UML based performance modeling of object-oriented distributed systems

Pekka Kähkipuro

Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, FINLAND
Pekka.Kahkipuro@cs.Helsinki.FI

This position paper briefly presents a performance modeling framework that is being developed at the University of Helsinki. We first give an overview of the framework, enumerate its main elements, and describe relationships between them. The technical structure of the framework is further analyzed by examining the four representations for performance models that are used within the framework. Finally, the use of the framework is illustrated with a simple example, and a number of possible extensions are discussed.

1. Elements of the framework

The architecture of the framework consists of four main elements:

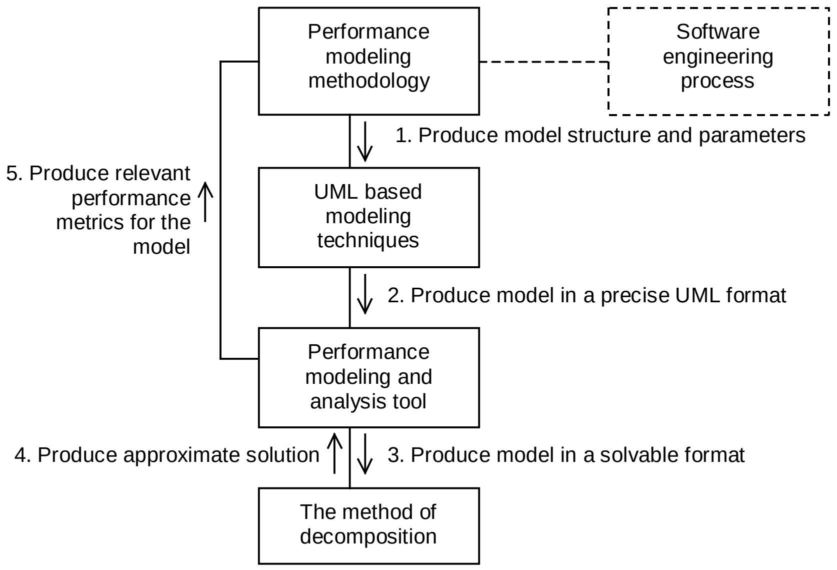
- The method of decomposition,
- UML based performance modeling techniques,
- Performance modeling methodology,
- Object-oriented performance modeling and analysis tool.

The elements and their relationships are illustrated in Figure 1. *The method of decomposition* (MOD) provides the foundation for the framework. It defines an algorithm for finding an approximate solution for performance models. The key features of the MOD algorithm are:

- Support for open, closed, and mixed workloads,
- Support for simultaneous resource possessions that arise, for example, from synchronous operation invocations in CORBA based systems,
- Support for circular calling dependencies, such as those emerging from callback interfaces,
- Support for recursive accesses that take place when objects call themselves, and
- Presentation of results on an access by access basis in order to allow results to be presented with UML sequence or collaboration diagrams.

The algorithm has been presented in [1]. The MOD alone, however, cannot be used for modeling large CORBA based distributed systems due to its low-level representation of the modeled system. Even simple application level configuration results into a complex performance model since all technical resources (e.g. network adapters and network latencies) are mixed with application resources.

UML based performance modeling techniques provide the means for modeling complex information systems. The key issue in these techniques is the use of abstractions for separating application level issues from the use of technical resources. These abstractions can be



Performance modeling methodology
Software engineering process
1. Produce model structure and parameters
5. Produce relevant performance metrics for the model
UML based modeling techniques
2. Produce model in a precise UML format
Performance modeling and analysis tool
4. Produce approximate solution
3. Produce model in a solvable format
The method of decomposition

Figure 1. Elements of the performance modeling framework.

structured into multiple layers, some of which can be omitted at the early stages of systems development. This way, performance modeling can be applied to all phases of the development process. Moreover, the proposed UML based techniques are close to the normal UML modeling style as proposed in the UML standard and literature, thereby allowing the use of existing functional UML models for performance modeling. These techniques are further discussed in [2].

