Towards a Semantic Framework for Non-functional Specifications of Component-Based Systems

Steffen Zschaler
Dresden University of Technology
e-mail: Steffen.Zschaler@inf.tu-dresden.de

Abstract

It is now widely recognized that the so-called non-functional or extra-functional properties of a software system are at least as important as its somewhat more classical functional properties and that they must be considered as early as possible in the development cycle in order to avoid costly failures. In this paper we define elements of a semantic framework for non-functional specifications of component-based systems. We focus on how the runtime environment uses components, whose non-functional properties have been specified, and the available system resources to provide a service with specified non-functional properties.

1. Introduction

It is now widely recognized that the so-called non-functional or extra-functional properties of a software system are at least as important as its somewhat more classical functional properties and that, therefore, they must be considered as early as possible in the development cycle in order to avoid costly failures [7]. Operating systems research—especially research in the area of real-time systems—performance analysis and prediction research, and research in security have produced a wealth of results describing how to analyse, predict, or guarantee selected non-functional properties of applications (cf. e.g., [15, 26, 28, 29]). However, all these approaches work at a rather low level—for example, they use concepts like tasks, periods, memory pages, queuing networks, etc. These concepts on their own are not sufficient to model today's increasingly complex software systems.

Component-based software engineering [27] offers a way to partition complex systems into well-defined parts. The relevant properties of these parts are precisely specified, so that these parts in principle can be developed independently and assembled at a later time, and possibly even by a different person than the original developer. Functional issues of component-based software engineering have been well investigated (e.g., [6, 27]), but very little research has been performed concerning non-functional properties. In particular, the higher-level component-based concepts must be mapped to the lower-level concepts already available in the literature.

In our work, we investigate this mapping by providing a semantic framework which explains how the different parts of a component-based system work together to deliver a certain service with certain non-functional properties. The contribution of this paper is to define the basic concepts of this framework. In doing so, we do not strive to reinvent the theories already available, but rather to enable them to be plugged into the framework when and where they are needed. This way we hope to achieve two objectives: To allow application developers to use component-based software engineering to structure their applications and thus lower the complexity of the software development process while at the same time enabling them to make use of tried and tested theories for providing non-functional properties of those applications. Our approach aims to enable the various roles participating in application development to write specifications independently of each other, while allowing for these specifications to be composed to a global system view for analysis. Thus, we enable the specification and development of applications to be distributed over time and development teams, which is of increasing importance as software systems get more and more complex.

A precise and formal semantic foundation for these concepts serves multiple purposes: First, it aids in clarifying the concepts themselves, and their relations. Second, it allows application developers to create precise and unambiguous models of the systems they develop. Last but not least, a semantic foundation forms a basis for providing tool-support to application developers. Such tool support can come in two main forms: *Analysis tools* help application developers to analyse their models and to spot problems early in the development cycle. *Refinement tools* help application develop-

ers while developing the models by providing decision support in refinement steps. The notion of a *feasible system* described in this paper is an example for a property that should be checked by an analysis tool. We have described an example for tool-supported refinement in [20].

There are two sides to providing non-functional properties of component-based systems:

1. Component developers must *implement* components in such a way that they have determinable non-functional properties.
2. Application assemblers and the runtime system must *use* these components so that the non-functional properties required from the application can be guaranteed.

For example, we will never be able to make any guarantees about the memory consumption (or time for data retrieval) for a FIFO queue component which was implemented using a linked list without any limits on its maximum size, but, even if the queue was implemented with a fixed-size array of length 64 kB, we can still use it in such a way that it consumes 256 kB of memory—by creating four instances. In our research, we are not interested in how components must be implemented so that their non-functional properties become determinable; these questions are only contingently related to components, but the real issues are completely independent of component-based software. Instead, our approach assumes such components with determinable non-functional properties to be available. Based on this, we provide a semantic framework, which allows

- component developers to describe the non-functional properties of these components, and
- application assemblers to describe how these components are used to provide guaranteed non-functional properties of an application.

Note that, although we are talking about non-functional properties of component implementations, we will—as above—very often simply talk of components where it is clear what we mean.

