Performance Aware Design of Communication Systems

Lukas Pustina, Michael Gerharz, Peter Martini, Simon Schwarzer
Institute of Computer Science IV, University of Bonn
D-53117 Bonn, Germany
{*gerharz,martini,pustina,schwarzer*}*@cs.uni-bonn.de*

Volker Deichmann
Nokia Research Center
D-44807 Bochum, Germany
volker.deichmann@nokia.com

Abstract

In this paper we present a methodology for a performance aware design of communication systems including protocols and devices. The goal is to evaluate the performance already in early stages of the design process to avoid costly re-designs of bottlenecks in finished products. From protocol specifications given as sequence diagrams, we derive multiclass queueing networks as a means to estimate the performance of the system architecture before any executable prototype exists. This makes queueing theory accessible to system and protocol designers even if the designer is not familiar with details of queueing network theory. The methodology supports an easy evaluation of design alternatives without affecting the functional model. To achieve this, it incorporates the performance information into the system model in a non-invasive fashion, such that the system model remains meaningful without this information.

1 Introduction

Due to increasing complexity, performance aspects become more and more important in system design. In communication systems, performance aspects are important on two different levels. First of all, end-to-end performance determines the user experience and allows for network resources planning. On a more detailed level, the performance of single devices is important to guarantee that each component on an end-to-end path is able to reach the required service level. This includes network elements as well as end user devices. For example, in a video conferencing scenario it is required that, on the one hand, the end devices, e.g. mobile phones, are powerful enough to decode the video stream in real-time and, on the other hand, the network delay is small enough.

Therefore, it is important to consider these performance aspects already in early phases of the system design (e.g. when developing a new hardware architecture) when executable prototypes are not yet available. This is the purpose of the research area of performance engineering.

Existing approaches to performance engineering focus on one performance aspect only. For example, to estimate the end-to-end performance of a network running a certain protocol family (such as e.g. video conferencing), queueing network analysis and simulation are well-established means. However, the performance of a network device is often analysed by using a detailed emulation of the hardware which prohibits reasonable performance estimates in early stages of a development.

Therefore, we have derived a performance engineering framework which makes queueing network theory accessible for device development as well. The framework is based on the fact that message sequence charts are a common way to describe communication protocols and algorithms. As UML is todays de-facto standard for system modelling, we propose a modelling approach which utilises UML sequence diagrams to specify the system behaviour (i.e. protocol procedures or algorithm functions). Hardware platforms for the devices running the protocols are defined using UML composite structure diagrams.

A major benefit of this UML based approach is that the power of queueing network theory becomes available to system designers unfamiliar with the details of this theory. A further benefit of this approach is that the system can be analysed on different levels of detail using the same methodology, e.g. modelling and analysing a single device as well as the complete system on an end-to-end basis. Furthermore, design alternatives may easily be evaluated without the need to build several prototypes.

The methodology we developed derives multiclass queueing networks from the given UML diagrams. A queueing network consists of interconnected queueing centers which process incoming jobs from one or several workloads. The processing is characterised by a workload dependent, stochastically distributed service time. The interconnection may include branches and feedbacks forming a loop. In order to distinguish between different visits of a job to the same queueing center, multiclass queueing networks assign a job class to each job.

1-4244-0419-3/06/$20.00 ©2006 IEEE

In the following, sec. 2 summarises related work in the context of queueing network based performance engineering. Sec. 3 describes our approach to modelling a system which is based on the fact that sequence diagrams are widely applicable in protocol design. Sec. 4 outlines our algorithm to derive queueing networks from a UML based system specification. In sec. 5 we present first results generated by our prototypical implementation. Finally, sec. 6 summarises the paper and outlines perspectives for future work.

2 Related Work

In their pioneering work of integrated performance analysis Williams and Smith introduced the usage of queueing networks in their SPE (Software Performance Engineering) approach [17, 18]. Since then, the usage of UML as a modelling language in the context of performance analysis has been proposed by several authors [2, 5, 7, 8, 16, 13, 19, 12, 17, 6] and may be considered as a de-facto standard. Therefore, we adopt the idea of generating performance models from a set of UML diagrams for performance analysis in early stages of development.

