Towards Pervasive Treatment of Non-Functional Properties at Design and Run-Time

Ronald Aigner, Martin Pohlack, Simone Röttger, Steffen Zschaler
Dresden University of Technology
Department of Computer Science
Institute of System Architecture
Operating Systems Chair
01062 Dresden

phone: +49 – 351 – 463 3 8321
fax: +49 – 351 – 463 3 8284

e-mail: {aigner, pohlack, roettger, zschaler}@inf.tu-dresden.de

Abstract

Disregarding non-functional properties is an important project risk. They have to be taken into consideration throughout the system's life cycle. Particularly, they must be considered as early as possible in the design phase. Especially in the area of component-based software development it is also important to be able to separate functional and non-functional requirements. Pervasive treatment of non-functional properties allows reusing components in previously uncovered environments.

In this paper we present an approach that aims at treating non-functional aspects of a system all the way from design time models to runtime enforcement and Quality of Service (QoS) guarantees. In particular, this includes transformation of non-functional property specifications from a human-readable in a machine-readable form, scheduling of component usage, and resource reservation.

Keywords: Component-Based Software, Quality of Service, Software Development Process, Resource Management, Container-Based Scheduling

1 Introduction

We aim at the development of component-based software [13] with non-functional requirements – and in particular Quality of Service (QoS) requirements. These QoS requirements play an increasingly important role in software development. They constitute an important project risk and therefore need to be taken into account as early as possible in the development process. This includes specifying, and analyzing non-functional properties of models of the system under development, but also using this information further down in the process. Figure 1 gives an overview of the process, which is being developed as part of the COMQUAD[1]-project.

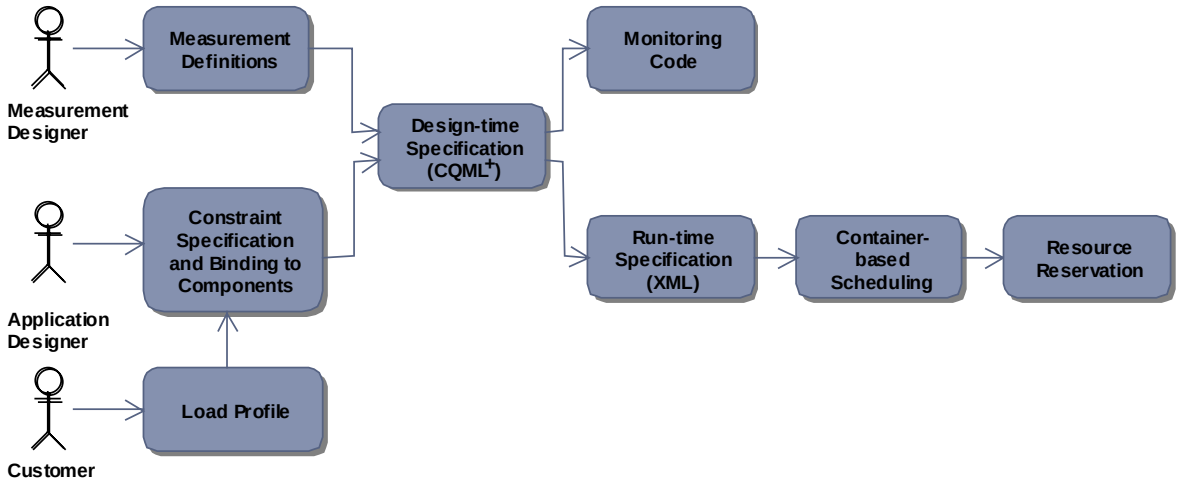

Figure 1 Process overview

Our process is based on the specification language CQML$^+$ [11], an extension to CQML [1], as well as UML, the unified modeling language [10]. The language allows separation of measurement definition and specification of non-functional properties of applications using these measurements. Measurement definitions can be very complex, but on the other hand will be developed only once. Therefore, we separate the roles of measurement designer and application designer in our process. Their combined efforts lead to a complete specification of the system including its non-functional properties.

This specification can be used for a variety of purposes. Besides generating code for runtime monitoring of QoS parameters, its main use is in providing a base for scheduling and resource reservation in the running system. To this end, the specification is first transformed from a human understandable form into a more compact, XML-based form that has been optimized for machine usage. The runtime environment – the component container (or just container) – uses the information in this specification to schedule component usage. This has been named Container-Based Scheduling in Figure 1. Based on the schedule the container can then reserve resources on behalf of each component.