The *performance modeling methodology* provides a link to the software engineering process. The aim is to indicate how the proposed UML modeling techniques can be used at different stages of systems development to produce useful performance models for the system. The methodology is based upon a layered model for CORBA based distributed systems, thereby emphasizing the separation of application functionality, the infrastructure, the network, and the actual configuration. The layers are specified individually but cooperatively to allow a comprehensive model to be built in a stepwise process that proceeds in parallel with the software development process. Initially, some of the layers can be empty and

The purpose of the *object-oriented performance modeling and analysis tool* (OAT) is to automate some of the tasks required by the framework. Currently, the prototype implementation of the tool implements the following tasks:

- Transformation of UML based performance models into a solvable format required by the MOD algorithm,
- Implementation of the MOD algorithm to produce an approximate solution for the performance model, and
- Conversion of the solution into a set of relevant performance metrics to be used in the performance modeling methodology.

In Figure 1, these tasks correspond to steps 3, 4, and 5.

2. Four performance model representations

The technical aspects of the framework and the operation of the modeling and analysis tool can be described in terms of four performance model representations that are used in the framework. Each representation has its own notation, and the framework architecture defines mappings between them, as shown in Figure 2. The idea is to start from the UML representa-

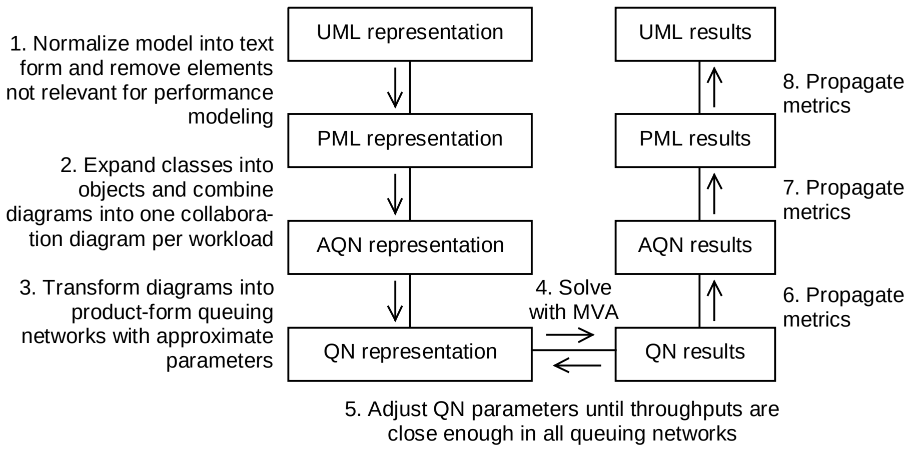

1. Normalize model into text form and remove elements not relevant for performance modeling
2. Expand classes into objects and combine diagrams into one collaboration diagram per workload
3. Transform diagrams into product-form queuing networks with approximate parameters
UML representation
PML representation
AQN representation
QN representation
4. Solve with MVA
5. Adjust QN parameters until throughputs are close enough in all queuing networks
UML results
PML results
AQN results
QN results
6. Propagate metrics
7. Propagate metrics
8. Propagate metrics

Figure 2. Four representations for performance models.

tion and proceed downwards using the mappings. Once the bottom has been reached, an approximate solution can be found for the model. The mappings also indicate how the obtained metrics can be propagated upwards.

The *UML representation* describes the system with UML diagrams. This representation may contain purely functional elements that are not needed for performance modeling. To reduce the complexity of the diagrams, we assume that the UML representation is divided into separate layers corresponding to different parts of the system, such as the application, the infrastructure, and the network.