For modelling non-functional hardware and software aspects, we adopt the UML Profile for Schedulability, Performance, and Time Specification (SPT profile) [14], which also was proposed in several papers [19, 3, 2, 11].

Concerning the performance analysis, different kinds of performance models have been proposed, e.g. execution graphs used in [18, 17, 7, 6], petri nets utilised in [5, 13], (extended) queueing networks instrumented in [2, 18, 17, 7, 12, 6, 11], or layered queueing networks [19, 15]. A good overview of different methodologies, categorised amongst others by the kind of performance model, the modelling language used, and – in the case of UML – the diagrams chosen is given in [1].

In this paper, we concentrate on a methodology that automatically derives queueing networks from UML 2.0 diagrams annotated according to the SPT profile. The presented methodology has been implemented and can be used seamlessly with UML.

A range of other publications follows a similar approach (see above). In [11], the UML 2.0 component diagram is used to specify properties of the components which are to be transformed into queueing centers, but the methodology focuses on software aspects only, while the hardware of the developed system is not considered.

The authors of [2] propose the usage of multiclass queueing networks derived from UML models using use case, activity and deployment diagrams. Software and hardware aspects can be modelled with these diagrams and are transformed into a queueing network. Using use case diagrams for specifying scenarios and the involved workloads is a widely used and accepted practise also applied by [7, 6].

In contrast to [2, 7, 6], we use the new composite structure diagram of UML 2.0 for architecture modelling. It supports an evolution of the model by a hierarchy of refining diagrams during the development process.

Moreover, we use sequence diagrams for modelling the behavioural aspects similar to [7, 6, 18, 19, 11] which use either generic message sequence charts or also UML sequence diagrams. In contrast to using activity diagrams which require an annotation of the required resource to the action [2], our approach employs composite structure diagram to associate software with hardware by reusing the system components defined in class diagrams. In this way, the mapping may be graphically performed in the model and developers need to change only one diagram resulting in an automatic propagation of this change to all related diagrams.

3 Modelling

In this section, we introduce our methodology to model communication systems based on UML 2.0 and the UML SPT profile (cf. sec. 2). Note that the SPT profile has been specified for UML 1.x, for obvious reasons not taking care of changes introduced by UML 2.0. A UML 2.0 compliant successor of the SPT profile is currently not available. Therefore, we adapted the current profile where necessary. The methodology proposed enforces a strict separation of the *system model* itself (cf. sec. 3.1) and the corresponding *performance model* (cf. sec. 3.2) in the sense that performance information is integrated within the model in a *non-invasive* manner. Following this clear separation, it is possible to maintain the system model independently from the performance aspects so that the system model remains meaningful without the performance information. Furthermore, the effects of altering the properties of a certain component may easily be evaluated without the need to modify the system model.

The methodology supports an evaluation of different *performance scenarios* (cf. sec. 3.3) in a straightforward way without the need to modify either the system model or the performance model. This is achieved by defining a set of use cases (the scenarios of interest) and defining workloads for each scenario. Furthermore, by specifying which part of the system should be analysed, it is possible to evaluate the performance of the end-to-end system, certain network elements, or even specific components of a single device.

For the presentation of the material we use a common use case which is a video conference run on a mobile phone using WLAN. Basically, we inspect the data decompression on a single mobile. The mobile consists of a general multipurpose processor (MPU), a digital signal processor (DSP) for decompression, and a display to show the video stream.

The data is received from a WLAN access point. If a transmission error occurs, the frame will be resent. The MPU passes the data to the DSP which decompresses each packet in several iterations. Finally, the packet is sent to the display.

3.1 System Modelling

For modelling the system, we follow the common approach (e.g. [9]) to distinguish between the components of the system, the topology of the system, i.e. the interconnection of the components, and the dynamic behaviour of the system.

