First, there has to be at least one architecture defined if the goal of the evaluation is to compare alternative architectures to each other then we will of course need more.
- Second, if we want to evaluate the performance of one or more candidates for a part of the software architecture then these components has to be available. This is usually no problem with COTS components but might pose a problem if the components are to be developed in house.

In addition, it is a preferable, but not necessary condition, that the target platform (or equivalent) of the architecture is available. If it is possible to run the prototype on the correct hardware, it will give more accurate results.

After integrating this adaptations in the evaluation method it ended up with the following method for prototype based architecture evaluation.

1. Define evaluation goal.
2. Implement an evaluation support framework.
3. Integrate architectural components.
4. Implement architecture model.
5. Execute prototype.
6. Analyse logs.
7. Predict quality attribute.
8. If necessary, reiterate.

**Define evaluation goal.** Define what it is that should be evaluated, are we looking at more one or more architecture candidates or architecture components, and which quality attributes are we interested in evaluating.
Implement an evaluation support framework. The evaluation support framework’s main task is to gather data that is relevant for fulfilling the evaluation goal that has been defined. Depending on the goal of the evaluation, the framework has to be designed accordingly, but the main task of the support framework is always to gather data. The support framework can also be used to provide common functions such as utility classes for the architecture model.

Integrate architectural components. The component of the architecture that we want to evaluate has to be adapted so that the evaluation support framework can interact with it.

Implement architecture model. Implement a model of the architecture with the help of the evaluation support framework. The model together with the evaluation support framework and the component that is evaluated becomes an executable prototype.

Execute prototype. Execute the prototype and gather the data for analysis in the next step. Make sure that the execution environment matches the target environment as close as possible.

Analyse logs. Analyse the gathered logs and extract information regarding the quality attributes that are under evaluation. The analysis is with advantage automated as much as possible since the amount of data easily becomes overwhelming.

Predict quality attribute. Predict the quality attributes that are to be evaluated based on the information from the analysed logs.

If necessary, reiterate. This goes for all the steps in the evaluation approach. As the different steps are completed it is easy to see things that were overlooked during the previous step or steps. Once all the steps has been completed and results from the analysis are available, you could let an expert review them and use the feedback for deciding if adjustments have to be done to the prototype. These adjustments can be necessary in both the architecture model and the evaluation support framework. Another advantage is that it is possible to make a test run to validate that the analysis tools are working correctly and that the data that is gathered really is useful for addressing the goals of the evaluation.

4. QN APPROACH

Performance evaluation of software architectures with queuing network model proposes an approach for software performance modeling based on UML as the software description notation, and Queuing Networks (Kleinrock, 1975) as the performance model.

4.1 The software model

The software system is described in term of the following UML diagrams: Use Case diagrams, representing workloads applied to the system; Deployment diagrams, describing the available physical resources (processors) where computations take place; Activity diagrams, describing both the order in which resources are used, and the corresponding service demand.
Use case diagrams  Each actor in a Use Case diagram may represent a stream of requests arriving at the system. There may be an unlimited sequence of requests (open workload), or a fixed population of users requiring service from the system (closed workload). Actors representing open workloads are stereotyped as "<<OpenUser>>", while actors representing closed workloads are stereotyped as "<<ClosedUser>>". "<<OpenUser>>" actors.

For the latter class of actors it is necessary to specify the total number of requests circulating in the system; this is done with the PApopulation tagged value associated to the actor. Multiple actors may be present in the same system, meaning that there are multiple concurrent streams of requests. Each actor has an associated use case representing the computations triggered by each actor. The details of the computations are described by an Activity diagram which must be associated to each Use Case.

Fig. 4 shows an example of annotated Use Case diagram, where Actor 1 represents an open workload and Actor 2 represents a closed population of 10 requests circulating in the system.

Deployment diagrams  Deployment diagrams are used to model the physical resources available in the system. Each resource is represented by a node in the Deployment diagrams. Each node, which must be stereotyped as "<<node>>", represents a processor with a given scheduling policy. The following tagged values can be associated to Deployment diagram nodes:

PAschedpolicy  Defines the scheduling policy of that processor, which can be one of “FIFO” (First-In-First-Out), “LIFO” (Last-In-First-Out) and “PS” (Processor Sharing).

PArate  defines the processing rate (speed) of the processor. This means that a user with a service request of S time units will complete in S/PArate time units.

PAservers  defines the number of processors concurrently executing the requests. This tag can be used to represent a multiprocessor resource having N equal processors executing in parallel. All the processors share the same queue of pending requests.

Fig. 4 shows an example of annotated Deployment diagram. Each node represents a processor with the characteristics described with the tagged values.

Activity diagrams  Having described the workloads and the physical resources available in the system, it is now necessary to specify how the resources are used, that is, which computations are performed in the system. Computations are described by associating an Activity diagram to Use Cases. In this way, the workload represented by an actor will trigger the computation represented by the Activity diagram associated to the corresponding Use Case.
Each action state of an Activity diagram, stereotyped as << ServiceCenter >>, represents a computation which requires service to one resource. The following tagged values can be specified to provide informations for building the performance model:

- **PAresource** is the name of the resource (node in one of the Deployment diagrams) from which service is requested.

