Explicit Modelling of QoS-Dependencies

Steffen Zschaler[1] and Marcus Meyerhöfer[2]

[1] Dresden University of Technology
Steffen.Zschaler@inf.tu-dresden.de
[2] Friedrich–Alexander–University Erlangen–Nürnberg
Marcus.Meyerhoefer@informatik.uni-erlangen.de

Abstract. When specifying Quality of Service (QoS) for components the usual method employed is to specify regions of QoS offers and requirements. The component offers a certain QoS under the condition that it will be provided some specified QoS by the environment. The dependency between offered and used QoS is essentially given through discrete examples, rather than an equation or inequity. In the COMQUAD project[3] we have come to the conclusion that in some cases it is both more natural and more efficient to specify these dependencies more explicitly, and precisely.
In this position paper we discuss the advantages and disadvantages of both approaches and give some first ideas as to how QoS–dependencies can be specified explicitly.

1 Introduction

The idea to develop software from already existing and well-understood parts is not a new one. After it had become clear that object-oriented software development was just another step towards a new way of software development – amongst other things because of the tight connection between the objects and the application they were developed for – it is commonly understood that component-based technologies facilitate software reuse and ease application development. This is achieved through the concept of software components – units of software that have no implicit dependencies among each other and therefore can be subject to independent reuse.
While there are a lot of different definitions what a component is, there is none that is commonly agreed upon. We follow the definition of Szyperski in [7] stating: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."
As in engineering sciences – like for example electrical engineering, where hardware is build out of predeveloped IC-cores or whole microchips –

[3] COMponents with QUantitative properties and ADaptivity is a project funded by the German Research Council. It was started October 1, 2001, at Dresden University of Technology and Friedrich–Alexander–University Erlangen–Nürnberg.

rigorous specification is the key factor for the success of such an approach, because developers need to be able to understand the function, usage and semantics of a component without knowing its source code.
From this perspective building software out of components deals with specifying contracts between it's parts. Commonly there are four different levels of contracts distinguished [2]. In the COMQUAD project, a project aimed at the development of a system architecture and development methodology for component-based software with quantitative properties and adaptivity, we focus on the fourth level of contracts, i.e. the specification of Quality of Service (QoS) and security-related properties of software. This will enable us to estimate properties of applications built from components and react to changing environmental conditions through adaptation. However, adaptation is outside the scope of this paper and not further considered herein (see for example [6]).
This position paper explores how to specify dependencies between the QoS offered and used by a component, showing that current approaches are not expressive enough to capture all dependencies. Examples are given in the specification language CQML [1]. Section 2 introduces the area of concern on the basis of a simple application and draws attention to the problems discovered. In Section 3 we present some possible solutions and open questions. In Section 4 we summarize the issues covered and outline the path we want to follow up in the future.

2 Problem Description

In QoS specification languages for components (e.g., CQML) the space of QoS characteristics is usually divided up into regions, that is intervals for measured quantities. These regions are defined in an assumption-guarantee ([3]) style through constraints on QoS characteristics, specifying what a component expects from the environment (including other components) and what it will offer to the environment. At any one moment in time, the component being executed "lives" in exactly one of these regions.
If the QoS state of the environment changes, moving the component out of its current region, the runtime system attempts to find a different region which fits the new situation. If it finds such a region, it performs a transition between the two regions, i.e. it informs the component that it now works in a different region and thus allows it to make any adjustments necessary.
For example, consider a web server which besides normal web-pages serves dynamic content, such as video streams. Depending on client hardware and system capabilities the video information can be sent in different levels of compression. The compression takes place on-the-fly in the web server. It is handled by an encoder component.
An encoder component performs a difference encoding by comparing two adjacent images in a stream of images and extracting the difference between those two images.
This component could be specified in CORBA IDL as follows:

```
interface ImageSource {
  Image getNextImage();
}

component ImageStreamEncoder {
  provides ImageSource encodedImages;
  uses ImageSource unencodedImages;
}
```

It is immediately clear that the time that passes between issuing a request for the next encoded image and receiving the result is at least as much as the time it takes to fetch the source image. Lets assume, we even know that the actual encoding takes another 5 milliseconds. To express this in CQML we first have to specify the characteristic `response_time` of an operation:

```
quality_characteristic response_time (op: Operation) {
  domain: numeric real [0..) milliseconds;

  values: (op.SE->last().time() -
           op.SR->last().time()).abs();
}
```