The system components are modelled by using class diagrams as in the example shown in fig. 1. It is reasonable to include an explicit definition of the interfaces the class provides. Interfaces may be defined in UML using the stereotype `<<interface>>` and attached to a class as "required" and "implemented" interfaces depicted by lollipops.

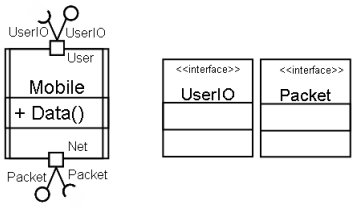

Figure 1: *Class Diagram describing the mobile component*

The system topology which specifies how the components are interconnected is modelled by using composite structure diagrams (cf. fig. 2 and 3 for an example). In UML terminology, the elements (components) of a composite structure diagram are interpreted as objects which are instances of the specified classes. Thus, the class and interface definitions modelled as mentioned above may be reused. Components are interconnected by links and support the usage of the specified interfaces. Of course, multiple instances of the same class may be interconnected.

It is possible to refine a class by a separate composite structure diagram specifying its internal structure. This approach results in a hierarchy of composite structure diagrams. Fig. 4 shows such a hierarchy with the root CSD `RootCSD` and the refinement composite structure diagram `Mobile`; the diamonds represent components without a refinement. We call the top-level diagram the *root composite structure diagram* (root CSD). From the root CSD all other components are reachable in the component structure diagram hierarchy in the sense that they are either parts of the root CSD itself or of a refining composite structure diagram.

From a design perspective, it is desirable that different design alternatives of a specific component can be easily evaluated (i.e. of its refining composite structure diagram) without the need to maintain two versions of the complete model. Note that this is only possible with a clear separation of the component functionality specified in the class diagram and the architectural properties (including capacity values, see below) specified in the composite structure diagram.

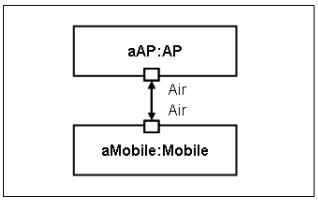

Figure 2: *Composite Structure Diagram describing the root CSD*

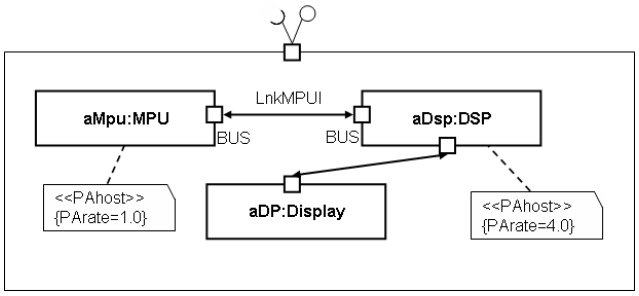

Figure 3: *Composite Structure Diagram describing the internal structure of the mobile component*

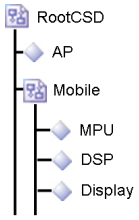

Figure 4: *Hierarchy tree showing the root and a refinement CSD*

The dynamic system behaviour is modelled by using sequence diagrams to specify use cases of the system. This is a well-established approach for modelling the message exchange between different entities in a closed system or in a network. Again, the classes defined in the class diagrams may be reused and instantiated as objects in sequence diagrams. Thus, the objects may exchange messages defined as methods in classes or interfaces (cf. fig. 5). Note that objects exchanging messages do not need to be directly connected in the system topology, but may communicate through intermediate components as well. In this case, it must be ensured that a communicating path in the system topology exists and routing information must be gathered

in order to establish a path between the sender and the receiver. This is discussed in more detail in sec. 4.1.