The remainder of this paper is structured as follows: We first present an overview of our system model in Sect. 2, before explaining the details of non-functional specifications in Sect. 3. After some notes on related work, we summarize the paper and provide an outlook on future work.

2. System Model

In this section we introduce the system model underlying the semantic concepts for non-functional specifications. We want to support component-based software and in particular the component-specific scheduling and usage decisions made by the runtime environment of the components

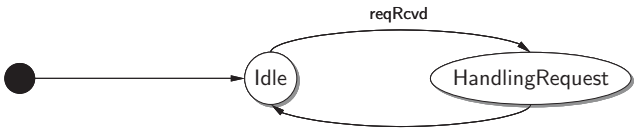

Figure 1. Basic model of a service.

(the container). Hence *components* and the *container* are integral parts of our system model. Because we are talking about non-functional properties, available *system resources* must be present in our model, too. Finally, non-functional requirements on systems are never expressed in terms of components or containers, but rather in terms of the services provided by the system. Therefore, we also introduce the concept of a *service* in our model. We will describe these four basic concepts in more detail in the following sections.

2.1. Services

Users view a system in terms of the services it provides. They do not care about how these services are implemented, whether from monolithic or component-structured software. A service is a causally closed part of the complete functionality provided by a system. As various authors [14, 23] have pointed out, we can model services as partial specifications of a system. Multiple services can then be combined into a total specification of the system's functionality. Users associate non-functional properties with individual services, for example, they will talk about the frame rate provided by a video player service independently of the response time of a cast query for that same film. So, from the user's perspective, the non-functional properties of individual services should be described independently. Note that this does not imply that the non-functional properties of two services cannot interact, for example, because their respective implementations run on the same system and share the same resources. However, although users may be able to specify preferences on services, indicating which service should prevail in case of resource contention, they need to be able to describe their non-functional requirements independently for each service.

We use an execution model of a service which distinguishes those execution phases that differ in their resource usage. Therefore, transition systems are an adequate approach for formalizing this model. Using them in all our models has the added benefit that it allows for easy composition using theorems as those described in [2]. A state machine for our service model can be seen in Fig. 1. At the moment we use a very simple model that essentially identifies a service with a single operation that a client can invoke. Each service can be in one of two states:

`Idle` or `HandlingRequest`. It moves from `Idle` to `HandlingRequest` when it receives a triggering event (denoted by `reqRcvd`—short for "request received"—in the figure); this corresponds to the operation call. While the service is in state `HandlingRequest` the corresponding computations are performed; the service moves back to `Idle` as soon as the result is delivered back to the caller.

2.2. Components

Components provide implementations for services. As has been pointed out in the literature [14, 23] a component can provide implementations for multiple services. In addition, services can be implemented by networks of multiple cooperating components. In this case, the service's functionality is composed from the functionality of the individual components.

2.3. Resources

The term *resource* is used in the literature essentially to refer to everything in the system which is required by an application (in our case the "application" includes both components and the container) in order to provide its services (e.g., [9, 28]). More specifically, Goscinski defines a resource as:

> "...each reusable, relatively stable hardware or software component of a computer system that is useful to system users or their processes, and because of this [...] is requested, used and released by processes during their activity." ([9, Page 440f.])

The most important properties of a resource are that it can be allocated to, and used by, applications, and that each resource has a maximum capacity. We do not consider resources with unlimited availability, because they do not have any effect on the non-functional properties of an application. We distinguish between the actual resource (e.g., CPU, memory) and the aspect it enables (e.g., execution of program code/computation, availability of space to store data). It will become clear further down why we believe this distinction is important.

2.4. Container

The components implementing a service require a runtime environment to be executed. We call this runtime environment the *container*. The container instantiates components, connects these instances to other instances according to the functional service specification, and provides various middleware services to the components, including access to the underlying platform. In short, the container manages and uses the components such that it can provide the services clients require. Extending this notion to non-functional properties, we see that the container needs to use components and resources in such a way that it can guarantee the required non-functional properties of the services it provides.