- **PAoccurrence** represents the interarrival time of this service request. The tag can be specified only if the QN performance model is an open QN (described in the following).

Action states are linked each other with a predecessors-successor relationship. This relationship is used to model the sequence of computations which are executed by a request. In general, one action state may have multiple successors. In this case, each transition must be labeled with the probability that the transition is traversed. This models nondeterminism in the sequence of actions which are executed.

Fig. 5 represents an example of annotated Activity diagram. Transitions are annotated with the PAprob tag showing the probability of traversing the corresponding arc. Values of the PAresource tags refers to node names of the Deployment diagram in Fig. 4.

Note that this approach does not support synchronization bars in Activity diagrams. Synchronization bars are used to split the execution flow in multiple, concurrent threads or to join multiple threads into a single flow. While synchronization bars could be translated into fork and join nodes in the QN model, the resulting QN would not be in product form, thus requiring more complex solution algorithms, which may produce only approximate results instead of exact ones.

### 4.2 The Performance Model

This approach derives the performance model from the UML specification, as sketched in Section 2, by defining a multiclass QN (Lavenberg, 1983). A Queuing Network is a set of service centers (also called nodes or service stations). Each service center is formed by a queue and a set of identical servers.

- **Requests** join the queue and are serviced by the first server becoming idle, according to a specific queuing discipline.

UML model components are translated into the corresponding QN model elements as follows. Each node in the Deployment diagrams defines a service center. From Actors in Use Case diagrams one can identify...
the type of QN model, e.g., open, closed or mixed QN. Finally, from Activity diagrams one can derive the network topology, that is the behavior of the classes of users circulating through the system. The mapping between UML and performance model elements is illustrated in Fig. 6.

Open QN can be generated from Use Case diagrams in which the actor is stereotyped as <<OpenUser>>. Note that no tagged values are defined for this kind of actor, as externally arriving requests can be defined by the PAoccurrence associated to each action state. This means that the action state can be executed also by an external stream of requests reaching the system at the given rate. Closed QN are made of a number of service centers in which a fixed population of requests circulates. Each request receives service from a service center, after which it is routed to another service center. This kind of network is generated from an actor stereotyped as <<ClosedUser>>, where the associated PApopulation tag is used to specify the number of requests. Mixed QN are an extension of the previously described networks: in mixed networks there are R different classes of circulating requests. Each class of requests moves through the network differently from requests of other classes. Some classes of requests move through the network with a closed topology, i.e., external arrivals and departures are not allowed, other classes of requests move according to an open topology. The tagged value PAserviceTime specifies the average amount of time that requests spend at a service center. Mixed QN are generated from Use Case diagrams with multiple actors, and where each actor represents one user class, depending on its stereotype.

4.3 The transformation algorithm

The proposed algorithm, named Algorithm UML-QNE, translates an annotated UML specification into the QN model. This uses the model notation shown in Table 1. In the algorithm it denotes with TagName(X) the value of tag TagName associated to UML element X.

| $N$ | Number of nodes in the UML Deployment diagram |
| $S_i$ | i-th service center |
| $\mu_i$ | Service rate of service center $S_i$ |
| $\lambda^C_i$ | Arrival rate of class C customers at service center $S_i$ |
| $NS^C_i$ | Number of servers in service center $S_i$ |
| $P_{ij}^C$ | $N \times N$ routing matrix for class C customers |

Table 1: Notation used in Algorithm 1

The algorithm is illustrated in Fig. 1 and works according to the following steps:
1. For each Deployment diagram node it defines a corresponding service center; if $N$ is the number of nodes in the Deployment Diagrams, it defines $N$ service centers $S_1, S_2, \ldots S_N$.
2. For each Actor in Use Case diagrams it defines a class of customers in the QN.
3. For each Activity diagram associated to an Actor we define the routing matrix for the current class of customers according to the predecessor-successor relationship of Action states. Routing probabilities are obtained from the PAprob tag associated to UML transitions.

The computational complexity of Algorithm is $O(N + T + A)$, where $N$ is the number of nodes in the Deployment Diagrams, $T$ and $A$ are the number of transitions and action states in all the Activity diagrams, respectively.
5. CONCLUSION

In this paper we have described the prototype based architecture evaluation approach and the steps that it consists of. The approach is based on the simulation based evaluation approach but adds mainly the construction of an evaluation support framework and a clearer focus on iteration during the evaluation. The use of the evaluation support framework simplifies the implementation of alternative architecture models, makes consistent gathering of data simpler, and makes it possible to evaluate alternative implementations of an architecture component. In this paper we have also discussed an approach for performance modeling of UML software architectures. The approach considers annotations based on the UML Performance Profile, and derives a performance model based on Queuing Network. Performance
evaluation of the QN by efficient algorithms provides a set of steady-state performance indices that characterize the software system behavior.

REFERENCES