CQML$^+$: Enhancements to CQML

Simone Röttger and Steffen Zschaler

Dresden University of Technology
Dresden, Germany
{Simone.Roettger, Steffen.Zschaler}@inf.tu-dresden.de

1 Introduction

In the last few years component-based software development has achieved a great improvement in software engineering. Component models can be found mainly for graphical user interfaces (JavaBeans [12], ActiveX [2]) and server components (EJB [7], COM [10]). Using the component paradigm is well understood for independent application development. In contrast, the reuse of existing components is still difficult. For reusing components they need an exact specification of their functional and non-functional properties. The existing component models do not support this in a sufficient manner. E.g., in EJB the deployment descriptor mainly supports functional specification. There exist no concepts for handling all aspects of the component paradigm, in particular non-functional aspects. The COMQUAD-project[1] has the goal to investigate such concepts. COMQUAD develops a system architecture for supporting and a methodology for developing components with guaranteed non-functional properties.

This paper gives an overview of issues involved in the specification of non-functional properties in a component-based development. It can be separated into the specification at development time and the representation of non-functional properties at runtime. First, we have to identify all non-functional requirements on the system and all non-functional offers of the system. As part of the development process these requirements have to be separated into offers and demands

[1] COMponents with QUantitative properties and ADaptivity started on October 1, 2001 at Technische Universität Dresden and Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany; supported by German Research Council; see also www.comquad.org

of individual components. Moreover, we have to consider that each component needs resources to work properly. To actually guarantee non-functional properties this resource demand has to be specified. In order to express these properties we need a precise specification language for non-functional properties in addition to the functional specification of a component based system. Since this specification language is used at design time and is therefore to be used by human beings, the language must have properties like readability, understandability and traceability.

To process components with specified non-functional properties a special runtime system is needed. Such a runtime system has to include services such as negotiation, resource allocation, monitoring of actual QoS and adaptation to changes in resources or required properties of communicating components. For all these services the runtime system uses the information included in a runtime representation translated from the design-specific non-functional specification. In contrast to the specification language the runtime representation is only used by the machine. Therefore the needed properties are effectiveness, processability for machines and the concentration on aspects needed by the target system and the used machine.

Because the specification language is a formal language, a design time toolkit can be built, encompassing various tools which allow to perform different kinds of analysis and evaluation on a QoS specification. At the heart of the toolkit is a parser which performs checking of syntax and static semantics, and translates the textual representation into an in-memory form, which can be accessed by all other tools. Furthermore, specifications of individual components can be composed to derive specifications of (sub-)systems. This can be supported by tools, which select pre-produced components from a component repository, based on their QoS-specification and the needs of the application under development. It may also be possible to determine the resource needs of a complete application already at design time, based only on the specifications of its constituting components. Additionally, various types of code can be generated from the QoS-specification: a) From the specification of individual QoS characteristics can be generated monitoring code that can be inserted into the runtime enviroment to measure current values of the characteristic, b) From the complete specification can be generated a pre-processed XML-representation that is reduced to the information needed by the runtime system.

In this paper we want to explain the solutions of the COMQUAD-project for these specification issues. For better understanding we will use a simple example for explanations. Consider a component (`myController`) which controls a process. Assuming the control of this process depends on measured values, we need another component, which manages a sensor (`mySensor`). Sensor and control component use stream-based communication to exchange measurement and control data. The control component needs sufficient CPU and memory to prosess the data. The connection between the components must provide for sufficient throughput.

Our work is based on CQML [1], a specification language for components with QoS properties. CQML is the most developed specification language with an appropriate semantic for non-functional properties. However the language is still only based on QoS-relations between components. There is no possibility to specify the demand on resources for components. One language exists which also uses resource description, HQML [4]. Since HQML has a predefined set of characteristics focused on web-applications we cannot use it for our project, which is mainly focused on server-side component architectures.

Our paper is focused on languages for expressing non-functional properties in component-based systems. In a first chapter we will introduce the basic concepts of CQML. The next chapter describes problems we could not solve with this language and our proposed improvements. In the last chapter we explain the reasons for having an XML-based runtime representation.