The *PML representation* provides an accurate textual notation for representing performance related items in the UML diagrams. The PML representation has the same layered structure as the UML representation, and the mapping from UML to PML is straightforward. The purpose of this representation is to filter out those features that have no significance for performance modeling, such as graphical UML variations and purely functional parts of the UML model. Moreover, the PML representation has an important role in the development of the framework, as it is currently the input format for the prototype implementation of the modeling and analysis tool. However, the use of PML is not mandatory, and any other human-understandable UML representation could be used instead. For example, we might later opt for the human-usable textual UML notation that is currently being developed by the OMG [3]. For the purposes of exploring the performance modeling framework, however, the current PML based approach is sufficient. Later improvements can be implemented as the standards become integrated into commercially available tools. An abstract syntax for the PML notation is given in Appendix A.

The *AQN representation* describes the system in the form of augmented queuing networks that may contain simultaneous resource possessions. This representation allows the use of the MOD algorithm for solving the model. The AQN representation is obtained from the PML representation by expanding object classes into object instances that correspond to individual resources in the system. Moreover, the UML collaboration and sequence diagrams describing the behavior of the application and the infrastructure are combined into one or more workload specifications. Each workload specification can be visualized with a collaboration diagram that indicates how the system's resources are used by that particular workload.

The *QN representation* consists of separable queuing networks with mutual dependencies that correlate them to the same overall system. The queuing networks are obtained from the AQN representation during the initial steps of MOD algorithm. Initially, the queuing networks contain a number of unknown parameters. To solve the networks, iteration is used for finding a solution with the specified level of accuracy. The transformation from the AQN representation into the QN representation involves a number of approximations that are needed to make the queuing networks solvable with efficient algorithms.

Similar tiered architectures are common in performance modeling [4, 5], but the top representation is often a sophisticated performance modeling notation, such as a variant of Petri nets, to support advanced modeling techniques. In our case, we use a non-technical top representation for hiding most of the underlying performance modeling issues. This approach is more aligned with the goals of the framework that emphasize the ease of use and the use of high-level abstractions.

3. Example

We illustrate the framework with a simple example of a network monitoring system. The system operates as follows. A number of receiver objects accept messages from network elements and forward them to handler objects that are responsible for executing appropriate actions. There is also a database for maintaining descriptions for the actions. These entities are represented by the *CReceiver*, *CHandler*, and *CDatabase* classes. To illustrate the use of interfaces, the operations of the *CHandler* class are actually defined in the *IHandler* interface. Figure 3 illustrates the static structure of the system. Service demand estimates for the operations are in milliseconds.

The model has two workloads: a background load for the database and the main load. The workloads are illustrated in Figure 4. The background load has an estimated rate of 1 database query per second. The main load represents the handling of messages arriving from network

```
class CReceiver {
  property queue;
  ReceiveJob() {cpu=10};
};
class CHandler {
  property queue;
  realizes IHandler;
};
class CDatabase {
  property queue;
  GetActions() {cpu=10,disk=50};
};
interface IHandler {
  AcceptJob() {cpu=5};
  DoActions() {cpu=20};
};
```

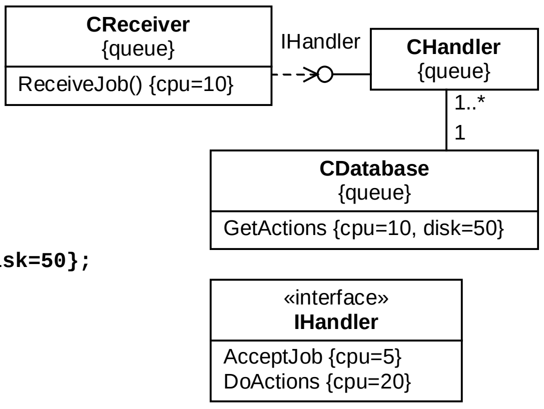


Figure 3. The static structure of the example system.

```
collaboration BgLoad {
  property arrivalrate=0.001;
  1: GetActions();
};

collaboration Jobs {
  property arrivalrate=0.007;
  property adjustopen=1;
  1: ReceiveJob();
  2: AcceptJob();
  2.1: GetActions();
  2.2: DoActions();
};
```