Where `op.SE` and `op.SR` are queues of events registering request and response, resp. The values clause itself describes the semantics of the given `quality_characteristic` in terms of the underlying computational model. Now we have to define regions of QoS for the component. For example:

```
quality low_response_time (op: Operation) {
  response_time (op) < 5;
}

quality med_response_time (op: Operation) {
  response_time (op) >= 5 and
  response_time (op) < 10;
}

quality high_response_time (op: Operation) {
  response_time (op) >= 10 and
  response_time (op) < 20;
}

profile response_times for ImageStreamEncoder {
  profile fast {
    uses low_response_time (unencodedImages.getNextImage);
```

```
    provides med_response_time (encodedImages.getNextImage);
  }

  profile slow {
    uses med_response_time (unencodedImages.getNextImage);
    provides high_response_time (encodedImages.getNextImage);
  }
}
```

Immediately, two questions arise:

1. What happens if fetching a source image takes 10 milliseconds or longer?
2. Is there any specific reason, why we chose 5 and 10 milliseconds as the regions' boundaries?

These two issues are actually two facets of one basic problem: The structure of the language forced us to represent an explicit dependency (i.e., computing the encoded image takes 5 milliseconds plus the time needed to fetch the unencoded image) in implicit form, by way of giving a finite number of examples. The only reason why we chose the numbers 5 and 10 as boundaries of the regions is that we had to choose some boundary value and 5 and 10 were just as good as any. Because we can only give a finite number of regions (i.e., examples), there will always be a region that we cannot specify.
The example shows that, although the specification of QoS dependencies using regions is useful, there are cases where it would be desirable to be able to specify the dependency more explicitly. Which type of specification is more useful depends on the context of the specification:

1. As can be seen from the example above, there are some cases where the explicit description is the more natural form. This is for example always the case when we already know an explicit dependency which we only want to formalize.
2. Specifying dependencies explicitly can provide the runtime system with more information. This may make adaptation to changing resource situations easier, because the runtime system can predict the effect of changes on a component's offered QoS.
   Adaptation is of course also possible if the system has "only" been specified using regions, but in this case the runtime system is restricted to adapt between the predefined regions. If no region has been defined for the current situation, the system fails. Note that providing "closure" regions just to make sure every possible situation has been covered in some way is not always feasible due to combinatorial explosion. If the different regions are classified by `uses` clauses constraining only one QoS characteristic, things are simple enough and one "closure" region will be sufficient. If the `uses` clauses constrain more than one different characteristics, there needs to be one "closure" region for each combination of characteristics.
   For example, if we have the characteristics `a` and `b` and the statements

a1 The characteristic `a` is in an undefined region.
a2 The characteristic `a` is in a well-defined region.
b1 The characteristic `b` is in an undefined region.
b2 The characteristic `b` is in a well-defined region.

then we need to provide "closure" regions for the following combinations: `a1 and b1`, `a1 and b2`, and `a2 and b1`. The number of combinations obviously explodes with the number of different characteristics.

3. Assumption-guarantee-style specifications [3] are more naturally expressed with `uses` and `provides` clauses using regions of offered and expected quantities.
   The semantics of QoS specifications with explicit dependencies is quite different when it comes to negotiating QoS contracts. Normal assumption-guarantee specifications can to a large extent be negotiated locally between two components at a time. If only explicit dependencies are present, contract negotiation needs to always "ripple through" all the way to the fundamental resources, because only there actual values of offered QoS exist, against which the required QoS (computed from the primary required QoS and the QoS dependencies specified) can be compared. Thus, the actual contract negotiation may be more efficient for assumption-guarantee-style QoS specifications.
4. In some cases the system model may not be sufficiently detailed to discern a clear dependency between offered and used QoS. Then, specifying regions will remain our best choice. One such case is when the offered QoS depends not only on the QoS provided by other components, but also on the actual data being handled. This can presently not be expressed easily in a CQML specification, so the best we can do is to provide a range of offered QoS which encompasses all values possible for any piece of data potentially to be processed.

Another reason for choosing one way of specifying over the other lies in the development approach used. The two specification methods differ in their appropriateness for top-down or bottom-up approaches, respectively.