2 CQML

In his thesis Aagedal defines the Component Quality Modeling Language (CQML) a specification language to describe Quality of Service (QoS) offers and requirements in component based systems [1]. Its terminology is based on the ISO QoS Framework [6].

The basic building block of a CQML specification is the quality characteristic. It represents an entity to be constrained by the specification. Part of the definition of the characteristic is a specification of how the current value of this characteristic could be determined in a running system. Examples for characteristics are delay, jitter, screen resolution, but also – in a different context – learnability. In a next step, quality statements are used to specify constraints of quality characteristics. Because both quality characteristics and quality statements are parameterized they allow for reuse of parts of the specification in different contexts.

The specification is completed by associating the quality statements with components of the system. For this, CQML offers the concept of quality profiles. Here, the formal parameters of the quality statements are replaced by actual elements (e.g., operations, streams) of the component for which the QoS constraint is meant to hold. There are two ways in which a quality statement can be associated to a component: as a QoS offer or as a QoS requirement. To express this difference, CQML offers the two keywords `provides` and `uses`, resp. Profiles can contain multiple sub-profiles which are used as a means to express adaptivity. Finally, CQML specifications can be structured using quality categories (essentially name spaces).

Although CQML appears to be a very useful language for QoS specifications in COMQUAD we have identified a few shortcomings for which we want to propose improvements.

3 Proposed Improvements

3.1 Computational Model

The computational model in the original specification of CQML is rather vague. In particular, it is not very clearly stated whether concepts like `Flow` and `EventSequence` are part of the language definition or need to be defined as necessary with every use of the language.

The computational model essentially defines what types are allowable for the parameters of characteristics and qualities and what parts of the component model can be annotated by QoS constraints. It is important to notice that the semantics of a QoS contract changes immensely when we allow values from the application to be used in the constraint. When only values from the meta-model (e.g., `Operation`, `Flow`) are allowed in the constraints they can be checked statically. As soon as values from the application can be used a complete check of the constraints can only be done at runtime. Notice also that in this case, the contracts have to be rechecked potentially at every change of an attribute's value.

We therefore propose to make the computational model explicit and part of the language. Only types from the computational model should be allowed as parameter types in a CQML specification. The only exception are basic types like `Real`, `Boolean` etc. as long as the actual parameters are constants and do not depend on any part of the running system.

For this purpose we have created a meta-model of the computational model using UML [8]. The meta-model is based on the informal description given in Aagedal's thesis [1] together with some clarification especially in the area of operations and event equality, where the informal description is ambiguous. The meta-model consists of four main areas:

1. Streaming interfaces and stream-based communication
2. Operational interfaces and invocation-based communication
3. Resources and resource usage
4. Event mechanism

The main concepts for areas 1 and 2 can be seen in fig. 1. The resource meta-model is illustrated in fig. 2. The model shows components having provided and used interfaces, which can be either operational or stream based. An operational interface is essentially a collection of operations. Operation calls can be announcements (one-way) or interrogations (where the caller expects a response). A stream based interface is a collection of flow endpoints. Streams and flows, which connect stream endpoints and flow endpoints resp., model the actual connection over which data flows. The meta-model also defines events and event queues and defines which events can be observed for which meta-model element (e.g., a service emittance event gets put into the caller's event queue when an operation call is started), but these details cannot be included here for lack of space.

3.2 Resources Clause

For the specification of component-based systems with non-functional properties we need a description of the non-functional offers to and