The resource management is based on DROPS – the Dresden Real-time OPerating System [2]. The DROPS resource management consists of hierarchical *resource managers* [7].

Our approach provides pervasive treatment of non-functional properties from design time to run-time. Other projects, such as VEST [12] or Globus [3, 4, 5] are only concerned with one of these steps. TAO [6], another project concerned with non-functional properties for component-based software, includes a flexible scheduling framework that allows applications to define their own scheduling strategies for their tasks. In contrast, our container based scheduling approach delegates CPU scheduling, and other lower level scheduling, to the appropriate resource manager and constrains itself to decisions about the number of component instances and the buffer size. It can therefore be considered scheduling at the component level, while TAO provides hooks for scheduling at the resource level.

In the remainder of this paper the individual steps of this process are examined in some more detail. The focus is on the transformation from CQML$^+$ to the XML-based runtime representation, the container-based scheduling, and the resource reservation.

[1] COMQUAD – COMponents with QUantitative properties and ADaptation is a research project at Dresden University of Technology and Friedrich-Alexander-University Erlangen-Nuremberg, funded by German Research Council.

2 Design Phase

Two different roles are involved in the analysis and design process of non-functional property specifications. An expert in the analysis and specification of non-functional properties holds the first role. He defines measures, which are entities needed in the specification, together with their semantics. The second one – the application designer – uses these measures in an application modeling tool to specify required system qualities through constraints on measures. He analyses the load profile provided by the customer in order to design the application. The load profile consists of

- QoS requirements per request,
- a specification of the interarrival times of requests, and
- the percentage of requests for which the required QoS must be fulfilled.

These two different views on the system can be modeled using different UML-profiles representing the information adequate for each role. We will not elaborate on the aspect of separating the tasks in this paper. Instead, we focus on the specification languages of non-functional properties and their transformation rules.

At design time we specify non-functional properties using the language CQML$^+$ [11], based on CQML [1]. The basic constructs of the language are represented in Figure 2. CQML$^+$ builds on quality characteristics, which correspond to the measures provided by the measurement designer (Figure 1). Quality characteristics have a name, a domain, and a semantic, given by the values clause – an expression in Object Constraint Language (OCL [14]) which specifies how values of the characteristic can be determined in a system. Examples for such characteristics are delay or screen resolution. In a next step quality statements are used to specify constraints on characteristics. Both quality characteristics and quality statements are parameterized and can therefore be reused in different contexts. To actually associate the non-functional property specifications with the functional one, CQML$^+$ provides the construct of quality profiles. In the profile specification the formal parameters are replaced by actual parameters, for example operations or streams of the component to which the non-functional constraint is applied. Quality statements can be associated to a component as offers (`provides`), requirements (`uses`), or resource demand (`resources`).

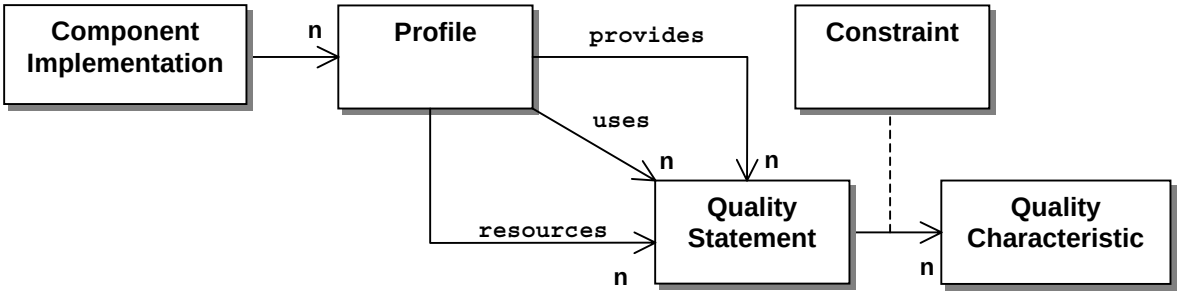

Figure 2 Structure of CQML+ represented as a UML class diagram

3 Transformation to Machine-Usable Format

Once the application designer has developed the final non-functional property specification, it can be deployed on the target system together with the application code. Because the requirements differ between a specification made by and for human beings and one made purely for machine use, it is useful to transform the specification to a separate language, designed for machine analysis. The language for design time specification (here CQML$^+$) is heavily structured to achieve reusability, readability, and understandability. In contrast the language for machine use has to be effective, processable, and has to concentrate on aspects needed by the target system.

Because of the availability of good programming libraries we decided to use XML for the runtime representation. The syntax is described using an XML Schema.

