Modeling of Non-Functional Contracts in Component-Based Systems using a Layered Architecture

Simone Röttger, Ronald Aigner

Technische Universität Dresden
Fakultät Informatik
email: {Simone.Roettger, Ronald.Aigner}@inf.tu-dresden.de

1 Introduction

In an enterprise environment multimedia and other real-time applications are only competitive with an appropriate support for Quality of Service (QoS). By QoS we refer to non-functional properties such as performance, reliability, timing, quality of data and security. To guarantee satisfactory Quality of Service a component of a system must be QoS aware so that it can communicate its expected QoS and its provided QoS to other components. Since resources vary, a component cannot be built to operate with a fixed level of available resources. The underlying software system needs to include special services such as negotiation, reservation of resources, monitoring actual QoS based on currently available resources and adaptation to changes in available resources.

Important industrial component technologies do not support the specification and enforcement of non-functional properties. The COMQUAD-Project[1] wants to add some approaches to this point. In the project a system architecture for supporting and a methodology for developing components with non-functional contracts will be developed. This project is interdisciplinary in the sense that it involves various specialists from different areas of computer science. The involved research groups are software engineering, operating systems, security, networks, databases and multimedia.

[1] COMponents with QUantitative properties an ADaptivity startet at October 1, 2001 at Technische Universität Dresden and Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, funded by DFG

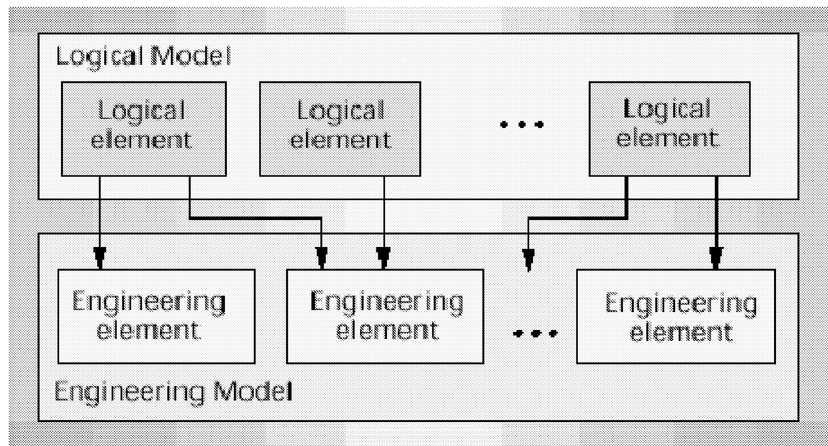

Figure 1: Realisation mapping [4]

This position paper illustrates the results of the discussion on modeling non-functional contracts. We give arguments that the modeling of non-functional contracts has to be seen in conjunction with the system architecture, especially a layered architecture. Further, we investigate the specification of resources.

2 Architecture

In this paper the widely-used definition of a component published by Szypersky [5] forms the basic understanding of a component: "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."

To guarantee non-functional properties in a component based system, this definition needs to be specialised. The interfaces enclose mechanisms of communication, e.g. interfaces to process sequences of events in real-time. For them the definition of QoS-parameters is significant. Furthermore, a component has to specify precisely its needs on resources to guarantee the offered QoS. We will investigate these two points in more detail. To react dynamical at runtime to possible changes of the environment, e.g. availability of resources or system configurations, the component system offers adaptation mechanisms. COMQUAD components are in the last three points different to todays industrial component technologies.

To model guaranteed quantitative properties we use a layered architec-

ture. A layered architecture is a system containig multiple, strongly separated layers with a hierachical relationship to manage complexity. This relationship is asymmetric with only one-way dependencies because the implementation of the upper layers depends on the lower layer but not the other way. The lower layer can be seen as an abstract virtual machine and the upper layer represents the program that drives the operations of this virtual machine [3]. In [4] this idea is further developed. To model QoS-contracts Selic distinguishes between the logical model and the engineering model. The logical model abstracts the details of how the components are actually realized. Only the engineering model describes how a particular technology implements the logical elements (fig. 1). The mapping between these two models describes a QoS-contract because the engineering elements are resources and the logical elements are their clients. If we apply this approach to the layered architecture described before, the lower layer can be seen as the engineering model for the upper layer. Therefore we can handle (specify, negotiate, allocate) resources between two layers.