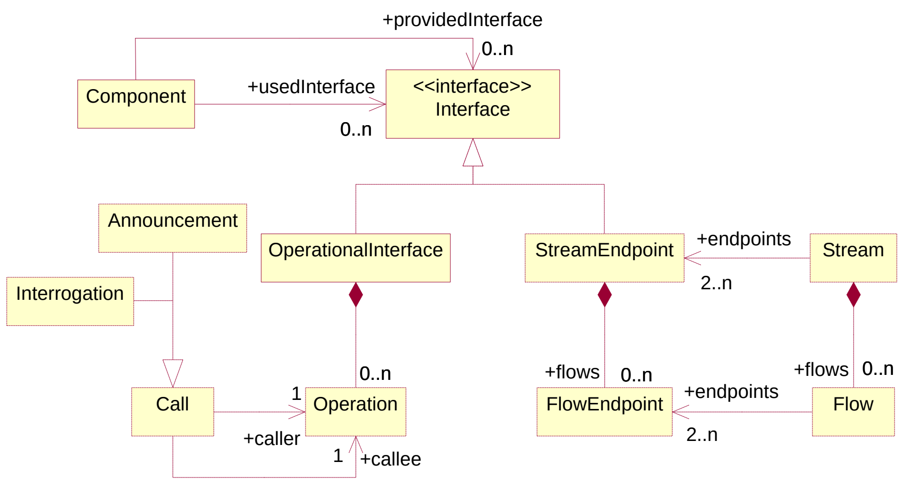

Fig. 1. Meta-model of main concepts of the computational model.

requirements on other components of the application additionally to the functional interface specification. It is also important to describe the requirements on the underlying system, in most cases the operating system and hardware platform [11]. The description of needed resources – e.g. the demand on CPU or memory – is a basic requirement in order to be able to guarantee non-functional properties. Notice, that there is a difference between the requirements on communicating components and on resources. Components can be fully managed by the container. In contrast, the container can request resources, but only the underlying system is able to actually manage (negotiate, reserve, allocate, monitor) resources. That means, the dependencies between component and component on the one hand and between component and resource on the other hand are differently processed by the system — therefore, the specification of resource demands must be separated from requirements on used components and as we will see later also needs a different syntax.

CQML provides one construct for describing non-functional properties provided by a component (`provides`) and one for properties needed from other components (`uses`). For specifying resource requirements we introduce a new meta-model for resources (fig. 2).

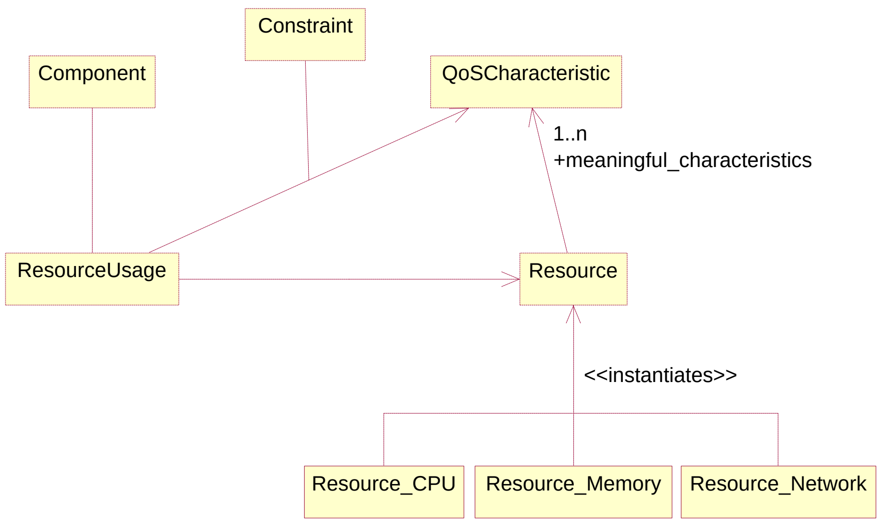

Fig. 2. Meta-model for resources and resource usage

Here resources are explicitly modeled as elements represented in the model as a type. In our view resources are models of real world resources such as CPU, memory or network. A resource has a name and a list of characteristics describing non-functional properties meaningful for this resource. However, the precise semantics of these characteristics is determined by the resource managers of the underlying system and the `quality_characteristic` definitions therefore contain no `values` clause.

Examples for such resources are `Resource_CPU`, `Resource_Memory` and `Resource_Network`, each of which describes a type of resource (e.g., CPU) rather than an actual instance (e.g., the first CPU of a two-processor-system). At runtime the container allocates specific resource instances according to the specified resource requirements of each component. Similar concepts and terminology can also be found in [9].