The degree of transformation should be somewhere between these two extremes:

1. The XML Schema maps all the concepts from CQML$^+$. The transformation does little more than check syntactic correctness and type conformance of the CQML$^+$ specification.
2. The XML file has been completely flattened, taking out any redundant or not strictly necessary information. This includes for example removing quality statements and characteristics and inserting constraints over the terms from the values clauses into the profile directly.

Clearly, the first option does not address the issue of differing requirements at all. However, also the second extreme is not entirely practical: When the runtime environment negotiates contracts, in many cases it will be sufficient to do comparison down to the level of quality characteristics only. Checking values clauses for equality will not be necessary. The assumption is that two characteristics of the same name represent the same thing and that this holds vice versa, too. As long as this assumption holds – and this seems reasonable for normal system development – there should be no need to consider comparing values clauses for contract negotiation. This is good, because values clauses can be any valid OCL expression, making comparison hard to do[2]. If the QoS specification is given according to the second option, we loose the possibility to compare at the quality characteristic level.

The best solution lies somewhere between the two extremes. It includes

- Replacing user defined names by globally unique alphanumeric identifiers: This allows us to get rid of namespaces, nested scopes and name overloading.
- Rephrasing of constraints in conjunctive normal form: This allows for more efficient traversal during contract negotiation.
- Removal of quality statements: These are essentially a means of reuse, giving a name to some constraint over quality characteristics. They can be removed by incorporating the constraint directly into the profile. However, the characteristics themselves should not be removed for the reasons mentioned above.
- Removal of unused parts from the specification: This is especially important when using libraries of QoS specifications, as they will normally contain much more definitions than will be used in a specification.

With all these changes to the structure of the specification, another problem comes up: traceability. In order to be able to understand problems in the finished system it needs to be possible to trace back from the XML-descriptor to the original CQML$^+$-specification. In order to support this, a log file for the generation of the XML-descriptor will be created. This log file stores information on the modifications performed.

4 Container-Based Scheduling

Given a component's QoS specification (i.e., the information from the XML representation of the CQML$^+$-specification) and a description of the load requirements it is the task of the container to schedule component usage in such a way as to ensure that QoS requirements are met. In the context of this paper we restrict ourselves to timeliness requirements, namely response times for operation calls. In this case there are two things the container can influence in order to meet the timeliness requirement (see Figure 3):

- the size of the request buffer and
- the number of component instances for serving the requests.

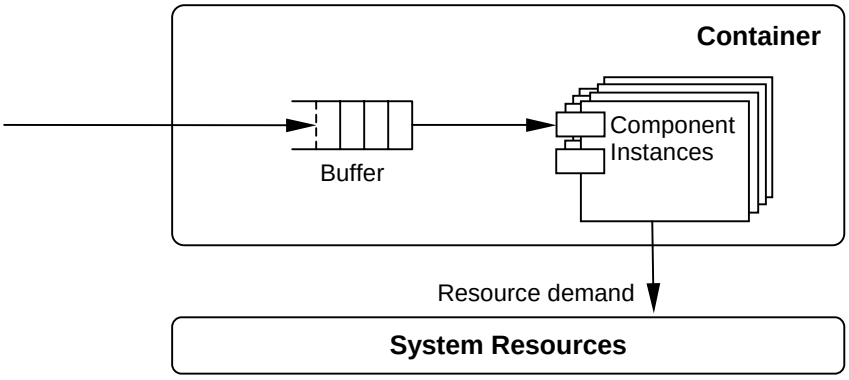

Figure 3 Schematic view of Container-Managed Scheduling

The buffer enables the system to accept incoming requests asynchronously and to restrict the number of component instances. However, the buffer adds delay to the overall processing time. An upper bound for buffer size is given by the difference between the requested response time and the response time the component offers. The lower bound of the buffer size determines the quality of the service, thus, the number of requests not dropped.

[2] Note that because OCL has finite quantification only, comparison is decidable.

To be able to assure a mathematically founded contract about the quality, we use an advanced methodology of the queuing theory – Jitter-Constrained Streams (JCS) [8]. A JCS consists of two tuples describing both, the interarrival times and the request sizes. The tuple describing the interarrival times consists of T, D, τ, and t_0 with

- $T > 0$ average event distance,
- $0 \leq D \leq T$ minimum distance,
- $\tau \geq 0$ maximum lateness, and
- $t_0 \in R$ starting time.

The tuple describing the request sizes comprises the parameters S, M, σ, and s_0 with

- $S > 0$ average request size,
- $0 \leq M \leq S$ minimum request size,
- $\sigma \geq 0$ maximum deviation from accumulated request size,
- $s_0 \in Z$ initial value.