For a bottom-up approach, in which components are developed independently and later assembled to form various applications, we would specify QoS dependencies explicitly. The reason is that in this case we know the components and can understand their behaviour by measuring it in a testbed. Thus, we are able to learn as much as possible about the component and we do not have to restrict ourselves to regions. Thinking further along these lines, we find that explicit dependency specification could be a good way of *documenting* QoS related issues for components off the shelf (COTS).

On the other hand, with a top-down approach, where we design an application and, in the process, break it up into components, using regions for QoS specification may be more appropriate. Here we cannot measure the component behaviour prior to the application design. Part of the requirements is a document specifying what non-functional properties are expected to hold with respect to the system. This is essentially, what

the system is supposed to `provide`. On the other hand, we may also specify what the system `uses` by saying what hardware we are willing to pay for. The specification only gives requirements, it does not give any assertion that there exists an actual dependency between used and provided QoS (or, in other words, that the demanded system can indeed be implemented). The specification with regions may be turned into a specification with explicit dependencies once we are working at the level of individual components. Composing these individual specifications may then give us an explicit specification of the QoS dependencies of the complete system which allows to check whether the required QoS can indeed be provided.
While the distinction by development process is important, the key factor is given by the context of specification as indicated above.
It is easily conceivable, that both styles of specification need to be mixed in a single specification document, to use the best of both worlds. It can even be imagined to mix the styles in the specification of a single component, such that a component's profile is split into various regions, some of which are specified as before, using `uses` and `provides` clauses, while others specify explicit dependencies of offered and used QoS. A region that is described using explicit dependencies would consist of two parts:

1. A `uses` clause specifying the conditions under which the component works in this region.
2. Equations and inequalities that provide the explicit dependencies.

3 Proposed Solutions

In section 2 we have argued that specification of QoS offers and requirements by regions is not always appropriate. In this section we give some ideas on how QoS dependencies could be expressed explicitly.
The solution that comes to mind immediately is to describe dependencies by writing down equations which map the QoS used into the QoS provided by a component. Staying with our example, we could imagine to rewrite the component's profile using a new clause `qos_dependency`:

```
profile response_times for ImageStreamEncoder {
  qos_dependency
    response_time (encodedImages.getNextImage) =
      response_time (unencodedImages.getNextImage) + 5;
}
```

However, this only works, if we can easily give such a precise dependency, that is, we know this exact factor and the ImageStreamEncoder component always shows this exact, deterministic behaviour. Very often it will be the case that we have only a rough idea of the kind of dependency, but may not know the precise factors. In our example, the time needed

to encode the image may well depend on the actual unencoded image – its complexity, color depth etc. – but also on the kind of encoding performed. But maybe we can say something more specific about the interval in which the provided QoS will be, if the usable QoS has a certain value.
Interval analysis (sometimes also called *interval computations*, [4],[5]) provides a way to handle intervals instead of exact values as parameters or results of functions. We could make use of these concepts to describe QoS-dependencies where some part is known precisely, while for some remainder we only know the possible range of values. For example, to state that the time it takes the ImageStreamEncoder to encode an image is the time it takes to get the next unencoded image plus a varying "encoding time" of 5 to 10 milliseconds, we could write:

```
profile response_times for ImageStreamEncoder {
  qos_dependency
    response_time (encodedImages.getNextImage) =
      response_time (unencodedImages.getNextImage) + [5,10];
}
```

As a generalization, both dependency specifications containing intervals and those containing only discrete values can be interpreted in an interval computational way by replacing all explicit values x by the interval $[x, x]$. This means that the statement above is valid independent wether `response_time (unencodedImages.getNextImage)` evaluates to a discrete value or to an interval itself. However, it is clear that having a discrete value enables us to calculate a sharper interval as response time for `ImageStreamEncoder`. Closely related to this is the question how to deal with open intervals expressing knowledge like for example "the response time is always greater than 5 milliseconds".
An even more powerful and expressive way compared to using intervals is to not state the interval explicitly, but to use two functions to specify an upper and a lower bound for the dependency, as follows:
Let $\underline{f}$ and $\overline{f}$ be two partial functions with $\operatorname{dom}\underline{f} = \operatorname{dom}\overline{f} \subseteq X_1 \times \ldots \times X_n$ where each X_i is a QoS characteristic domain (e.g. the quality characteristic `response_time`) and

$$\forall\,(x_1, \ldots, x_n) \in \operatorname{dom}\underline{f} : \underline{f}\,(x_1, \ldots, x_n) \leq \overline{f}\,(x_1, \ldots, x_n)$$