2.1 Horizontal and Vertical Dimension in a System

There is a difference between the layered direction and the horizontal structural relationship. The latter defines a peer relation between two communicating components. In this dimension we define an export interface which specifies all offered functionality and an import interface which specifies all functionality used of this component. Additional to this functional description we can specify non-functional properties.

As an example consider a component which controls a process. Assuming the control of this process depends on measured values, we need another component, which manages a sensor. These two components have a symmetric communication relationship and belong to the same abstraction layer. The sensor offers a functional interface providing a method for reading the sensor data (eg. getData()). To this interface we can add a maximum delay time, which is a non-functional property.

In contrast, the vertical dimension describes the relationship between two abstraction layers. As we pointed out before, the lower layers are needed for the execution of the upper layer. Hence we can denote this as the resource dimension. As resources we can model not only resources of the operating system (eg. CPU and memory) and network resources (eg. bandwith), we can also model devices and components as resources. For a component it is transparent if a used resource belongs directly to the underlying layer or to an abstraction level further down in the hierarchy. The component can use this resource and can allocate it (resp. the container allocates it).

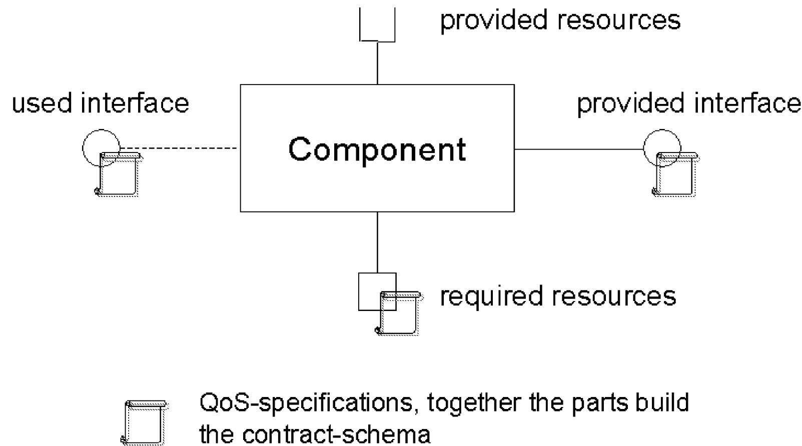

Figure 2: Component with interfaces

To meet the selfdescription aspect of a component we add in the vertical dimension two more interfaces to the component model. Downwards we specify an interface describing the required resources und upwards we specify an reservation interface for the offered resources (fig. 2). To the first one non-functional properties can be added, the second one is a purely functional interface for resource reservation.

The two dimensions depend on each other. In the peer dimension (horizontal) we can specify non-functional properties. These properties are mapped to the underlying resources. The simplest case is a tablebased mapping. In the majority of cases complex mapping functions and heuristics are necessary resp., due to many different parameters influencing these mapping. The properties described in the peer dimension can only be guaranteed if the resource contracts in the layered direction are fulfilled.

Remembering the example, we can only guarantee the provided maximal delay time, if we have enough CPU-time at the right moment. Additionally we need memory to buffer the data.

3 Contract Model

Precise specifications of requirements on other components as well as precise specifications of needed resources are preconditions of contracts. There exist some approaches for quality specification languages in distributed environments. But these languages primary provide support for the specification of non-functional properties as profiles to interfaces in a single abstraction

layer. In a component-based development process it is more concise to join all profile specifications to one contract-schema. The contract-schemes consists of three parts as we described before. These parts are the specification of offered non-functional aspects, of the required non-functional aspects and of the resources. To each component one or more contract schemes can be added. It is also important that the contract schemes are described independently of the functional interfaces to guarantee reuse [2].

For writing such contract-schemes CQML [1] particularly meets the demands specific to components, e.g. selfdescription of QoS-properties and contract specification. CQML is a language for precise specification of QoS. It enables specification at different levels of abstraction at design time.

CQML provides constructs to describe requirements on other components (`<uses>`) and offers to other components (`<provides>`). To handle the specification of resources we extend this approach trough the keyword `<resources>` under perpetuation of the grammatical structures. In this paper, we will not go in to details on the language support any further due to limited space.

In CQML an appropriate representation is missing to manage quantitative contracts at runtime. Hence we use CQML for specifying non-functional properties at design time. When deploying the components, a XML-representation is generated.

3.1 Resource Specification

As a result of the above discussion we have to describe the resources. Basically all resources can be described using a name (or another unique identifier) and a set of properties or attributes. A property of a resource can then be specified as a name-value pair. Here the name describes the property and the value specifies the quantity. We think this is sufficient to describe all resources. As an example, we will develop the specification for memory as a basic resource using CQML.