Additionally, the system's environment also plays an important role. In particular, the container may need to make assumptions about the environment in order to provide its services. In this case, the container will only be able to provide a certain level of non-functional properties as long as its assumptions about the environment are still valid. Environment assumptions may include information on the interarrival times of requests (for time-based properties), assumptions about the abilities of system attackers (for security properties), usage profiles, etc.

3. Non-functional Specification

In this section we explain how we specify non-functional properties of component-based systems. There are two parts to a non-functional specification: measurements and constraints. We will explain how measurements can be used to define non-functional dimensions along which systems can be specified. Finally, we introduce the term of a feasible system which describes a system which has sufficient resources to provide a certain service with certain non-functional properties.

3.1. Measurements

We use the concept of a *measurement* to represent non-functional dimensions of systems. Non-functional specifications can then be expressed as constraints over measurements. Our concept of a measurement is based on the measurement theory one (e.g., [8]) where a measurement is a mapping from physical or empirical objects to formal objects. The "physical or empirical objects" in our case are states of the system, thus measurements can be represented as state functions. We have explained in other publications [20, 21] how we use *context models* to specify measurements independently of the concrete applications on which they are to be used.

We distinguish two kinds of measurements:

1. *Extrinsic measurements* describe a non-functional dimension which is applicable to a service and is relevant from a user perspective. They view the system as a whole and do not make distinctions to allow for other services, other components, or resource contention. In effect, extrinsic measurements can be used to describe users' non-functional requirements on a service.

   An example for an extrinsic measurement is response time of a service, a state machine with a def-

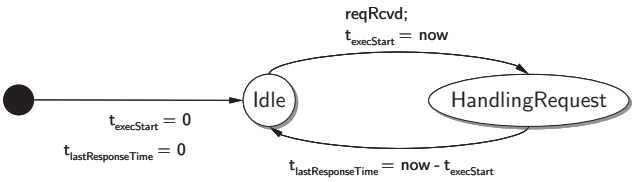

Figure 2. State machine with definitions for the response time measurement.

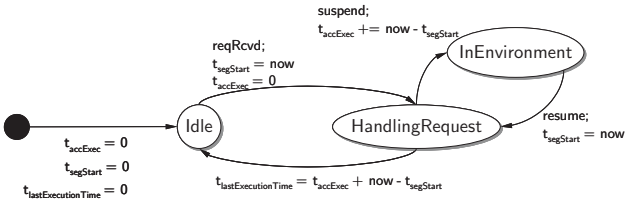

Figure 3. State machine with definitions for the execution time measurement.

inition can be seen in Fig. 2. The state machine has been annotated with assignments which are executed when the corresponding transition is triggered. In any state $t_{lastResponseTime}$ holds the response time of the last service invocation. The definitions use the special variable now, introduced by Abadi and Lamport in [3], which represents the current time and advances independently of the state machine shown. $t_{execStart}$ is a helper variable holding the start time of the last service invocation.

2. *Intrinsic measurements* describe non-functional dimensions of component implementations. The value of an intrinsic measurement for a specific implementation depends principally on the way the implementation is realised. If two implementations differ in their values for an intrinsic measurement, they use different algorithms or implementation techniques to provide their functions. Definitions of intrinsic measurements account for the presence of other components, and for resource contention, i.e., for the environment in which the component will be executed. In effect, intrinsic measurements can be used to describe the properties of an actually existing implementation independently of how this implementation is used.

   An example for an intrinsic measurement is execution time of an operation, a state machine with a definition can be seen in Fig. 3. This state machine is very similar to the one in Fig. 2, $t_{lastExecutionTime}$ holding the execution time of the last invocation of the operation. Note, however, the additional state `InEnvironment` which is used to model the fact that the component's execution may be interrupted by the environment in favour of another component (`suspend`) and later resumed again (`resume`). The assignments at the transitions are designed such as to ensure that execution time only counts the time actually spent executing the component and does not count the time spent executing other components. This is achieved using the helper variables $t_{segStart}$, the start time of the last execution segment, and $t_{accExec}$ the accumulated execution time so far.