In our example we need to express memory requirements. To do so, we first define a resource memory, using the new keyword `resource`. The description of memory consists of two characteristics, one characteristic describes the required memory size and the other one the (maximum) time needed for accessing the memory. Specific resource requirements are expressed in quality statements.

```
resource memory {
  quality_characteristic size (r: Resource) {
    domain: numeric kilobytes;
  }

  quality_characteristic access_delay (r: Resource) {
    domain: numeric milliseconds;
  }
}

quality memory_high (r: Resource) {
  size (r).minimum > 200;
}
```

These resource requirements can be associated with a component using the newly defined `resources` statement in the component's profile:

```
profile good for myController {
  ...
  resources memory_high (memory);
}
```

Another example is the specification of required throughput on a communication channel (e.g., network connection). For this we could define a resource network with a characteristic `throughput`. It becomes clear immediately, however, that in a meaningful specification we need to be able to say which of the potentially many connections between one component and other components needs to be able to provide the specified throughput. Our concept of resource as a type of resource is not sufficient here. Instead, we need to be able to reify resource instances in the specification.

To this end, we further enhance the `resource` clause which declares a resource. We add a `with instances` clause listing those elements of the meta-model, instances of which can be viewed as modeling instances of the resource. For the network example this includes `Flow`, `Stream` and possibly also `Association`. We write:

```
resource network with instances Flow, Stream, Association {
  quality_characteristic throughput (ri: ResourceInstance) {
    domain: numeric real [0..) bytes/second;
  }
}
```

The parameter type also changes from `Resource` to `ResourceInstance`. This declaration would now allow us to specify ressource demands as follows:

```
quality good_throughput (ri: ResourceInstance) {
  throughput (ri) >= 10000;
}

profile good_communication for mySensor {
  resources good_throughput (channelToMyComponent);
}
```

Where `channelToMyComponent` identifies a `Flow` that transports measurement data between the sensor and the component.

Note that the two ways of declaring resources can be used in parallel. Which kind of declaration is most appropriate depends only on the kind of resource to be specified.

3.3 Structured Characteristics

In some cases a description of a composition of characteristics is useful. For example, we can describe CPU-demand by specifying a period and an execution time. Using CQML we can define a characteristic for period and for execution time. But we have no means of saying that only both together describe CPU-demand in a reasonable way and how they depend on each other.

CQML provides only three basic domain types. These are `numeric`, `enumeration` and `set`. To solve the discussed problem we introduce `tupel` as a new type. This construct allows us to structure characteristics. In the example, we can now define the CPU-demand as a structured characteristic consisting of the characteristics `period` and `execution_time`.

```
resource cpu {
  quality_characteristic cpu_demand (r: Resource) {
    domain: tupel {
              period (r),
              execution_time (r)
            };
    invariant: execution_time < period;
  }

  quality_characteristic execution_time (r: Resource) {
    domain: numeric real [0..) milliseconds;
  }

  quality_characteristic period (r: Resource) {
    domain: numeric real [0..) milliseconds;
  }
}
```

We can then use this characteristic in the following statement.

```
quality lots_of_cpu (r: Resource) {
  cpu_demand (r).period = 1000 and
  cpu_demand (r).execution_time = 999;
}
```

Notice how the parameter `r` is automatically handed down to the elementary characteristics. Also, because the two have been combined into one structured characteristic `cpu_demand`, the given invariant can now be meaningfully checked.

3.4 Explicit Dependencies

At present, CQML only allows to describe regions of acceptable QoS for a component. This is done by specifying a `profile` for each region. Such a profile contains a `uses` clause which specifies the required QoS and a `provides` clause which specifies the QoS the component offers. These regions are effectively examples that specify the component's behaviour in certain, but usually not all, situations and in a rather vague manner, by giving intervals in which required and offered QoS are located.

In some cases it may be more efficient to specify the relation between offered and used QoS directly by giving an equation or inequity that relates the two. In a separate paper ([13]) we explore the issues around this idea in more detail. There we propose to introduce a new clause `qos_dependency` which allows to write down such relations. We describe three different approaches for specifying the relation: explicit term, interval computational term, or using two functions to bound the relation. However, these concepts are not yet completely examined and need further research.