The quantitative property which is associated with memory is its size. This can be extended by introducing the factor of time into the description. We have to specify how much memory is needed at which times. We could only specify the maximum amount of memory an application might use. But it can be insufficient to reserve the whole amount of memory for the complete execution time of this component. Therefore we use a distribution function to optimize the usage of memory for utilization.

An additional property is whether the memory is pinned or not. This can also be transferred into a quantitative property. The difference between pinned and not pinned memory is that the first one may be swapped to disk.

Thus, access to unpinned memory might result in a pagefault which initiates a read of the swapped page. We can describe these with the time needed to access the memory. This is the same as the latency between the start of the access operation and its completion. When pinning memory we specify that a memory access will need a certain amount of time.

Regarding these descriptions we have two properties related to memory: the distribution function describing the size over time, and its response time.

```
quality_characteristic memory {
    size;
    access_delay;
    }

quality_characteristic size {
    domain: decreasing numeric kilobyte;
    minimum;
    maximum;
    average;
    }

quality_characteristic access_delay {
    domain: decreasing numeric milliseconds;
    }
```

3.2 Example for a Contract

For the sensor described before we could now define the following contract-schema:

```
quality sensor_delay{
        maxDelay < 30;
        }

quality memory_high{
        memory.size.minimum > 200;
        memory.size.maximium < 500;
        memory.size.average = 250;
        memory.access_delay = 10;
        }

profile fast_support for Sensor {
        provides {
             sensor_delay;
             }
        resources {
             memory_high;
             }
        }
```

We could define more profiles and specify transitions between these profiles to handle adaption. To simplify the example, we assume we have only one profile. Therfore we can write for the contract schema[2].

```
contract_schema for Sensor {
        fast_support;
        }
```

4 Development of Contracts

To create a contract scheme for a component we have to go through several steps in a development process. The first one is the specification of the functional component interfaces.

The second is the specification of the quantitative properties in one layer. To find the mapping to the underlying resource layer we plan the implementation of a reference container, which can be used to measure the actual resource usage of a component at run-time. The reference container monitors and protocols the usage of other components and resources. A disadvantage is that all components used have to be available on the reference container. The reference container can also be instrumented to simulate contention situations or high usage of a component. This way the optimal, maximal and minimal quantitative properties of resources can be measured.

The reference container does not have the capabilities to produce quantitative properties for all resources for all future platforms, e.g. the CPU usage of a component may vary if it is running on an 800 MHz Pentium II processor or on a 1.2 GHz Pentium 4 processor. Thus, the reference container can only generate reference values. During the deployment of the component the target container has to translate the reference values into specific values.

At deployment time the target container has no knowledge about specific run-time requirements of the component, e.g. the resource usage of a video-player varies for different videos and run-time properties. If a user requires a high quality video the player requires more resources than for a low quality video. Thus, the contract scheme might contain unresolved reference values after the deployment as well. Only after all reference values have been resolved we have an instantiated contract scheme, which is the final non-functional contract between two components.

The above discussion shows that a non-functional contract is not a static construct. In contrast it is a highly dynamic construct, especially at runtime.

[2] This is also an additional construct we defined in the project. It is not part of CQML. In CQML profiles can be nested in profiles, we use contract_schema instead.

5 Conclusion

This position statement shows that a layered approach is a useful instrument to model non-functional properties and contracts. We gave arguments that we have to extend the common contract descriptions. To guarantee non-functional properties we need a precise specification of required resources. Hence our contract model consists of three parts (provides, uses and resources) to handle all aspects of non-functional properties.

As an example we showed that resources can be described using a unique identifier and a set of properties or attributes and that a property of a resource can be specified as a name-value pair.

Further, we pointed out that we have to distinguish between the contract schema provided by a component and the contract between two or more deployed components. We need the former one to satisfy the reqirement that components have to be selfdescript. The latter one is a runtime instance.

At this point it seems reasonable to mention a disadvantage of layers. In many cases non-functional aspects are defined in realtime applications and to guarantee these properties realtime systems have to support these applications. In this environment layers are discussable. For developing and maintenance aspects layers are very powerful but at runtime every layer transition needs time.

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. *Computer*, 32(7):38–45, July 1999.

[3] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.

[4] Bran Selic. A generic framework for modeling resources with UML. *Computer*, 33(6):64–69, June 2000.

[5] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, November 1997.