The distinction between these two kinds of measurements can perhaps be most clearly described by the following example: For a component which provides an operation to add two integer values, we can determine the execution time of this operation by adding the time taken to load the two parameter values into the processor, to perform an `add` machine operation, and to store the result into the memory slot for the return parameter. However, there is no way to determine the response time this component will exhibit solely from the component's code. The reason for this is that the response time depends on how the component is *used* instead of how it is *implemented*. The execution time only determines a lower bound for the response time, but the container can use the component to provide any response time above this lower bound. For example, the container may buffer incoming requests and distribute them to multiple instances of the component in order to be able to service a large number of incoming requests. In this case the allocated size of the buffer generates an additional delay which must be added to the execution time to determine the response time. Other examples in support of this distinction include integrity of component code, and load balancing.

3.2. Non-functional Properties

Non-functional properties are constraints over measurements. Examples are properties like "The response time of service `login` is always less than 50 ms", or "The execution time of operation `login` is always less than 30 ms." (Note the difference between service and operation and how—accordingly—an extrinsic respectively an intrinsic measurement is used. The difference between these two specifications will be further explored below.) As usual, any non-functional property can be interpreted as a non-

functional specification, stating that the property holds for the element being specified.

We distinguish four kinds of non-functional specifications:

intrinsic specification specification of component implementation properties

extrinsic specification specification of properties of a service

resource specification specification of resource properties

container specification specification of container behaviour with respect to non-functional properties

We will now look at each of these specifications in turn.

3.2.1. Intrinsic Specifications Component implementation properties are described using constraints over intrinsic properties. These constraints describe relations between the various intrinsic measurements relevant for the component implementation. The most simple example is a statement like: "The execution time of operation `login` is less than 30 ms". More complex properties constrain the relation between multiple measurements. For example, for a component computing a numerical value (e.g., the adder component from Sect. 3.1) the execution time might depend on the number of exact decimals the component computes. Intrinsic specifications describe the effect of the algorithms and implementation techniques used to create a component implementation.

Note, that component implementation specifications make no explicit mention of the resources required to provide the component's services. It is not useful to express resource demand of a component as an intrinsic property, because it depends largely on how the component is used. For example, CPU demand depends on both the intrinsic property execution time, and the number of requests per second the component has to serve. For this reason, component implementation specifications constrain intrinsic properties only, but some of these intrinsic properties (e.g., execution time) correspond to an aspect enabled by a certain resource (e.g., CPU). The relation between resource specifications and component implementation specifications is established by the container specification described later in Sect. 3.2.4.

3.2.2. Extrinsic Specifications Service specifications essentially constrain extrinsic measurements for a single service. These constraints express how users expect the system to behave. In addition to such a simple classification into acceptable and unacceptable behaviours we also investigate using value functions (e.g., [16, 22]) to express users' preferences on acceptable behaviours as well as their preferences for certain services. However, in the context of this paper we constrain ourselves to simple constraints on extrinsic measurements. The property "The response time of service `login` is always less than 50 ms" from above is an example for an extrinsic specification.

3.2.3. Resource Specifications As we explained in Sect. 2.3, resources enable some non-functional aspect, provided their capacity is sufficient to serve the specified load. This leads directly to a resource specification schema which has two parts: a) an antecedent describing the capacity limits, and b) a consequence describing the non-functional aspect enabled by the resource. While the antecedent depends strongly on the specific resource and may use arbitrary parameters and formulæ, the consequence is in effect a constraint over intrinsic measurements, although some mapping needs to be provided in the container specification (Sect. 3.2.4). For example, for a CPU with a rate-monotonic scheduler [15] the capacity limit can be expressed by the following formula:

$$\sum_{i=1}^{n} \frac{t_i}{p_i} \leq n \cdot \left(\sqrt[n]{2} - 1 \right)$$

where n is the number of tasks, and t_i and p_i refer to the worst case execution time and period of the ith task, resp. The non-functional aspect enabled by this resource is that these n tasks described by these parameters can be scheduled to execute jobs with a period p_i which are allowed to execute for at least t_i units of time between the begin and end of their respective period. This is in essence a constraint over execution time.

As shown in the example above, resource specifications are the "hook" at which the well understood theories from operating systems research can be plugged into the semantic framework.