UML 2.0 includes so called *interaction frames* which may be used to model complex control flows in sequence diagrams. These interaction frames include `alt`-frames (alternative execution), `loop`-frames (repeating execution), `ref`-frames (inclusion of other sequence diagrams) et al. The `par`-frame defining parallel execution with forks or joins, is excluded in our methodology, because there is no equivalent representation in queueing networks. [11] suggests to use so called "Extended Queueing Networks" to incorporate parallelism. This approach could be easily integrated into our simulative evaluation (cf. sec. 4.3). Note however, that a valid way to model parallelism is to use several use cases in a performance scenario (cf. sections 3.2 and 3.3) resulting in multiple workloads each with its own jobs.

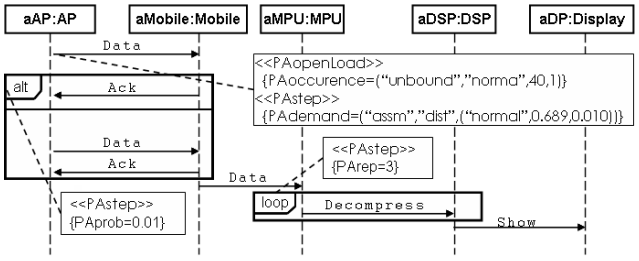

Figure 5: *Sequence Diagram describing data receipt ion and processing*

3.2 Performance Modelling

A performance model describes the non-functional aspects of a system with respect to its performance. Two steps are necessary to fully specify a performance model. First, the performance capacity of each component has to be configured, and secondly, the resource demand of each action defined for the functional behaviour has to be provided for each component and each use case.

3.2.1 Modelling the Component Capacities

Similar to the system topology, the specification of the capacity of each component is an important and at the same time difficult part of the design decisions. In order to evaluate different topologies, the composite structure diagram was selected for topology modelling because of its flexibility and the possibility to easily exchange and compare it with other topologies. Thus, it is sensible to define the capacity of components in the corresponding composite structure diagram as well. In this way, the configuration of the components is as exchangeable as the topology itself. Thus, a designer is able to exchange entire parts of the architecture at once. The alternative approach to incorporate the performance capacity into class diagrams would require to change the model at two different places in order to alter the architecture.

For the component capacity specification, we use the SPT profile tagged value `PArate` which is part of the `<<PAhost>>` annotation. It allows us to specify the component capacity as a relative speed value compared to a reference component with a default value of `PArate=1` (this component does not need to be part of the diagram). If a component provides this reference speed, the annotation may be omitted.

To obtain absolute component capacities, the relative speed values are multiplied along the composite structure diagram hierarchy. As the system topology evolves from the hierarchy of composite structure diagrams, the component capacities should affect all refinement components as well. Thus, a speed-up of a component results in speeding up all its refinement components as well. This allows the designers to change the architecture of either the whole subsystem or merely specific components if applied on the lowest layer.

This inheritance approach is more sensible than specifying absolute capacities directly in the `PArate` values, because a desired change of the architecture would impose changes on every single sub-layer as well.

In order to transform the system model into a queueing network, it is also necessary to specify the scheduling policy of each component. The tagged value `PAschdPolicy` is used for this purpose. It is added to the `<<PAhost>>` annotation in the same way as the `PArate` tagged value. Because the `FIFO` service discipline is the most common one, it is used as the default policy.

3.2.2 Modelling the Resource Demand

The second step of performance modelling is the modelling of the resource demand resulting from the functional behaviour of the system executing a given use case. Note that each message defined in a sequence diagram triggers an action on the receiving component and thus, uses resources on that component. Therefore, in this step it is ultimately required to provide estimates for the *resource demand* of every action defined in the sequence diagrams which specify the functional behaviour of the system.

According to the SPT profile, resource demand is specified by the `PAdemand` tagged value inside a `<<PAstep>>` annotation. The demand is represented by a triple consisting of a source modifier specifying how the demand was captured (e.g.measured or estimated), a type modifier giving the type of the value (e.g. average value or distribution), and a time value which is the actual service time and can

also be expressed by a probability distribution (for further details refer to the SPT profile[14]) (cf. `PAdemand` annotation in fig. 5). To get the actual resource consumption from this demand, the demand value has to be divided by the speed factor of the component defined above (cf. sec. 4.2).