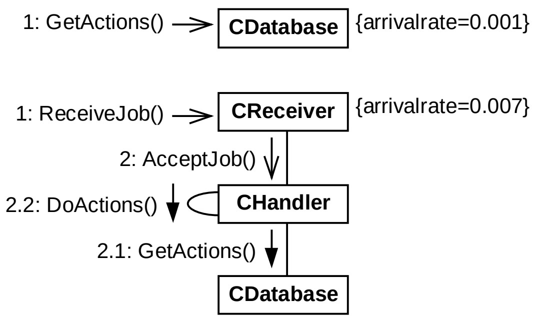


Figure 4. Workloads for the example system.

```
connection LAN {
  property msgpeer=LANMsgpeer;
  Latency : Delay;
};
collaboration LANMsgpeer {
  1: Latency() {d=3};
};

node CNode {
  property msgpeer=CNodeMsgpeer;
  Ctxswitch : Delay;
  Cpu : Queue {d=cpu};
  Disk : Queue {d=disk};
};
collaboration CNodeMsgpeer {
  1: Ctxswitch() {d=2};
};
```

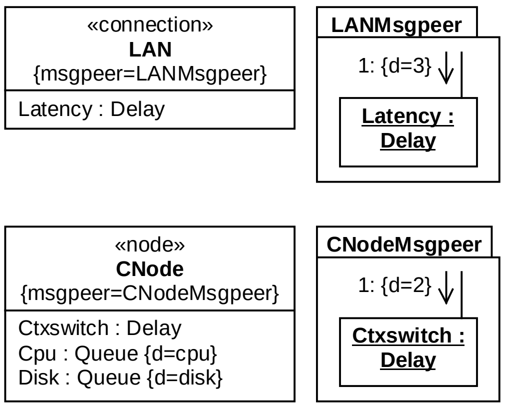


Figure 5. Node and network specifications for the example system.

elements. In steps 1 and 2, the message is received and forwarded to a handler. In steps 2.2 and 2.3, the handler consults the database and executes appropriate actions. We initially estimate an arrival rate of 7 messages per second.

The system uses an object-oriented infrastructure for implementing object communication. To keep the example short, we model the infrastructure with simple communication delays. We assume that there is an average 3 ms delay when the sender and receiver are in different nodes, and a 2 ms context switch delay when they are in the same node. We also model hardware resource contention by presenting explicit CPU and disk resources for all nodes. The service demands for application level operation requests are bound to these resources. The definitions for the network infrastructure and the nodes are illustrated in Figure 5.

We consider two different configurations for the system. The basic configuration has a single server node containing a receiver, a handler, and a database. The advanced configuration has a receiver in one server node and three handlers together with a database in another server node. These configurations are illustrated in Figure 6.

When the basic configuration is transformed into the AQN representation, the result contains six resources and two job classes as illustrated in Figure 7. A few issues are worth noting. First, the actual execution takes place exclusively in hardware resources as a result of service demand binding. Software resources are only controlling the order of accessing the hardware. This is illustrated by the gray color in the figure.

```
MyLAN : LAN {
  Server1 : CNode {
    : CReceiver;
    : CDatabase;
    : CHandler;
  };
};

MyLAN : LAN {
  Server1 : CNode {
    : CReceiver;
  };
  Server2 : CNode {
    : CDatabase;
    [3] : CHandler;
  };
};
```

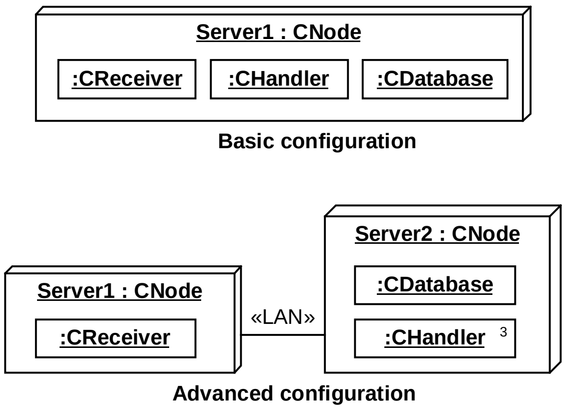


Figure 6. Two configurations for the example system.