3.2.4. Container Specification The container uses resources and components to provide a service with certain non-functional properties. In order to reason about the extrinsic properties of a system based on the intrinsic properties of its components and the available resources we need to specify precisely how the container uses the components and resources. A container specification is written in rely–guarantee style [13] with the antecedent asserting that:

1. the available component implementations have the provided intrinsic properties. This pre-condition essentially enumerates the intrinsic properties the container takes into account.
2. the system's environment guarantees certain properties. Depending on the algorithms implemented by the container, the system environment will need to give different guarantees. A typical example of the kind of guarantees given by the system environment is the distribution of request interarrival times. This can be used

together with queuing theory based techniques to determine the optimal number of components and buffer size to achieve a required response time with components with a known execution time [4].

3. the available resources will enable the required non-functional aspects. What non-functional aspects are required depends on the intrinsic properties of the available components, the extrinsic property to be provided, the guarantees given by the system environment, and the algorithms implemented by the container. This antecedent is the central part of the container specification which describes the mapping from extrinsic and intrinsic non-functional properties of services and components to the lower-level concepts of resource specifications.

Provided these conditions hold, the container guarantees that it will deliver a certain service with specified extrinsic properties. The container specification thus forms a second "hook" at which results from performance analysis, security analysis, etc. can be plugged into the framework.

The concrete implementation of such a specification is out of the scope of this paper. However, we have previously elaborated on an example of mapping component execution times to service response times in [4] where it was called Container-Based Scheduling. In this example, the container computes the number of instances to pre-instantiate, as well as the size of a buffer to use for storing incoming requests, so that a certain service can be provided with the specified extrinsic properties. It uses the intrinsic properties of the component implementation and information on the distribution of the interarrival times of incoming requests to compute the results.

3.3. Feasible Systems

In the last sections we have described four types of specifications. It is important to realise that for any software project these specifications are written by different people, and at different times in the development cycle. Resource specifications are written by the people who have built the hardware or provided the operating system code managing and scheduling the resources. Container specifications are provided by the people who have built the container. Intrinsic specifications are written by component developers, while extrinsic specifications are laid down by application designers.

All these specifications are only useful if we can compose them to obtain a global view of the system which we can use for analysis. One useful analysis is to test whether the available resources are sufficient to provide the required extrinsic properties given the available components and the container specification. This is equivalent to proving that the composition of resource specifications, container specification, intrinsic specifications, and system environment guarantees implies the extrinsic specification. This can be stated as

$$(R \wedge C \wedge I \wedge E) \Rightarrow S$$

where

R: conjunction of the resource specifications of all resources available in the system

C: container specification

I: conjunction of the intrinsic property specifications of all available component implementations

E: environment guarantees, if any are needed by the container specification

S: required extrinsic property specification

We define a *feasible system* to be a system made up from components, resources, and a container which together fulfil the above condition under given environment assumptions. We can use Abadi/Lamport's rule of inference [2] to prove feasibility, provided all the antecedents of all specifications are safety properties. A complete example of such a proof cannot be given here due to space restrictions.

4. Related Work

In his thesis [1] Aagedal defines CQML, a specification language for non-functional properties of component-based systems. The definition remains largely at the syntactic level, semantic concepts are mainly explained in plain English without formal foundations. Staehli [25] describes a formal technique for specifying non-functional properties of multimedia presentations. As an extension and combination of these efforts, the two authors recently and independently of our research published a short paper on "QoS Semantics for Component-Based Systems" [24]. Their work is restricted to timeliness and data quality properties and does not cover resource demand at all. In contrast, we use more abstract definitions which cover any kind of measurement, including but not limited to timeliness and data quality. Also, resource demand and resource allocation is a central element of our semantic domain.