In a straightforward approach, every message in a sequence diagram (describing a certain use case) would be annotated with a `<<PAstep>>` definition specifying the resource demand triggered by that action. However, the authors of [11] propose a solution that allows for the specification of a *default resource demand* for each component. For this purpose, a component in a composite structure diagram may be annotated with a `<<PAstep>>` value such that whenever an action on this component is triggered, the default resource demand is assumed unless a *scenario dependent resource demand* is specified in the sequence diagram. Note that this use of the `<<PAstep>>` annotation is not SPT profile compliant.

3.3 Performance Scenario

The system and performance modelling steps described in sec. 3.1 and 3.2 show how to specify the functional and the non-functional aspects of a system with the focus on performance evaluation. Next, we address how to use these models to actually evaluate the performance of a given system.

The components to be evaluated vary depending on the developer's perspective and interests which determine the *performance scenario* to be examined. A performance scenario consists of the system topology description and one or more separate use cases. The starting point of a single use case is its *root step* which does not necessarily need to be the first message specified, but depends on the designer's interest. The SPT profile stereotypes `<<PAopenLoad>>` and `<<PAclosedLoad>>`, which assign a workload (basically an arrival process for the annotated message), determine the root step and must be unique in each sequence diagram.

The *processing composite structure diagram* (processing CSD) of a performance scenario is the composite structure diagram which, together with its refinement CSDs, describes the topology of the components participating in the selected use cases. Thus, it is possible to evaluate only a part of the whole system by selecting a subset of the use cases and a processing CSD representing the first hierarchical layer of the topology of the system part. The developed methodology supports this composition of performance scenarios as proposed by the SPT profile. According to the interests of the designers, they incorporate several sequence diagrams to one performance scenario in order to evaluate the effects of the parallel execution of these diagrams and the resulting concurrent demands.

4 Transformation

In this section, we present an algorithm using the system model and the performance scenario definition as described in the previous section to generate a multiclass queueing network. This queueing network may be evaluated by third party tools in an analytical or simulative way.

The transformation algorithm consists of four steps as depicted in fig. 6. For each use case specified in the performance scenario, the *chain of actions* is identified which basically describes the flow of execution triggered by the reception of messages in the sequence diagrams. Thereafter each action is *mapped to the components* performing the action by evaluating the composite structure diagram hierarchy. This information is used to specify the *queueing network* for this specific use case. Steps 1–3 are repeated for each use case until finally, in step 4, all individual queueing networks are *merged to the resulting queueing network* covering the whole performance scenario. For this purpose, multiclass queueing networks are used.

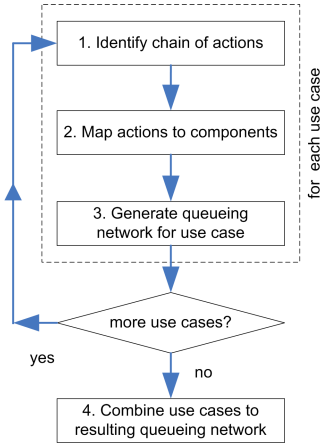

Figure 6: *The four steps of the transformation algorithm*

4.1 Identifying the Chain of Actions

In the first step each sequence diagram associated with the involved use cases is transformed into what we call a *chain of actions*. Note that every message in a sequence diagram corresponds to an action in the receiving object triggered by the reception of that message. The corresponding chain of actions for the use case described by the sequence diagram in fig. 5 is depicted in fig. 7. Following the sequence of messages, a chain of such actions evolves. Note that loops and alternatives are independent elements in the chain of actions which are resolved in a later step (cf. 4.3). Finally, for each action the resource demand is identified according to their `<<PAStep>>` annotation (cf. sec. 3).

Figure 7: *Chain of actions for data reception use case*

As mentioned in sec. 3.1, our methodology supports both *direct* and *indirect* message exchange. In case of an indirect message exchange, i.e. when the communicating components do not share a direct link in the system topology, a route from the source of the message to its destination component must be determined, because resources are consumed on these components as well. (Note that the designer must ensure for the existence of such a route.)