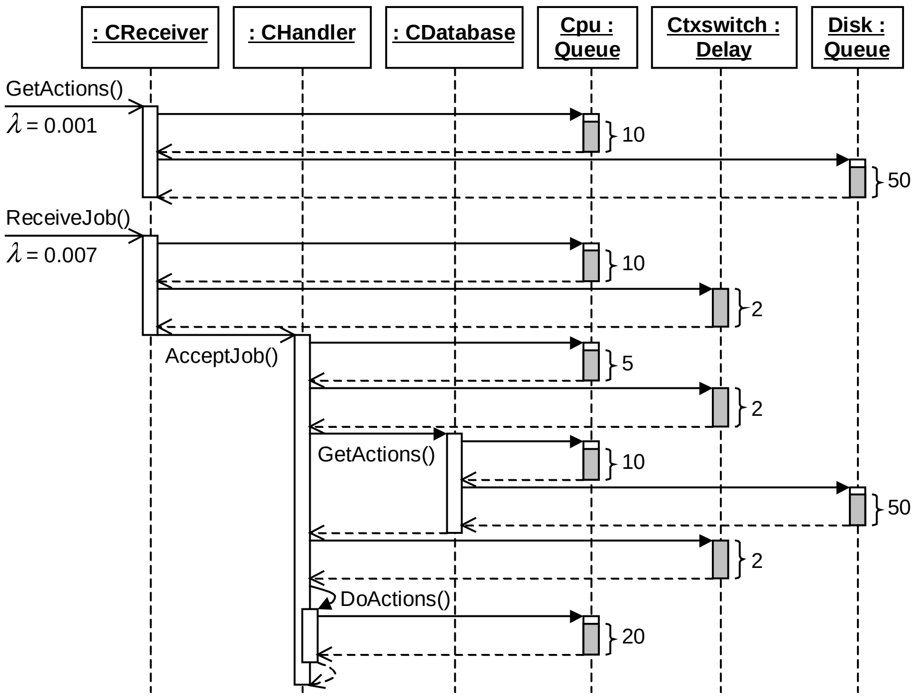


Figure 7. The AQN representation for the basic configuration.

Second, the *Ctxswitch* resource illustrates the effect of triggering properties. It is accessed three times, once for the *AcceptJob* message and twice for the synchronous call *GetActions*. However, there is no context switch for the recursive call *DoActions*.