For both, the source of requests and the consumer of requests (serving component), we use one JCS. Additionally, we restrict our model to use equal request sizes for input to and output from the buffer to be able to determine the quality.

To illustrate the usage of JCS we provide a short example. In this example, users may generate requests, which are described by a JCS J_{in}, which has interarrival times $\{T = 5s, D = 2s, \tau = 12s, t_0 = 0\}$ and request sizes $\{S = 1000$ Bytes, $M = 300$ Bytes, $\sigma = 4500$ Bytes, $s_0 = 0\}$. t_0 is set to zero for the sake of simplicity and s_0 is set to zero because we do not pre-buffer requests. The consuming component is described with the JCS J_{out}. The tuples describing the request sizes have equal parameters for both, J_{in} and J_{out}, as the consumer always dequeues entire requests. The consuming component is described by $\{T = 15s, D = 10s, \tau = 10s, t_0 = 0\}$.

Looking at the average interarrival times from J_{in} and the average processing time from the component described by J_{out}, we see that the container has to create at least three instances. Using the formulas from [8] we calculated a minimal buffer size B_{min} of 13,400 Bytes.

The container determines a concrete buffer size by balancing the demands and the delivered qualities of concurring services. The container may also determine the number of component instances it should try to use. The container can admit the load, if it can reserve the resources needed for the buffer and the component instances. However, creation of additional instances is only useful, if the resource utilization is increased (for instance, using four instances of a component concurrently on a four processor machine). Thus, the total amount of available resources defines an upper bound for the number of instances that can be created.

5 Resource Management

To be able to guarantee resource reservation at component level, the container has to run in an environment, which assures resource availability, such as DROPS.

The DROPS resource management consists of the *resource managers* as described in [7], each of which manages a specific resource, and the *QoS manager* [9], which is the central instance to negotiate with the resource managers a contract on behalf of a client. The client can be the container, which in turn acts on behalf of a specific component.

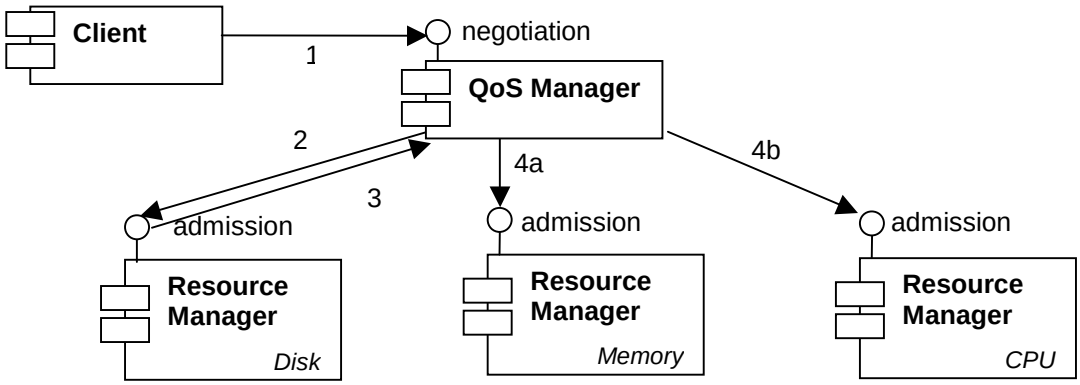

Figure 4 Resource Management: Reservation Negotiation

Figure 4 gives an overview of the resource reservation process. First, the QoS manager receives a request for the resource in a generic format (Step 1). The request consists of two parts – a generic part and a resource specific

part. The generic part is interpreted by the QoS manager and is used to identify the requested resource and its corresponding manager. The second part specifies the requested resource properties. This part is handed unmodified to the resource manager.

The container uses the CQML$^+$-specification to formulate a resource reservation request. The generic part – namely the name of the resource – is derived from the identifier of the resource as defined in the CQML$^+$-specification. The resource specific part is a list of attribute-value pairs derived from the quality statements related to this resource.

The resource managers implement an *admission interface*, which is used by the QoS manager to negotiate the reservation for a specific client. The QoS manager itself implements the *negotiation interface*, which is used by its clients. These interfaces are resource independent as to be able to add resources and their corresponding managers transparently. The interfaces take an XML input parameter that describes the request and return an XML output parameter containing a reference to the reserved resources. By using these parameters, the interfaces are simple and can be implemented by all resource managers. The resource managers in turn can interpret the input parameter in a way transparent to the QoS manager.