In general, such a route may be discovered using common routing algorithms such as Dijkstra, Bellman-Ford or a simple breadth first search. In that case, the components of the composite structure diagrams would correspond to the nodes inside a graph for which the links are defined by the connections between the components. A drawback of this approach is that the selected route might not necessarily be the desired one if several alternative routes exist (even if it is optimal under a specific metric). To leverage this situation, additional information could be incorporated into the model to guide the algorithm in discovering the desired route, e.g. by specifying required and implemented interfaces in intermediate components. However, note that this additional information violates the non-invasive principle since it modifies the system model.

If a message is exchanged indirectly, the implicit actions on the intermediate components are additionally inserted into the chain of actions in front of the message target. Since these intermediate components are not annotated with a use case specific demand in the currently processed sequence diagram, the default resource demand as defined in (cf. sec. 3.2.2) is used instead.

4.2 Mapping Actions to Components

Step 2 takes the elements from the chain of actions and maps each of them to its executing component which may be derived from the hierarchy of composite structure diagrams. As the resource consumption is analysed on the level of the *processing CSD*, all components in sub-diagrams have to be mapped to their containing component in the processing CSD. In the video conference example, the components are mapped to the `Mobile` composite structure diagram depicted in fig. 2 which is used as the processing CSD. For additional components which the processing CSD does not contain but which are required for the use case definition (e.g. source and sink in an end-to-end scenario), the corresponding components are derived from the root CSD or sub-hierarchies thereof. In the example, this is necessary for the access point component `AP`.

Additionally, in order to configure the queueing network properly the capacity of each processing resource has to be determined. From the hierarchy of composite structure diagrams, a component's capacity may be calculated from the `PArate` tagged value as described in sec. 3.2.1, i.e. the relative speed values propagate from the processing CSD down to the refinement CSDs by multiplying the respective values.

The result of this step is the sequence of processing resources in the way they are "visited" in the current use case and the resource demand that is generated by the actions corresponding to that step. Thus, each step of the sequence is annotated with the corresponding resource demand as well as the capacity of the associated component.

4.3 Generating a Queueing Network for a Single Use Case

In step 3, the sequence of processing demands and components generated in step 2 is transformed into a queueing network corresponding to the original use case. First of all, each participating component is transformed into a queueing center. The service times for each queueing center are calculated by dividing the resource demand as derived in the chain of actions and the resource capacity as derived in step 2. The scheduling policy of the queueing centers is taken from the `PAschdPolicy` tagged value of the processing CSD `<<PAhost>>` annotations.

The connection of the queueing centers, which determines the possible paths a job may take through the queueing network, is derived from the chain of actions. In a first step, the structural elements identified in step 1, i.e. loops, alternatives and plain actions, are connected as specified in the sequence of processing resources. In a second step, loops and alternatives are resolved. A special treatment is also required for *implicit loops* in which the same resource is visited twice or more in the sequence of processing resources. These revisits evolve from either another message sent to the same component in the sequence diagram or by mapping different refinement components to the same processing CSD component. In contrast to implicit loop, *explicit loops* are explicitly specified in the sequence diagrams using the `loop`-frame. Alternatives, specified using `alt`-frames, distribute the jobs to several subsequent queueing centers according to branching probabilities provided in the sequence diagram.

The actual transformation of these structural elements heavily depends on the desired type of queueing network evaluation. For an analytical evaluation, currently only product form queueing networks[10] which conform to the BCMP rules[4] are supported. Non-product form queueing networks are supported by simulation.