Finally, the complexity of the diagram is worth noting. The AQN in Figure 7 was generated from a simple model with only three application-level messages in a single node. This way, number of automatically generated messages is small and the sequence diagram in

```
MyLAN.Latency : Delay;                  5.1: Cpu() {d=1.66667};
MyLAN.Server1.$CReceiver : Queue;       5.2: Ctxswitch() {d=0.666667};
MyLAN.Server1.Cpu : Queue;              5.3: GetActions() {d=0};
MyLAN.Server1.Ctxswitch : Delay;        5.3.1: Cpu() {d=3.33333};
MyLAN.Server1.Disk : Queue;             5.3.2: Disk() {d=16.6667};
MyLAN.Server2.Cpu : Queue;              5.4: Ctxswitch() {d=0.666667};
MyLAN.Server2.Ctxswitch : Delay;        5.5: DoActions() {d=0};
MyLAN.Server2.Database : Queue;         5.5.1: Cpu() {d=6.66667};
MyLAN.Server2.Disk : Queue;             6: AcceptJob() {d=0};
MyLAN.Server2.Handler[1] : Queue;       6.1: Cpu() {d=1.66667};
MyLAN.Server2.Handler[2] : Queue;       6.2: Ctxswitch() {d=0.666667};
MyLAN.Server2.Handler[3] : Queue;       6.3: GetActions() {d=0};
                                        6.3.1: Cpu() {d=3.33333};
collaboration BgLoad {                  6.3.2: Disk() {d=16.6667};
  property arrivalrate = 0.001;         6.4: Ctxswitch() {d=0.666667};
  1: GetActions() {d=0};                6.5: DoActions() {d=0};
  1.1: Cpu() {d=10};                    6.5.1: Cpu() {d=6.66667};
  1.2: Disk() {d=50};                   7: AcceptJob() {d=0};
};                                      7.1: Cpu() {d=1.66667};
collaboration Jobs {                    7.2: Ctxswitch() {d=0.666667};
  property arrivalrate = 0.007;         7.3: GetActions() {d=0};
  1: ReceiveJob() {d=0};                7.3.1: Cpu() {d=3.33333};
  1.1: Cpu() {d=10};                    7.3.2: Disk() {d=16.6667};
  2: Latency() {d=1};                   7.4: Ctxswitch() {d=0.666667};
  3: Latency() {d=1};                   7.5: DoActions() {d=0};
  4: Latency() {d=1};                   7.5.1: Cpu() {d=6.66667};
  5: AcceptJob() {d=0};               };
```

Figure 8. The AQN representation for the advanced configuration.

Figure 7 is still readable. However, more complex models would clearly produce large and inconvenient diagrams. This observation provides an additional justification for the proposed modeling techniques.

When the advanced configuration in Figure 6 is transformed into the AQN representation, the result contains twelve resources and 35 messages between them. We use the PML notation to illustrate the results of the transformation in Figure 8. The list of instantiated resources is given before the actual workload messages.

A point worth noting in the advanced configuration is the expansion of the *AcceptJob* message into several resource accesses. There are three *Handler* resource instances in the AQN and the service demand of the *AcceptJob* message is divided evenly between them. In addition, all nested accesses that are made during the activation of *AcceptJob* are divided in three equal shares. Notice that the recursive call *DoActions* is routed to a single handler unlike the *AcceptJob* message. This is because the current job is already accessing one of the three handlers and it does not make sense to route a recursive access to any other handler. This feature

```
Utilization     Type    Device
-----------     ----    ------
0 %             Delay   MyLAN.Latency
53.3116 %       Queue   MyLAN.Server1.$CDatabase
67.7291 %       Queue   MyLAN.Server1.$CHandler
8.37806 %       Queue   MyLAN.Server1.$CReceiver
32.5002 %       Queue   MyLAN.Server1.Cpu
4.20001 %       Delay   MyLAN.Server1.Ctxswitch
40.0002 %       Queue   MyLAN.Server1.Disk

BgLoad
------
Resp.time:  132.338     Throughput: 0.001      Nbr.in system: 0.132338
Time share: 38.4263 %   MyLAN.Server1.$CDatabase
            10.3279 %   MyLAN.Server1.Cpu
            51.2458 %   MyLAN.Server1.Disk

Jobs
----
Resp.time:  314.887     Throughput: 0.007      Nbr.in system: 2.20421
Time share: 1.81783 %   MyLAN.Server1.$CDatabase
            62.4507 %   MyLAN.Server1.$CHandler
            0.347649 %  MyLAN.Server1.$CReceiver
            16.7913 %   MyLAN.Server1.Cpu
            1.90545 %   MyLAN.Server1.Ctxswitch
            16.6871 %   MyLAN.Server1.Disk

collaboration BgLoad {                 // Throughput: 0.001
  property arrivalrate = 0.001;        // Residence times
  1: GetActions() {d=0};               // 50.8526
  1.1: Cpu() {d=10};                   // 13.6677
  1.2: Disk() {d=50};                  // 67.8176
};                                     //*132.338

collaboration Jobs {                   // Throughput: 0.007
  property arrivalrate = 0.007;        // Residence times
  1: ReceiveJob() {d=0};               // 1.0947
  1.1: Cpu() {d=10};                   // 11.9684
  2: Ctxswitch() {d=2};                // 2
  3: LAN.AcceptJob() {d=0};            // 196.649
  3.1: Cpu() {d=5};                    // 6.96839
  3.2: Ctxswitch() {d=2};              // 2
  3.3: GetActions() {d=0};             // 5.72412
  3.3.1: Cpu() {d=10};                 // 11.9684
  3.3.2: Disk() {d=50};                // 52.5454
  3.4: Ctxswitch() {d=2};              // 2
  3.5: DoActions() {d=0};              // 1.34865e-005
  3.5.1: Cpu() {d=20};                 // 21.9684
};                                     //*314.887
```