Hissam et al. [11] describe a prediction-enabled component technology (PECT). This work is very similar to our work in that it attempts to provide a framework in which specific analysis methods and specific component models can be combined. However, their work is somewhat more abstract. Also, they seem to be exclusively concerned with modularisation into components, whereas our work explicitly takes into account the container and resources as an important yet separate part of an overall system. Bertolino, Mirandola and Vincenzo [5, 10] presented work attempting to merge techniques from software performance engineering

with component-based software engineering. They distinguish two model layers: the software model which represents the logical component structure of a system, and the machinery model which models properties relevant for performance analysis. Their work is based on the UML profile for schedulability, performance and time specification [17]. Reussner et al. [19] describe work on analysing non-functional properties of components and component-based systems using parametrised contracts [18]. They provide arguments which support our claim that properties of components and the effects of using components must be treated separately. Specifically, they use Markov-chains to model and analyse reliability of component-based systems. Their work only considers properties on the component level and does not capture influences from the runtime environment and the underlying resources.

There are various publications proposing the use of value functions for the specification of non-functional properties [16, 22], using a task-based computational model. Resources are considered in [16], however only where resource demand can be adjusted during execution. The model is completely oriented towards adaptation, admission control is not considered in this model. In contrast, we defined the notion of a *feasible system* which captures admission control. The authors of [22] combine the concepts of measurement and constraints over measurements into the notion of a "metric".

A large host of literature dealing with non-functional properties (especially real-time properties) exists from the areas of operating systems research and performance analysis and prediction (cf. e.g., [15, 26, 28, 29]). Our approach aims to integrate these approaches with ideas from component-based software engineering to enable software developers to consider non-functional properties of complex applications as early as possible in the development cycle.

5. Conclusions and Outlook

In this paper we have presented semantic concepts which form the basis of a semantic framework for the specification of non-functional properties of component-based software. We have presented a system model in which components with intrinsic properties (depending on the algorithms and techniques used in their implementation) are used by a container to provide a specified service with specified extrinsic non-functional properties using the available resources of the system. Measurements are used to express non-functional dimensions, and non-functional properties (or specifications) are expressed as constraints over such measurements. We distinguish two types of measurements: intrinsic and extrinsic measurements. We have shown what is expressed in specifications of the individual parts of the system, and how these parts work together to form a system. Finally, we have introduced the term *feasible system* describing a system in which sufficient resources, and the necessary component-based scheduling algorithms in the container are available to provide a service with the required non-functional properties. We are currently working on formally describing the concepts presented in this paper as well as developing concrete component-based scheduling algorithms, such as the one described in [4]. We have shown that the individual specifications can be developed independently and indeed by different people and at different moments in time. This enables us to distribute the specification and development work in a development team, which allows us to handle much more complex problems than without this modularisation. Because the specifications can be composed to provide a global view of the system, we can still ensure the system will meet its requirements.

Furthermore, we plan to extend our approach in various ways:

- We want to include stream-based applications into our concept of a *service*. Stream-based communication is an important concept, especially in multimedia applications, which frequently have real-time and other non-functional requirements. So far, however, our model only supports request–response scenarios, where a service essentially represents a single operation.
- Because different services may share the same resources, there can be dependencies between their non-functional properties. These dependencies can lead to conflicts, so that not all of the services can be performed with their specified extrinsic properties. In order to resolve such conflicts, we need another specification indicating users' preferences on services. Similarly, some extrinsic properties of the same service may interact. For this case, we also need a specification of users' preferences to resolve conflicts. We are going to use value functions, and to provide more elaborate container specifications to support this.
- We have not shown in this paper, how intrinsic properties of different components can be composed to derive intrinsic properties of a component network. This can essentially be done by simply composing their specifications, but we need to perform further research to support this claim.
- Non-functional properties often depend on the actual data being processed. We plan to extend our approach to cover this aspect for special cases where the resulting non-functional property can be combined from a specification of the component and a specification of the data.

Acknowledgements

This research is carried out in the context of the COMQUAD project, a project funded by the German Research Council (DFG, project no. FOR 428). See www.comquad.org for more details. I am grateful to Prof. Heinrich Hussmann, Sten Löcher, Simone Röttger, and everybody in the COMQUAD project for many fruitful discussions. I also want to thank Ralf Reussner and the anonymous reviewers for providing very helpful comments towards the final version of the paper.

References

[1] J. Ø. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[2] M. Abadi and L. Lamport. Composing specifications. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 1–41, Berlin, Germany, 1989. Springer-Verlag.