4.3.1 Structural Transformation for Analytical Evaluation

If the queueing network is in product form, an analytical evaluation may be performed by considering each queueing center separately. In this case, it suffices to determine the service demand on each component. According to the forced flow law (cf. [10]) the service demand D_i of a job in a queueing center i is the product of the service time S_i for that job and the number of visits V_i of this job to that queueing center. Thus, for each queueing center, we have to determine how often it is visited by each job on average.

$$D_i = V_i \cdot S_i \tag{1}$$

Implicit loops correspond to revisits of a particular queueing center and increase the number of visits by 1. Explicit loops are handled in a similar way by increasing the number of visits by the number of repetitions of the respective loop. Since explicit loops may span several queueing centers, the number of visits must increase for each center. In case of an alternative, the corresponding queueing centers are visited with a certain probability. Thus, the number of visits is increased by this probability. Note that an alternative may span several queueing centers.

4.3.2 Structural Transformation for Simulative Evaluation

In contrast to the analytical simulation, the sequence of the queueing centers is very important for the simulative model, because it determines the paths a job can take to move through the network. Thus, to generate the structure of the queueing network it is necessary to resolve the three structural elements, i.e. implicit loops, explicit loops, and alternatives, in such a way that each single path through the network possibly initiated by the current workload is available in the resulting queueing network for this scenario.

As mentioned above, loops essentially result in revisiting the same queueing center several times. To be able to service each visit of a job of the same workload with a different service time, it is necessary to distinguish between each visit. For this purpose, we use the concept of multiclass queueing networks. Basically, the job class of a job needs to be increased for every loop iteration so that a different service time may actually be used for each iteration.

For the implementation of implicit loops, the transformation needs to reconnect the outgoing link of a queueing center for the incoming job class to the input of the same queueing center for the increased job class. Concerning explicit loops, the transformation must ensure that all queueing centers that are part of the loop are connected. The outgoing link of the last queueing center of the loop needs to be reconnected to the input of the first queueing center of the loop. This way, the jobs of the job class to loop are continuously rerouted back to the loop start until they finished the last repetition after which they leave the last queueing center of the loop to the remaining queueing network.

The conversion of an alternative exploits the fact that each individual branching connection to a sub-queueing network can be handled like a regular connection. Before the jobs leave the queueing center in front of an alternative, they are pseudo-randomly assigned to one of the branches according to the given probabilities. Additionally, the output of each branch is connected to the queueing center succeeding the alternative. Here, it is necessary to recombine the split job stream by assigning a uniform job class to every job leaving the different branches.

Fig. 8 shows the result of the transformation of the video conference scenario (cf. fig. 5).

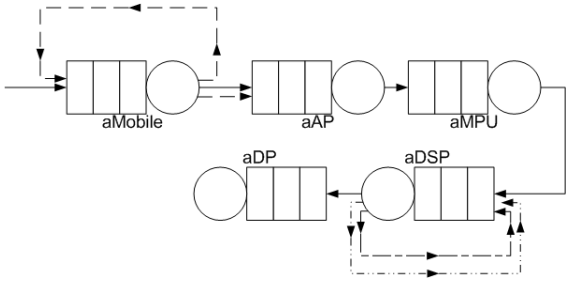

Figure 8: *Single queueing network for data reception use case*

4.4 Combining Use Cases to the Resulting Queueing Network

The last transformation step merges the use case specific queueing networks generated in step 3 to a unified queueing network representing the entire performance scenario. In this step, it has to be assured that the job classes assigned in the use case specific queueing networks are mapped to unique job classes of the unified queueing network. This allows the queueing centers to distinguish between the use cases, and thus to process jobs of different workloads with the appropriate service times. In the video conference example, there is only one use case, and thus the resulting queueing network equals the single queueing network depicted in fig. 8.

5 First Results

This section presents first results generated by our prototypical implementation. The performance values used here are not based on real world measurements and are simply used to demonstrate the applicability of our methodology. A sophisticated validation is out of scope of this paper and will be part of future work.

The evaluated performance scenario consisted of the use case described by fig. 5 only. The mobile device receives video frames from the AP, decodes them in the MPU, iteratively decompresses them by the DSP, and finally shows them on the display. In case of a transmission error, the missing ACK forces an immediate retransmission by the AP. For clearness, the workload has been annotated in this diagram as well. Tab. 1 presents the performance values and some of the results given by our implementation for the DSP component.