Figure 9. Example report from the OAT tool.

of the transformation can be overridden by giving explicitly the scope of the invocation in the original collaboration diagram.

Figure 9 illustrates the output from our experimental OAT tool. It corresponds to the basic configuration in Figure 6.

4. Conclusions and future work

We have briefly presented a framework for creating, using, and maintaining performance models of object-oriented distributed systems. The framework consists of four main elements. The method of decomposition provides an algorithm for solving performance models efficiently for complex systems. A set of UML based performance modeling techniques help to raise the abstraction level of the model to suit the needs of software engineering. A performance modeling methodology provides guidelines for using the framework, and an experiment modeling and analysis tool has been created to illustrate the concepts.

The framework is currently at an experimental stage, and the need for various extensions has already been detected. We briefly mention four obvious directions to go. First, the OAT tool only supports the textual PML notation and a natural step is to provide support for graphical modeling. This can be implemented either with self-made graphical extensions or with an existing graphical modeling tool. The use of the XMI format for model exchange is an attractive alternative. Second, the framework does not currently support all commonly used features of UML collaboration and sequence diagrams. In particular, there is no way of starting, killing, and synchronizing threads. A better support for threads would require new UML based modeling techniques and corresponding additions to the MOD algorithm. Third, UML state and activity diagrams are currently not used by the framework but they can express performance related information in a convenient manner. For example, activity diagrams could be used to combine multiple collaboration diagrams together in a complex interaction. This way, workload diagrams would not grow excessively. Finally, the current approach limits the available scheduling disciplines, service time distributions, and arrival rate distributions. These limitations are mostly inherited from the MVA algorithm that we use during the method of decomposition. However, approximate techniques exist for extending these limitations (e.g. [4]) and they could be integrated into the MOD algorithm.

References

[1] Kähkipuro, P., The Method of Decomposition for Analyzing Queuing Networks with Simultaneous Resource Possessions, In Proceedings of the Communications Networks and Distributed Systems Modeling and Simulation Conference (CNDS'99), The Society for Computer Simulations International, San Diego, California, 1999.

[2] Kähkipuro, P., UML Based Performance Modeling Framework for Object-Oriented Distributed Systems, In France, R., Rumpe, B. (eds.), «UML»'99 – The Unified Modeling Language : Beyond the Standard, LNCS 1723, Springer-Verlag, Berlin Heidelberg, Germany, 1999.

[3] Object Management Group, A Human-Usable Textual Notation for the UML Profile for EDOC, Request for Proposal, OMG Document ad/99-03-12, Framingham, Massachusetts, 1999.

[4] Agrawal, S.C., Metamodeling: A Study of Approximations in Queuing Models, MIT Press, Cambridge, Massachusetts, USA, 1985.

[5] Haverkort, B.R.: Performance of computer communication systems: a model-based approach. John Wiley & Sons, New York, New York, USA, 1998.