Then we call $F = \left(\underline{f}, \overline{f}\right)$ a *dependency function* with $\operatorname{dom} F = \operatorname{dom}\underline{f}$. The function evaluates to the closed interval between $\underline{f}\,(x_1, \ldots, x_n)$ and $\overline{f}\,(x_1, \ldots, x_n)$:

$$F\,(x_1, \ldots, x_n) = \left[\underline{f}\,(x_1, \ldots, x_n)\,, \overline{f}\,(x_1, \ldots, x_n)\right]$$

We then can use such dependency functions, for example:

```
profile response_times for ImageStreamEncoder {
```

```
  qos_dependency
    response_time (encodedImages.getNextImage) =
      [response_time (unencodedImages.getNextImage) + 5,
       response_time (unencodedImages.getNextImage) + 10
                * image_size(unencodedImages.getNextImage)];
}
```

An advantage of this approach is, that $\underline{f}$ and $\overline{f}$ do not need to run in "parallel".[4] As can be seen in the example we now can express the fact, that the encoder component has a minimum execution time of 5 milliseconds (independent of the size of the actual image to be compressed), but for the upper bound the size of the actual unencoded image has to be considered, too (we assume an operator `image_size` to exist and to denote the size of the parameter image). We would not have been able to state this dependency with just one function as in the second example.
So far, we have given examples for one provided quality characteristic only. What happens if the provided QoS affects more than one characteristic is another interesting issue. In addition to the relationship between offered and used QoS of the component there may now also exist dependencies between different characteristics of the offered QoS: For example, for an operation computing the sine of a value there may be two options: (a) highly accurate result at the price of longer execution time or (b) less accuracy of the result balanced by a shorter execution time. How to express this kind of dependency most efficiently remains to be looked into. An interesting question in this context is whether to give dependencies for all characteristics of the provided QoS in one statement or to provide individual statements dealing with one provided characteristic at a time.

4 Conclusion

We have argued that specifying QoS offers and requirements by means of `provides` and `uses` clauses is not always appropriate. Sometimes it is more natural to specify the dependency explicitly by giving an equation or inequality which relates offered and used QoS directly.
Depending on how precisely the relation between offered and used QoS is known there are several levels of how to express it. The most precise is by giving an explicit equation, which assigns one value of the offered QoS characteristic to each value of the used QoS characteristics. If this precision cannot be achieved, the next weaker option is to use interval computations to express the parts of the relation expression for which only intervals are known. Another option would be to use two functions one for the upper and one for the lower bound. The weakest way of specifying QoS dependencies is by using `uses` and `provides` clauses. Although the given examples had a rather simple structure, the benefit of using explicit specification should be obvious. Explicit specifications

[4] i.e., it is not required to hold that $\overline{f}\,(x_1, \ldots, x_n) - \underline{f}\,(x_1, \ldots, x_n) \equiv const$

will be of even more use if they incorporate parameters of the methods specified (e.g. the image size of the images to be encoded by our `ImageStreamEncoder` component).
So far, we have motivated and proposed a syntax for explicitly specifying QoS dependencies. It remains to precisely define the semantics of such constructs and to research how using them influences QoS management systems. Another field of research is to look into actual applications of the concepts trying to validate the hypothesis that they help making specifications clearer and easier to use.

References

1. Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems.* PhD thesis, University of Oslo, 2001.
2. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.
3. C[liff] 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.
4. Vladik Kreinovich, Daniel J. Berleant, and Misha Koshelev. Interval computations website. URL: http://www.cs.utep.edu/interval-comp/.
5. R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs N. J., 1966.
6. Christoph Pohl and Steffen Göbel. Integrating orthogonal middleware functionality in components using interceptors. In *Proc. Kommunikation in Verteilten Systemen (KIVS'03, Leipzig)*, Leipzig, Germany, February 2003. To appear in Informatik Aktuell, Springer.
7. Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley Publishing Company, November 1997.