[3] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM ToPLaS*, 16(5):1543–1571, Sept. 1994.

[4] R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. In *Proc. Int'l Conf. on Software & Systems Engineering and their Applications (ICSSEA)*, Paris, Dec. 2003.

[5] A. Bertolino and R. Mirandola. Towards component based software performance engineering. In *Proc. 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction at ICSE 2003*, pages 1–6. ACM/IEEE, May 2003.

[6] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison Wesley Longman, Inc., 2001.

[7] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. The Kluwer international series in software engineering. Kluwer Academic Publishers Group, Dordrecht, Netherlands, 1999.

[8] G. Ford. Measurement theory for software engineers. In *Lecture Notes on Engineering Measurement for Software Engineers*. Carnegie Mellon University, 1993. CMU/SEI report CMU/SEI-93-EM-9.

[9] A. Gościński. *Distributed Operating Systems: The logical design*. Addison-Wesley Publishers Ltd., 1991.

[10] V. Grassi and R. Mirandola. Towards automatic compositional performance analysis of component-based systems. In *Proc. 4th Int'l Workshop on Software and Performance WOSP 2004*, pages 59–63, California, USA, Jan. 2004.

[11] S. A. Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau. Packaging predictable assembly. In J. Bishop, editor, *Proc. IFIP/ACM Working Conf. on Component Deployment (CD 2002)*, volume 2370 of *LNCS*, pages 108–126, Berlin, Germany, June 2002. Springer-Verlag.

[12] IASTED. *Proc. IASTED Int'l Conf. on Software Engineering (IASTED SE'04)*, Innsbruck, Austria, Feb. 2004. ACTA Press.

[13] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Manson, editor, *Proceedings of IFIP '83*, pages 321–332. IFIP, North-Holland, 1983.

[14] I. H. Krüger. Service specification with MSCs and roles. In *Proc. IASTED Int'l Conf. on Software Engineering (IASTED SE'04)* [12].

[15] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, NJ, 2000.

[16] J. W. S. Liu, K. Nahrstedt, D. Hull, S. Chen, and B. Li. EPIQ QoS characterization. ARPA Report, Quorum Meeting, July 1997.

[17] Object Management Group. UML profile for schedulability, performance, and time specification. OMG Document, Mar. 2002. URL http:// www.omg.org/ cgi-bin/ doc?ptc/02-03-02.

[18] R. H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[19] R. H. Reussner, I. H. Poernomo, and H. W. Schmidt. Contracts and quality attributes for software components. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proc. 8th Int'l Workshop on Component-Oriented Programming (WCOP'03)*, June 2003.

[20] S. Röttger and S. Zschaler. Model-driven development for non-functional properties: Refinement through model transformation. In *Proc. <<UML>> Conf.*, 2004. To appear.

[21] S. Röttger and S. Zschaler. A software development process supporting non-functional properties. In *Proc. IASTED Int'l Conf. on Software Engineering (IASTED SE'04)* [12].

[22] B. Sabata, S. Chatterjee, M. Davis, J. J. Sydir, and T. F. Lawrence. Taxonomy for QoS specifications. In *Proc. 3rd Int'l Workshop on Object-oriented Real-Time Dependable Systems (WORDS'97)*, Newport Beach, California, Feb. 1997.

[23] C. Salzmann and B. Schätz. Service-based software specification. In *Proc. Int'l Workshop on Test and Analysis of Component Based Systems (TACOS) ETAPS 2003*, Electronic Notes in Theoretical Computer Science, Warsaw, Poland, Apr. 2003. Elsevier.

[24] R. Staehli, F. Eliassen, J. Ø. Aagedal, and G. Blair. Quality of service semantics for component-based systems. In *Middleware 2003 Companion, 2nd Int'l Workshop on Reflective and Adaptive Middleware Systems*, 2003.

[25] R. Staehli, J. Walpole, and D. Maier. Quality of service specification for multimedia presentations. *Multimedia Systems*, 3(5/6), Nov. 1995.

[26] J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[27] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, Nov. 1997.

[28] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2002.

[29] A. M. Tilborg, editor. *Foundations of Real-Time Computing*. Kluwer Academics, 1991.