The QoS manager checks the generic part of the request for the requested resource. It uses the specified resource name to find all resource managers, which provide this resource type[3]. The QoS manager then sends the resource specific part of the request to the resource managers together with the identifier of the client that initiated the request (Step 2). The resource manager checks if it can serve the client with the requested quantity of the resource. The QoS-manager checks each of the matching resource managers and builds a reservation tree. If a resource needs other resources to fulfill the requested service it returns a request describing these (Step 3), which is negotiated by the QoS manager with the respective resource managers (Step 4a and 4b).

If a resource manager was able to make a reservation for a client, it will generate a handle, containing an identifier, which can be used to address the resource, the identifier of the client, and an internal identifier to address the specific reservation. This handle is returned to the QoS manager. After the QoS manager collected all handles for the resources specified in the request, it hands them to the client. Upon resource usage, a client has to provide the resource handles to the resource managers or the instances of a resource to verify the reservation.

In case there is a change in the availability of the quantities of a resource, for instance varying bandwidth of a mobile device, the resource manager of the corresponding resource has to inform its clients about the change. It does so by sending a message to the *notification interface* of its affected clients. Each container has to provide an implementation of this interface. The message contains the handle to the reservation that can no longer be satisfied. A container can react to the message in several ways: it may ignore the message; it may adapt the component to adjust to the new situation; or it may renegotiate the component's resource allocations.

6 Conclusion

We presented an approach to treat non-functional properties, such as Quality of Service, throughout the whole development process, including the design time models, implementation, and run-time enforcement. We showed how to specify non-functional properties for a component, how to transform such a specification into a machine-readable format, how to schedule component instances depending on these properties, and we gave an overview about a run-time resource management system, using these properties to allocate requested resources.

We currently investigate scheduling mechanisms for the container, using an adaptation of the theory of Jitter Constrained Stream with the restriction of equal request sizes, as described in Section 4. We plan to extend the theory to utilize arbitrary resource-usage distributions. Another focus of our research is the full tool support for the described development steps. Currently we have tool support to transform a CQML$^+$ description into an XML Schema. Other tools are needed to perform design time validation of non-functional properties. Also, graphical support for UML-based modeling is under development.

References

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

[2] R. Baumgartl, M. Borriss, H. Härtig, Cl.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, J. Wolter: *Dresden Realtime Operating System*; In the proceedings of the Workshop of System-Designed Automation (SDA'98), March 1998, Dresden, Germany

[3] Memory, for example, can be provided by more than one memory manager. Each memory manager may provide different properties, such as non-paged memory or persistent memory.

[3] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy: *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation*; Intl Workshop on Quality of Service, 1999

[4] I. Foster, A. Roy, V. Sander: *A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation*; 8th International Workshop on Quality of Service, 2000

[5] I. Foster, M. Fidler, A. Roy, V, Sander, L. Winkler: *End-to-End Quality of Service for High-end Applications*; Computer Communications, Special Issue on Network Support for Grid Computing, 2002

[6] Christopher D. Gill, David L. Levine, Douglas C. Schmidt. *The Design and Performance of a Real –time CORBA Scheduling Service*. In: Real-Time Systems, Kluwer Academic Publishers, March 2001.

[7] H. Härtig, L. Reuther, J. Wolter, M. Borriss, T. Paul: *Cooperating Resource Managers*; In Proceedings of Workshop on QoS Support for Real-Time Internet Applications, Vancouver, Canada, June 1999

[8] Cl.-J. Hamann, A. Märcz, K. Meyer-Wegener: *Buffer optimization in real-time media streams using jitter-constrained periodic streams*; TU Dresden technical report, January 2001

[9] Jörg Nothnagel: *Ressourcenverwaltung in DROPS*; Diploma, Dresden, Germany, July 2002

[10] Object Management Group. *Unified Modelling Language Specification Version 1.4*. Document, September 2001.

[11] Simone Röttger, Steffen Zschaler: *CQML$^+$: Enhancements to CQML*. Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering; Cepadues Editions, Toulouse, France, 2003.

[12] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis: *VEST: An Aspect-Based Composition Tool for Real-Time Systems*; IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), Washington DC, May 2003

[13] C. Szyperski: *Component Software –Beyond Object-Oriented Programming*; Addison-Wesley, Harlow, England, 1997/98.

[14] Jos Warmer; Anneke Kleppe: *The object constraint language: Precise modeling with UML*. Reading, Massachusetts: Addison Wesley Longman, Inc., 1999