4 XML Representation

CQML has originally been developed as a specification language for use by human specifiers. Its syntax is therefore constructed with readability and reusability in mind. This has led to the introduction of some concepts that make life easy for human developers, but conversely make matters unneccessarily complex for the QoS management system, e.g.:

1. The use of names and namespaces (called quality categories) to identify entities of the specification. This is very suitable for human readers, but it introduces effects such as name hiding and overloading of the same name through the use of different parameter types. For a computer system globally unique identifiers are much easier to handle.
2. CQML allows for separation of concerns by allowing multiple profiles to be defined for one component. The implicit assumption is that they will need to be merged by the runtime system. This work could be done once, at design time, thus freeing the runtime system from the work load. There is also no reason why the runtime system should have to distinguish between simple and compound profiles (a distinction very helpful to the human designer). Simple profiles can be viewed, without loss of correctness, as compound profiles with just one child profile and no transitions.
3. CQML was built to enable reuse of parts of the specification. To this end, a specification is heavily structured into building blocks like `quality_characteristic`, `quality` and `profiles`. Once a specification is complete and deployed on the running system, there is no need for such extensive structuring. Especially removing the `quality`-layer may improve the efficiency of the QoS management system.

These points illustrate that a translation of the specification from the CQML representation into a different form for the runtime system is indeed useful. In the COMQUAD-project we are currently working to integrate QoS support into an Enterprise Java Beans (EJB, [7]) based runtime environment. In this environment all meta-information is stored in so-called deployment descriptors, which are written in XML. So we decided to translate the CQML specification into an XML representation as well. We have already designed an XML Schema [3] file defining the syntax of the XML representation.

5 Conclusion

In this position paper we made suggestions for handling all aspects of non-functional properties in a specification language together with the component paradigm. With the presented improvements we are able to actually describe the demand on resources of a component. Moreover, we are able to compose characteristics and specify constraints on a structured characteristic and describe dependencies between parts of it. Since we only allow types from the computational model as part of the specification it is possible to make semantic checks at design time. By providing an XML representation we make specification processing less complex for the QoS management system.

This work was carried out as part of the COMQUAD project. This project aims at a development methodology as well as a runtime support system for component-based applications with guaranteed Quality of Service properties and adaptation. Based on the Dresden Realtime Operating System (DROPS, cf. e.g. [5]) an Enterprise Java Beans based runtime enviroment is currently being developed. So far, we have defined an XML representation of the QoS specification and a resource management component in the operating system. Currently we are working to build both a compiler that translates our CQML specification into the XML representation and the actual runtime system that uses these XML documents.

References

1. Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems.* PhD thesis, University of Oslo, 2001.
2. David Chappell. *Understanding ActiveX and OLE.* Microsoft Press, 1996.
3. XML Schema Working Group. XML Schema w3c recommendation – parts 0–2. http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/TR/xmlschema-2/, May 2001.
4. Gu, Nahrstedt, Yuan, Wichadakul, and XuA. An xml-based quality of service enabling language for the web. Technical Report UIUCDCS-R-2001-2212, Department of Computer Science University of Illinois at Urbana-Champaign,Urbana, 2001.

5. H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proc. 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications (Sintra, Portugal, Sept. 1998)*, Sintra, Portugal, September 1998.
6. Information technology – quality of service: Framework. ISO/IEC 13236:1998, ITU-T X.641, 1998.
7. Sun Microsystems. Enterprise JavaBeans Specification, version 2.0. Final Release, August 2001.
8. Object Management Group. Unified Modelling Language Specification Version 1.4. OMG Document, September 2001.
9. Object Management Group. UML profile for schedulability, performance, and time specification. OMG Document, March 2002. URL http:// www.omg.org/ cgi-bin/ doc?ptc/02-03-02.
10. Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
11. Simone Röttger and Ronald Aigner. Modeling of non-functional contracts in component-based systems using a layered architecture. In *Component Based Software Engineering and Modeling Non-functional Aspects (SIVOES-MONA), Workshop at UML 2002*, October 2002.
12. Sun Microsystems. JavaBeans API Spezification, Version 1.01, July 1997.
13. Steffen Zschaler and Marcus Mayerhöfer. Explicit modelling of qos-dependencies. *to be published in Proceedings of QoS in CBSE workshop*, June 2003. Dresden University of Technology and Friedrich-Alexander-University Erlangen-Nürnberg.