Element	**Parameter**	
Workload	dist.: normal(40, 1)	25 fps
loop	repetition: 3	
aDSP	dist.: normal(10, 1)	service time
Results		
Utilisation	75%	
Avg. queue len.	1.75	

Table 1: *Performance data for DSP component*

In this example, the utilisation of the DSP is 75% and the average queue length is 1.75. These results may be easily validated by manual calculations.

6 Conclusion & Further Work

In this paper, we have introduced a performance engineering framework which allows the evaluation of communication systems in early phases of their development. The methodology is based on using UML sequence diagrams to model protocol behaviour and use cases. From these diagrams, multiclass queueing networks are derived which can either be simulated or analysed mathematically such that the power of queueing network theory becomes available to system designers not familiar with the details of this theory.

For specifying hardware properties, UML composite structure diagrams have been proposed which allow a system to be designed in a hierarchical manner. Performance relevant information is incorporated into the model in a non-invasive fashion leaving the system model independent from the actual performance scenario.

This way, design alternatives may be easily evaluated and compared. Furthermore, the system may be analysed from different perspectives, focusing either on the systems end-to-end performance, the performance of single devices running the specified protocols, or a combination of both. Different abstraction levels are possible for different devices on an end-to-end path.

An implementation of the presented methodology exists. Future work will focus on a seamless integration into existing UML environments to enhance its practical applicability in protocol and system design. In this context, it is also envisioned to make use of a more detailed functional model provided by UML state charts.

References

[1] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 2004.

[2] S. Balsamo and M. Marzallo. Performance evaluation of uml system architectures with mutliclass queueing network models. In *WOSP*, 2005.

[3] S. Balsamo, M. Marzolla, A. D. Marco, and P. Inverardi. Experimenting different software architectures performance techniques: a case study. In *WOSP*, 2004.

[4] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 1975.

[5] S. Bernardi, S. Donatelli, and J. Merseguer. From uml sequence diagrams and statecharts to analysable petri net models. In *WOSP 2002*, 2002.

[6] V. Cortellessa, M. Gentile, and M. Pizzuti. Xprit: An xml-based tool to translate uml diagrams into execution graphs and queueing networks. In *QEST*, 2004.

[7] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In *Workshop on Software and Performance*, 2000.

[8] V. Cortellessa and R. Mirandola. Prima-uml: a performance validation incremental methodology on early uml diagrams. *Sci. Comput. Program.*, 2002.

[9] ETSI. *UML Profile for Communicating Systems (draft)*. 2005.

[10] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, New York, 1991.

[11] A. D. Marco and P. Inverardi. Compositional generation of software architecture performance qn models. In *WICSA*, 2004.

[12] M. Marzolla and S. Balsamo. Uml-psi: The uml performance simulator. In *QEST*, 2004.

[13] J. Merseguer and J. Campos. Software performance modeling using uml and petri nets. In *MASCOTS Tutorials*, 2003.

[14] OMG. *UML Profile for Schedulability, Performance, and Time Specification: Version 1.0*. Object Management Group, 2003.

[15] D. C. Petriu and X. Wang. From uml descriptions of high-level software architectures to lqn performance models. In *AGTIVE 1999*, 2000.

[16] R. Pooley and P. King. The unified modeling language and performance engineering. In *IEE Proceedings — Software.*, 1999.

[17] C. U. Smith and L. G. Williams. *Performance Solutions, A Practical Guide to Creating Responsive, Scalable Softwarre*. Addison-Wesley Pearson Education, 2001.

[18] L. G. Williams and C. U. Smith. Performance evaluation of software architectures. In *WOSP*, 1998.

[19] J. Xu, C. M. Woodside, and D. C. Petriu. Performance analysis of a software design using the uml profile for schedulability, performance, and time. In *Computer Performance Evaluation / TOOLS*, 2003.